Patchen von Modellen

Udo Kelter, Timo Kehrer, Dennis Koch
Praktische Informatik
Universität Siegen
{kelter,kehrer,dkoch}@informatik.uni-siegen.de

Abstract: Für die modellbasierte Softwareentwicklung werden spezialisierte Werkzeuge für ein professionelles Versions- und Variantenmanagement von Modellen benötigt. Insbesondere Anwendungsfälle wie das Patchen oder Mischen von Modellen stellen sehr hohe Anforderungen an die Konsistenz der synthetisierten Modelle. Während für das klassische 3-Wege-Mischen von Modellen bereits erste brauchbare Ansätze vorgeschlagen wurden, bestehen erhebliche Defizite bei Patch-Werkzeugen. Anhand realer Einsatzszenarien von Patch-Werkzeugen erörtern wir die wesentlichen Anforderungen und Schwierigkeiten und analysieren potentielle Fehlerquellen bei der Anwendung von Patches. Daraus leiten wir wesentliche Entwurfsentscheidungen zur Konstruktion von Patch-Werkzeugen ab und stellen unseren Ansatz zum konsistenzerhaltenden Patchen vor.

1 Einleitung und Motivation

Die modellbasierte Softwareentwicklung leidet nach wie vor an einer unzureichenden Unterstützung durch Versionsmanagement-Werkzeuge [EM12]. Während teilweise schon brauchbare Werkzeuge zum Anzeigen von Differenzen und Mischen von Modellen vorhanden sind, bestehen erhebliche Defizite bei Patch-Werkzeugen. Die verfügbaren Werkzeuge sind nicht annähernd so ausgereift und breit einsetzbar wie z.B. die UNIX-Standardwerkzeuge `patch` oder `diff`.

Dieses Papier behandelt das Patchen von Modellen. Patchen (*to patch* - ausbessern, flicken, korrigieren) bezeichnet ganz allgemein den Vorgang, ein Dokument durch Anwendung einer Änderungsvorschrift (*the patch* - der Flicken, die Korrektur) abzuändern. Ein Patch entsteht i.d.R. durch Berechnung der Differenz zwischen zwei Dokumenten; er besteht aus einer Sequenz von Editierschritten, die das erste Dokument in das zweite überführen. Abschnitt 2 definiert grundlegende Begriffe und grenzt das Patchen vom 3-Wege-Mischen ab.

Patch-Werkzeuge werden für mehrere Entwicklungsaufgaben benötigt, bei denen unterschiedliche Randbedingungen zu beachten sind (s. Abschnitt 3). Das Hauptproblem beim Patchen ist die Anwendung eines Patches auf ein anderes Modell als das, das beim Vergleich als Basismodell diente: hierdurch kann die Anwendung des Patches scheitern und das Modell so inkorrekt werden, daß es nicht mehr mit Modelleditoren verarbeitet werden kann. Abschnitt 4 analysiert diese Probleme und führt die in diesem Papier unterstellte Definition korrektheitserhaltender Editieroperationen ein.

Hauptbeitrag des Papiers ist ein Konzept zum korrektheitserhaltenden Patchen von Modellen. Es unterstellt einen Satz korrektheitserhaltender Editieroperationen auf Modellen und ein Modellvergleichsverfahren wie z.B. in [KKT11] vorgestellt, das Differenzen mit Hilfe dieser Editieroperationen darstellt. Für so gewonnene Patches wird ein Verfahren und zugehöriges Werkzeugdesign vorgestellt, wie ein Patch kontrolliert auf einem Zielmodell angewandt werden kann. Hierbei werden diverse Fehlerfälle behandelt und Möglichkeiten für manuelle Eingriffe angeboten. Details sowie eine beispielhafte Benutzung der prototypischen Implementierung des Patchwerkzeugs werden in Abschnitt 5 vorgestellt. Abschnitt 6 vergleicht das hier vorgestellte Design mit anderen Ansätzen und Abschnitt 7 faßt die wesentlichen Erkenntnisse zusammen.

2 Grundlegende Begriffe

Ein **Patch** ist eine halbgeordnete Menge von Editierschritten (die üblicherweise in einer mit der Halbordnung konsistenten linearen Ordnung notiert bzw. gespeichert wird).

Ein **Editierschritt** ist analog zu einem Statement in einem Programm ein Aufruf einer Editieroperation mit passenden Parametern, darunter immer wenigstens eine Referenz auf ein Dokumentelement. Die Menge der zulässigen Editieroperationen hängt vom Dokumenttyp ab und wird als dessen **Editierdatentyp** bezeichnet.

Die **Anwendung** eines Patches auf ein Dokument, das in diesem Zusammenhang auch als das **Zieldokument** (*target document*) bezeichnet wird, besteht darin, die in dem Patch enthaltenen Editierschritte auf dem Zieldokument auszuführen. Hierzu wird i.d.R. ein Interpreter benutzt, der Implementierungen der Editieroperationen beinhaltet. Bei der Anwendung eines Patches können diverse Fehler auftreten, die vom Einsatzszenario abhängen.

Erzeugt wird ein Patch normalerweise durch Berechnen einer Differenz zwischen zwei Dokumentversionen v0 und v1 und eine eventuelle Nachverarbeitung der initial gewonnenen Differenz[1]; v0 wird in diesem Fall auch als **Basisversion** bezeichnet, v1 als **geänderte Version**.

Bild 1 illustriert die Gewinnung und Anwendung eines Patches: zunächst werden die Dokumentversionen v0 und v1, die aus Repository 1 stammen, verglichen; die resultierende Differenz wird zu einem Patch verarbeitet. Der Patch soll nun auf das Zieldokument vt angewandt werden, das potentiell aus einem anderen Repository stammen kann. Hierzu wird vt in den Workspace 1 ausgecheckt, der Patch dort angewandt und das veränderte Dokument als Nachfolgeversion von vt wieder eingecheckt.

Das Patchen wird oft auch als “Mischen” bezeichnet, arbeitet aber deutlich anders als das 3-Wege-Mischen[2]. Das 3-Wege-Mischen ist in Bild 1 in Workspace 2 illustriert: gemischt werden die Version v1, die sich im Repository befindet, und die modifizierte Kopie von v0 im Workspace 2, hier als v2 bezeichnet; v1 und v2 haben beide v0 als gemeinsame Basisversion. Die beiden Differenzen diff(v0,v1) und diff(v0,v2) können Editierschritte

[1] Allerdings sind auch andere Methoden denkbar, z.B. eine manuelle Erstellung oder die Protokollierung von Änderungsoperationen in syntaxbasierten Editoren offener Entwicklungsumgebungen [SZN04, HK10]

[2] Auf das 2-Wege-Mischen gehen wir hier nicht ein.

beinhalten, die unverträglich sind, d.h. einer der Editierschritte muß “geopfert” oder manuell modifiziert werden. Paare derartiger Editierschritte, bei denen keine automatisierte Entscheidung möglich ist, werden als **Konflikt** bezeichnet.

Beim Patchen gibt es i.a. keine gemeinsame Basisversion, v0 und vt können völlig unkorreliert sein und in verschiedenen Repositories liegen. Der Begriff Konflikt ist daher hier nicht anwendbar; man könnte zwar die Differenz diff(v0,vt) bilden und hätte dann scheinbar die 3-Wege-Situation mit v0 als Basisversion, die Differenz diff(v0,vt) steht aber nicht zur Disposition, Editierschritte daraus können nicht geopfert werden. Ferner steht bei manchen Einsatzszenarien (s. Abschnitt 3) die Version v0 gar nicht für einen Vergleich zur Verfügung. Wenn überhaupt, können nur Editierschritte im Patch geopfert werden.

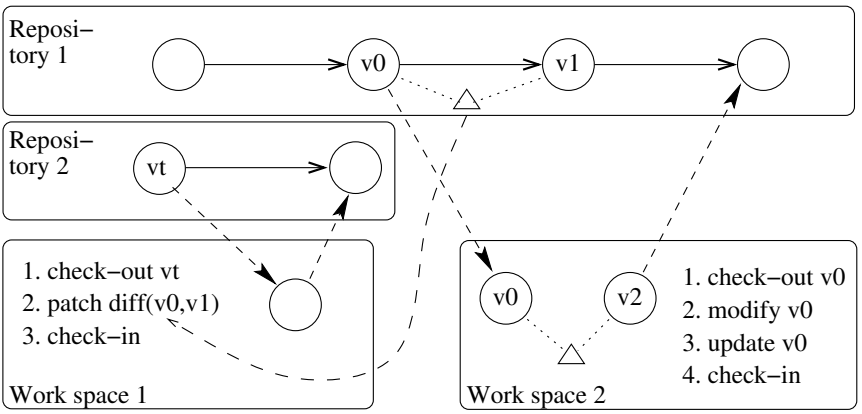

Abbildung 1: Patching vs. 3-Wege-Mischen

Wird ein auf diff(v0,v1) basierender Patch auf v0 angewandt, entsteht wieder v1, d.h. man kann mit Hilfe des Patches aus v0 die Version v1 rekonstruieren. Die Umkehrung gilt i.a. nicht, weil man aus einem Editierschritt nicht ohne weiteres den inversen Editierschritt ableiten kann. Daher werden Patches auch als **asymmetrische Differenz**, **directed delta** [Me02] oder **edit scripts** bezeichnet.

3 Einsatzszenarien von Patch-Werkzeugen

Das Patchen von Softwaredokumenten hat eine lange Tradition bei textuellen Dokumenten; die entsprechenden Einsatzszenarien ergeben sich analog für Modelle:

(1) Abgleich identischer Kopien eines Dokuments: Wenn z.B. ein Dokument auf mehreren Rechnern identisch vorhanden ist, müssen die Kopien im Rahmen eines Rollouts eines neuen Software-Releases auf einen neuen Stand gebracht werden. Man könnte theoretisch auch die kompletten Dokumente verschicken, Patches sind aber i.a. wesentlich kleiner; der Hauptvorteil ist daher die Reduktion des Transportvolumens. Ein äquivalentes

Einsatzszenario ist die Delta-Speicherung einer Revisionshistorie in einem Repository. Hauptmerkmal dieser Einsatzszenarien ist, daß der Patch immer nur auf eine Kopie seiner Basisversion angewandt wird und daher ohne Fehlerfälle abgearbeitet werden kann.

(2) "Cherrypicking changes" [CFP11]: Unterstellt wird hier typischerweise ein Versionsgraph wie in Bild 1 dargestellt. Es liegen zwei parallele Zweige vor, im unteren Zweig, in dem vt liegt, sollen selektiv die Änderungen, die im anderen Zweig zwischen v0 und v1 auftraten, übernommen werden. v0 und vt müssen nicht notwendig einen gemeinsamen Vorgänger haben, sondern können in verschiedenen Repositories liegen.

(3) Propagation von Änderungen in einer Menge von Varianten: In der Praxis werden vielfach Software-Systemfamilien als Mengen von autarken Varianten gehandhabt, namentlich in einer Anfangsphase, bevor Methoden des Product Line Engineering eingeführt werden. Eine Verbesserung in einer Variante, z.B. eine Fehlerkorrektur oder ein neues Feature, muß dann auf alle anderen Varianten propagiert werden. Technisch ähnelt dies stark dem Cherrypicking: die Verbesserung stellt sich als Patch dar, der durch Vergleich von Revisionen in der Ursprungsvariante gebildet wird und der auf *mehrere* Ziel-Dokumente angewendet werden soll.

4 Fehlerquellen bei der Anwendung von Patches

4.1 Identifikation von Operationsargumenten

Bei der Anwendung eines Patches müssen die Referenzen auf die Dokumentelemente aufgelöst werden, welche als Argumente von Editierschritten verwendet werden; jeder Referenz muß ein konkretes Element im Zieldokument zugeordnet werden.

Sofern ein Patch auf seine Basisversion oder eine Kopie hiervon angewandt wird (s. Abschnitt 3, Einsatzszenario 1), ist die Auflösung der Referenzen unproblematisch; als Referenzen können hier Zeilennummern, eindeutige Pfadnamen oder diverse andere Daten benutzt werden, mit denen die zu ändernden Dokumentelemente zuverlässig wiedergefunden werden.

Andernfalls (s. Abschnitt 3, Einsatzszenarien 2 und 3) steht man vor zwei gravierenden Problemen bei der Auflösung von Referenzen:

1. Ein referenziertes Dokumentelement ist ggf. nicht vorhanden, z.B. weil es gelöscht wurde. Der entsprechende Editierschritt ist somit prinzipiell **nicht ausführbar**.
2. Das referenzierte Dokumentelement befindet sich im Zieldokument an einer anderen "Position" als im Basisdokument; d.h. die im Patch angegebene Referenz, die letztlich ein Datenwert ist, der ein Modellelement identifizieren kann, muß als inkorrekt erkannt und wenn möglich korrigiert werden. Offensichtlich besteht hier die Gefahr **der Falschausführung von Editierschritten**, weil die Inkorrektheit der Referenz nicht er-

kannt wird oder der Korrekturversuch fehlschlägt und somit der Editierschritt an einem falschen Dokumentelement ausgeführt wird.

Die vorstehenden Fehlerquellen stellen schon bei textuellen Dokumenten ein Problem dar, bei Modellen werden sie durch die einzuhaltenden Korrektheits- bzw. Konsistenzkriterien wesentlich vergrößert.

4.2 Korrektheitsgrade von Modellen und korrektheitserhaltende Editieroperationen

Konzeptuell wird ein Modell als abstrakter Syntaxgraph (**ASG**) betrachtet, dessen Knoten, Kanten und Attribute durch ein Metamodell spezifiziert sind. Neben der elementaren Syntax spezifiziert das Metamodell oft zusätzliche, nicht-lokal wirksame Konsistenzbedingungen (z.B. mit Hilfe der OCL).

Die Metamodelle von Modellierungssprachen spezifizieren üblicherweise "weitgehend korrekte" Modelle, die zumindest so korrekt sind, daß man die übliche graphische Darstellung generieren kann. In der Praxis weichen viele Editoren von den Standards ab, sowohl durch zusätzliche Korrektheitskriterien wie durch nicht beachtete. Im Endeffekt entscheidend ist der minimale Korrektheitsgrad, den Modelle einhalten müssen, um überhaupt von Editoren weiterverarbeitbar zu sein. Patchwerkzeuge müssen sicherstellen, daß gepatchte Modelle diesen minimalen Korrektheitsgrad aufweisen und in dieser Hinsicht an die jeweiligen Modelleditoren in einer Entwicklungsumgebung angepaßt sein. Dieser minimale Korrektheitsgrad führt zu mehreren unangenehmen Konsequenzen:

1. *Nichttriviale minimale korrektheitserhaltende Editieroperationen*: Einzelne elementare (generische) Graphmodifikationen können aus Benutzersicht sinnlos sein und einen nicht mehr graphisch anzeigbaren, inkonsistenten Zustand erzeugen [KKT11]. Selbst wenn in den meisten Fällen elementare Graphmodifikationen erlaubt sind (z.B. das Setzen von Namen), treten Fälle auf, in denen ein korrektheitserhaltender Zustandsübergang nur durch mehrere elementare Graphmodifikationen bewirkt werden kann. *Minimale korrektheitserhaltende Editieroperationen* bestehen daher i.a. aus mehreren elementaren Graphmodifikationen, die analog zu einer Transaktion einen bisherigen konsistenten Zustand in einen anderen überführen und nur ganz oder gar nicht ausgeführt werden dürfen.

   Die zulässigen Editieroperationen müssen für jeden einzelnen Modellelementtyp individuell bestimmt werden. I.f. gehen wir davon aus, daß ein geeigneter Editierdatentyp bestimmt wurde.
2. *Zustandsabhängige Ausführbarkeit*: Eine korrektheitserhaltende Editieroperationen ist ggf. nur ausführbar, wenn bestimmte Bedingungen erfüllt sind[3]. Beispielsweise darf in einem Zustandsautomaten kein Startzustand erzeugt werden, wenn schon einer vorhanden ist, bei der Erzeugung einer Vererbungsbeziehung in einem Klassendiagramm darf

[3] Im Gegensatz dazu sind Editieroperationen auf Texten immer ausführbar, da es (normalerweise) keine Vorbedingungen oder Konsistenzkriterien gibt.

kein Zyklus entstehen usw. Die Ausführung einer Editieroperation kann also abhängig vom Modellzustand scheitern.

Durch scheiternde Editieroperationen entsteht (neben fehlenden Argumenten) eine neue Kategorie von Ursachen, warum einzelne Editierschritte eines Patches auf einem Zielmodell nicht ausführbar sein können.

5 Korrektheitserhaltendes Patchen

In Abschnitt 4 wurden mögliche Fehlerquellen analysiert, welche bei der Anwendung eines Patches auf ein von seiner Basisversion abweichendes Zielmodell auftreten können. Insbesondere bei der Propagation von Änderungen in einer Systemfamilie ist für jedes der Zielmodelle im Einzelfall zu prüfen, ob die Änderung hier sinnvoll und zulässig ist, speziell bei sicherheitskritischer eingebetteter Software. Daher wird man hier Wert auf Werkzeugfunktionen legen, mit denen man (a) die Auswirkungen eines Patches kontrollieren kann[4], (b) bei Bedarf den Patch sinnvoll modifizieren und in dieser Form archivieren kann und (c) die Anwendung des Patches steuern kann, z.B. durch manuelle Vorgaben, wie Referenzen auf Operationsargumente aufzulösen sind. Derartige Werkzeugfunktionen können auch beim Einsatzszenario Cherrypicking sinnvoll sein, werden aber von derzeit verfügbaren Werkzeugen nicht angeboten.

Das hier vorgestellte Konzept basiert auf vier separaten Werkzeugen bzw. integrierten Werkzeugfunktionen: einem Modellvergleichswerkzeug, das asymmetrische Differenzen auf Basis korrektheitserhaltender Editieroperationen erzeugt (s. Abschnitt 5.1), einem erweiterten Differenzanzeigewerkzeug zum Editieren von Patches (s. Abschnitt 5.3), der Matching-Funktion eines Modellvergleichwerkzeugs zur Auflösung von Referenzen auf Modellelemente im Zielmodell (s. Abschnitt 5.4) und einem interaktiven Werkzeug zur kontrollierten Anwendung von Patches (s. Abschnitt 5.5).

Unsere prototypische Implementierung basiert auf dem Eclipse Modeling Framework. Als Transformationstechnologie wird das auf Graphtransformationstechniken beruhende Modelltransformationssystem EMF Henshin [Ar10] verwendet.

5.1 Bilden von Patches

Erzeugt wird ein Patch durch Vergleich des Ursprungsmodells mit dem geänderten Modell. Gängige Verfahren [KRPP09] liefern im Kern jedoch *symmetrische* Differenzen [Me02], d.h. eine Menge von Korrespondenzen, die "gleiche" Elemente der beiden Modelle einander zuordnen. Indem eines der Modelle zum Basismodell erklärt wird, werden Modellelemente, die im anderen Modell nicht mehr bzw. neu vorhanden sind, als *gelöscht* bzw.

[4] Die Auswirkungen eines Patches kann man ggf. kontrollieren, indem man das Zieldokument in einem Workspace auscheckt, den Patch anwendet und das modifizierte Dokument mit dem Original vergleicht. Diese Notlösung scheitert aber, sofern der Patch nicht erfolgreich anwendbar ist.

erzeugt angesehen. Hierdurch entsteht eine *asymmetrische* Differenz vom Basismodell zum geänderten Modell, welche jedoch beliebig viele "low-level" Editierschritte enthalten kann, die potentiell zu inkonsistenten Zwischenzuständen führen.

Wir nutzen hier das in [KKT11] vorgestellte Verfahren, welches in einem Folgeschritt Gruppen von low-level Änderungen identifiziert, die korrektheitserhaltende Editieroperationen realisieren (s. Bild 2). Der in [KKT11] vorgestellte regelbasierte Ansatz garantiert ferner die Konsistenz der erkannten Editierschritte und der definierten Editieroperationen, da für die Operationserkennung benötigte Erkennungsregeln automatisch auf Basis von Editierregeln generiert werden. Es wird also kein Aufruf einer Editieroperation erkannt, welche nicht definiert ist.

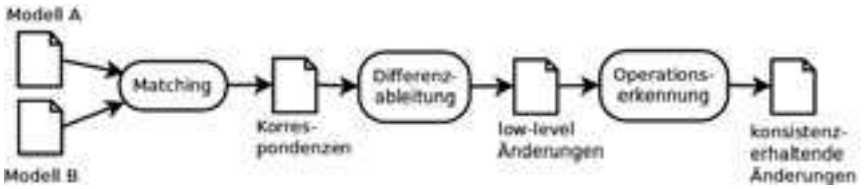

Abbildung 2: Struktur zustandsbasierter Differenzberechnungen

Für die Zwecke der Patchbestimmung wurde dieses Verfahren an mehreren Stellen erweitert, die für eine reine Differenzdarstellung nicht notwendig sind: (a) die Spezifikation der Editieroperationen wurde um Ein- und Ausgabeparameter ergänzt; (b) beim Bilden der Gruppen werden nun den formalen Parametern konkrete Werte, insb. Referenzen auf Modellelemente, zugeordnet; (c) Abhängigkeiten zwischen Editierschritten, die durch Ein- und Ausgabewerte entstehen, werden verwaltet.

5.2 Repräsentation von Patches

Unabhängig von dem zur Berechnung asymmetrischer Differenzen eingesetzten Verfahren ist von entscheidender Bedeutung, daß ein Patch bei Bedarf modifiziert und in dieser Form archiviert werden kann. Aus werkzeugtechnischer Sicht wird hierdurch ein Patch zu einem *editierbaren Dokument* und muss in geeigneter Form repräsentiert werden.

In der Literatur vorgeschlagene Datenmodelle zur Repräsentation von Modelldifferenzen entstammen dem Kontext des zustandsbasierten Vergleichs und basieren meist auf low-level Änderungen [CRP07, RV08], welche keinerlei Konsistenzkriterien beachten. Die Gruppierung von low-level Änderungen zu konsistenzerhaltenden Editierschritten ist zwar in wenigen Datenmodellen für Differenzen vorgesehen [KKT11, Ko10], Abhängigkeiten zwischen Editierschritten sowie aktuelle Aufrufparameter der entsprechenden Editieroperationen werden jedoch nicht explizit modelliert.

Ein Datenmodell zur Repräsentation asymmetrischer Differenzen EMF-basierter Modelle, welches den Anforderungen für das korrektheitserhaltende Patchen genügt, ist in Bild 3 dargestellt. Eine asymmetrische Differenz (*AsymmetricDifference*) zwischen einem Basis-

modell und einem geänderten Modell (jeweils repräsentiert durch ein EMF *ResourceSet*) besteht aus einer Menge von Editierschritten (*OperationInvocation*), welche durch eine Menge zyklenfreier Abhängigkeitsbeziehungen (*Dependency*) halbgeordnet wird.

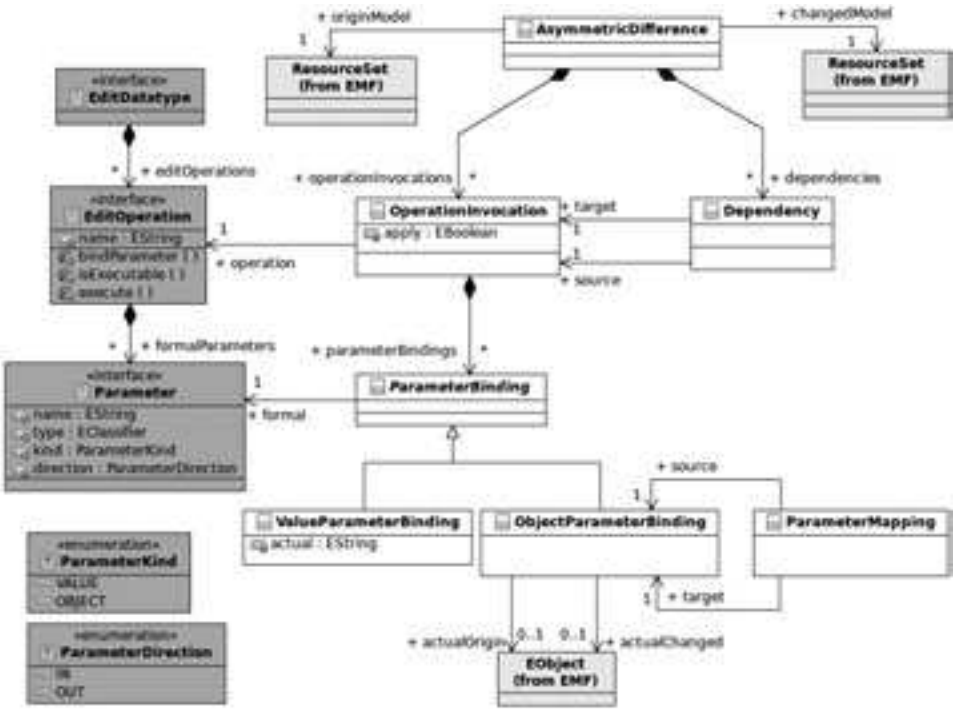

Abbildung 3: EMF-basierte Repräsentation von Patches

Einem Editierschritt wird die Signaturdeklaration der aufgerufenen Editieroperation (*EditOperation*) des zugrundeliegenden Editierdatentyps (*EditDatatype*) zugeordnet. An jeden formalen Parameter (*Parameter*) einer Editieroperation werden die aktuellen Parameterbelegungen gebunden (*ParameterBinding*): An Wertparameter (*kind = ParameterKind::VALUE*) wird die String-Repräsentation des entsprechenden Datenwerts gebunden (*ValueParameterBinding.actual*). An Objektparameter (*kind = ParameterKind::OBJECT*) werden Objekte der internen Repräsentation (*EObject*) des Basismodells (*ObjectParameterBinding.actualOrigin*) bzw. des geänderten Modells (*ObjectParameterBinding.actualChanged*) gebunden.

In einem Editierschritt geänderte Objekte existieren in beiden Modellversionen, gelöschte Objekte existieren lediglich in der Basisversion und erzeugte Objekte existieren nur in der geänderten Version. In einem Editierschritt erzeugte Objekte sind Argumente formaler Ausgabeparameter (*direction = ParameterKind::OUT*). In einem Editierschritt verwendete Objekte sind Argumente formaler Eingabeparameter (*direction = ParameterKind::IN*). Ausgabeargumente, welche in einem späteren Editierschritt als Eingabeargumente verwendet werden, werden entsprechend aufeinander abgebildet (*ParameterMapping*).

5.3 Editieren von Patches

Die Modifikation eines Patches vor seiner eigentlichen Anwendung besteht i.a. darin, den Patch auf eine Teilmenge der enthaltenen Editierschritte zu reduzieren. Die wesentliche Herausforderung besteht darin, daß der modifizierte Patch bei seiner Anwendung auf das Zielmodell korrektheitserhaltend ist.

Zum Editieren von Patches wird unterstellt, daß Ursprungsmodell und geändertes Modell zur Verfügung stehen. Ein Werkzeug zum Editieren von Patches kann somit auf Basis eines bestehenden Anzeigewerkzeugs für Differenzen realisiert werden, sofern dieses die Darstellung nicht-trivialer Editierschritte unterstützt, so z.B. das Darstellungskonzept der klickbaren Liste lokaler Unterschiede [KKOS12, KSW08]. Diese Anzeigeform ermöglicht die visuelle Darstellung des Kontexts eines Editierschritts, d.h. die aktuellen Parameterbelegungen. Dies ist insbesondere dann entscheidend, wenn "anonyme" Modellelemente, so z.B. arithmetische Operatoren in ASCET- oder Matlab/Simulink-Blockdiagrammen, als Operationsargumente verwendet werden.

Bild 4 zeigt die grafische Bedienschnittstelle unseres prototypischen Editors für Patches. Das Ursprungsmodell sowie das geänderte Modell werden jeweils in einem eigenen Editorfenster angezeigt. Das Beispielszenario zeigt zwei Revisionen eines in Ecore modellierten Entwurfsklassendiagramms für ein exemplarisches Flugbuchungssystem.

In einem Steuerfenster wird die klickbare Liste der in der Differenz enthaltenen Editierschritte angezeigt. Ein Editierschritt ist der angewendeten Editieroperation entsprechend benannt. Die lineare Anordnung der Editierschritte ist konsistent zur Halbordnung, welche aus den Abhängigkeiten der Editierschritte resultiert. Die von einem Editierschritt abhängigen Schritte können auf Wunsch visuell hervorgehoben werden. Die einzige Abhängigkeit im Beispiel aus Bild 4 besteht zwischen dem Erzeugen des Attributs *birthday* und dem anschließenden Setzen des Attributtyps auf den Datentyp *EDate*.

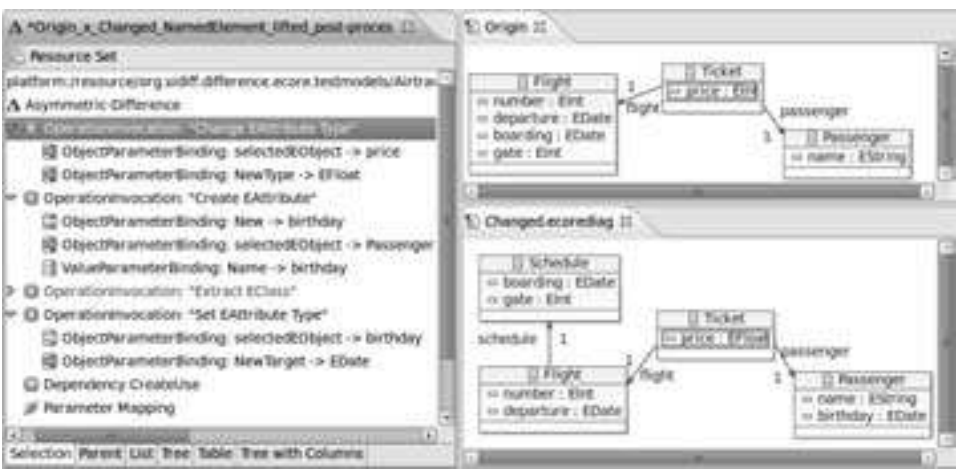

Abbildung 4: Grafische Bedienschnittstelle zum Editieren von Patches

Ferner werden die aktuellen Aufrufparameter der einzelnen Editierschritte im Steuerfenster dargestellt. Objektparameter werden explizit als Eingabe- bzw. Ausgabeparameter gekennzeichnet, wobei aufeinander abgebildete Ein- und Ausgabeparameter farblich markiert werden, so im Falle des zuerst erzeugten und später verwendeten Attributs *birthday*. Wird ein Listeneintrag auf der linken Seite angeklickt, so werden die involvierten Modellelemente, d.h. die aktuellen Parameter eines Editierschritts, in den entsprechenden Editorfenstern fokussiert und farblich hervorgehoben.

Per Kontextmenü kann ein Benutzer einzelne Editierschritte aus dem Patch entfernen, davon abhängige Editierschritte werden ebenfalls entfernt, um den Patch konsistent zu halten. Die beiden in Bild 4 dargestellten Revisionen sollen einer Variante zur Ablaufplanung am Flughafen entstammen. Der Editierschritt zur Extraktion der Klasse *Schedule*, welcher lediglich variantenspezifische Informationen (das Abflug-Gate und die Boarding-Zeit) modifiziert, wurde daher aus dem Patch entfernt.

5.4 Auflösen von Referenzen auf Modellelemente

Zur Identifikation von Operationsargumenten im Zielmodell müssen die Referenzen auf Elemente des Ursprungsmodells aufgelöst werden. Hierzu kann der Matcher des Vergleichswerkzeugs (s. Bild 2) benutzt werden [5]: Die Basisversion v0 (s. Bild 1) wird mit der Zielversion vt verglichen, allerdings wird nur das Matching berechnet, die Differenzableitung und Operationserkennung entfallen. Das Matching ordnet letztlich den Elementen der Basisversion die entsprechenden Elemente im Zielmodell zu. Je nach Modelltyp und technischen Rahmenbedingungen können hier unterschiedliche Matching-Verfahren [KRPP09] eingesetzt werden.

Bei allen Verfahren kann der Fall eintreten, daß einzelne Referenzen nicht aufgelöst werden können. Dies muss i.d.R. manuell bereinigt werden, indem die Referenz vom Entwickler aufgelöst wird oder der involvierte Editierschritt und alle von ihm direkt oder indirekt abhängigen Editierschritte im Patch gelöscht werden (s. Abschnitt 5.5).

Im Falle ähnlichkeitsbasierter Matcher besteht ferner die Gefahr der Falschauflösung von Referenzen. Eine Maßnahme zur Erkennung derartiger Fälle ist die Bewertung der Zuverlässigkeit von Korrespondenzen. Ein quantitatives Maß wird in [We11] vorgeschlagen, eine qualitative Bewertung auf Grundlage alternativer Matching-Kandidaten wird in [GK11] beschrieben. Beide Verfahren werden von dem von uns eingesetzten Vergleichswerkzeug SiDiff [KKPS12] unterstützt und integriert. So wird bspw. in dem in Abschnitt 5.5 dargestellten Anwendungsszenario die Zuverlässigkeit der Korrespondenz zwischen der Klasse *Passenger* des Ursprungsmodells und der Klasse *Passenger* des Zielmodells mit ca. 44% bewertet (s. Bild 5).

[5] Wir unterstellen hier, daß die Basisversion des Patches auf dem gleichen System wie die Zielversion verfügbar ist. Dies ist bei den Szenarien 2 und 3 typischerweise der Fall.

5.5 Kontrollierte Anwendung von Editierschritten

Nach der Auflösung der Referenzen im Zielmodell folgt die interaktive Überprüfung und kontrollierte Anwendung eines Patches. Abbildung 5 zeigt den initialen Zustand der interaktiven Anwendung des Patches unseres Beispielszenarios aus Abschnitt 5.3 auf eine vom Basismodell abweichende Variante des exemplarischen Flugbuchungssystems. Der Aufbau der Liste der anzuwendenden Editieroperationen (links oben) entspricht i.W. der Operationsliste des Editierwerkzeugs. Einzelne Editierschritte können bei Bedarf auch hier deselektiert werden, eine Konsistenzprüfung garantiert die automatische Deselektion abhängiger Editierschritte.

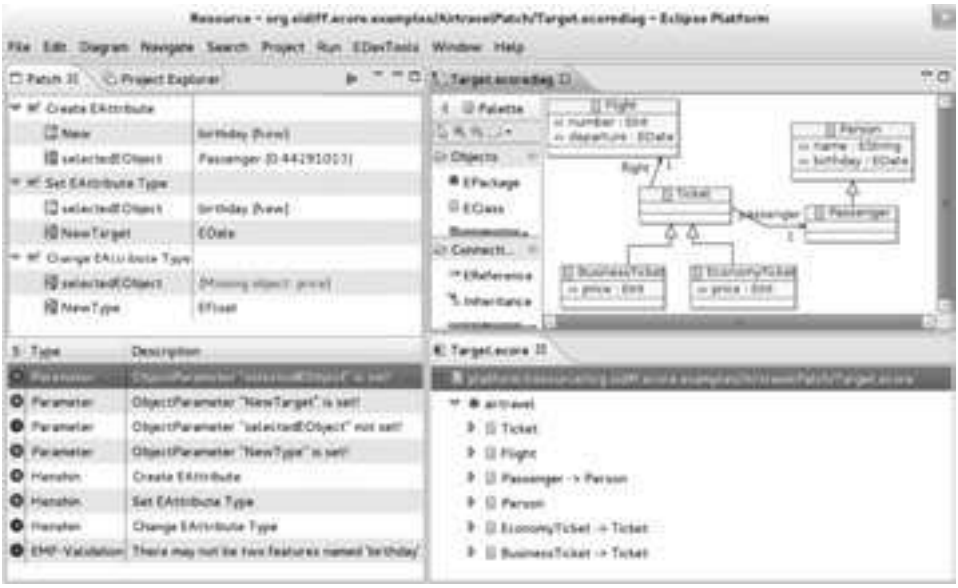

Abbildung 5: Interaktive Anwendung von Patches

Im Gegensatz zum Editierwerkzeug können Operationsargumente während der Anwendung des Patches explizit geändert (im Falle falsch aufgelöster Referenzen) oder gesetzt (im Falle nicht aufgelöster Referenzen) werden. Fehlende Operationsargumente werden hervorgehoben, so z.B. der Eingabeparameter für das Ändern des Typs des Attributs *price* unseres Ursprungsmodells[6]. Fehlende Argumente sind vom Benutzer zu selektieren und können ggf. im Zielmodell auch zunächst erstellt werden. Das Zielmodell unseres Anwendungsszenarios kann bspw. dahingehend refakturiert werden, daß die Attribute *BusinessTicket.price* und *EconomyTicket.price* durch ein Attribut *Ticket.price* der gemeinsamen Oberklasse ersetzt werden, welches anschließend als Eingabeparameter ausgewählt wird.

Die zustandsabhängige Ausführbarkeit einer Editieroperation wird überprüft, indem die Editieroperation versuchsweise auf einer internen Kopie des Zielmodells ausgeführt wird,

[6] Aufgrund von Uneindeutigkeit wurde im Beispielszenario keine Korrespondenz für dieses Attribut erzeugt

anschließend geprüft wird, ob Korrektheitsbedingungen verletzt wurden, und im Fehlerfall alle Änderungen zurückgesetzt werden. Zur Korrektheitsprüfung wird das EMF Validation Framework aufgerufen, eine entsprechende Implementierung der Überprüfung von Invarianten und Constraints wird vorausgesetzt. Validierungsfehler werden im Fehlerprotokoll (links unten) gemeldet. Im Beispielszenario schlägt die Erzeugung des Attributs *birthday* in der Klasse Passenger fehl, da ein gleichnamiges Attribut bereits durch die Oberklasse Person vererbt wird. Der entsprechende Editierschritt sowie der davon abhängige zum Setzen des Attributtyps kann vom Benutzer deselektiert werden, um die Fehlersituation aufzulösen.

Nach der Auflösung aller Fehlersituationen kann der modifizierte Patch auf das Zielmodell angewendet werden. Technisch ist hierfür jede aufzurufende Editieroperation, welche in der Repräsentation des Patches nur durch ihre Signatur im Sinne abstrakter Datentypen gegeben ist (vgl. Abbildung 3), durch eine Implementierung zu ersetzen. Konkrete Transformationstechnologien und deren ggf. vorhandene Einschränkungen werden somit gekapselt. In unserer prototypischen Implementierung verwenden wir hier die in EMF Henshin spezifizierten Editierregeln, welche bereits zur Operationserkennung (s. Abschnitt 5.1) benötigt werden. Die Ausführung einer Editierregel wird an den Henshin Regel-Interpreter delegiert, welcher die aktuellen Aufrufparameter einer Operation entgegen nimmt.

6 Vergleich mit anderen Arbeiten

Das Patchen von Modellen ist bisher erst in wenigen Projekten behandelt worden. Praktisch nutzbare Patchwerkzeuge sind zur Zeit kaum auffindbar.

Der in [CRP10] vorgestellte Ansatz fasst die Anwendung eines Patches als Graphtransformation auf und nutzt hierzu ein Graphtransformationssystem (ATL). Die bei der Anwendung eines Patches zu benutzenden Transformationsregeln werden aus einer Differenz zwischen zwei Modellen abgeleitet. Es wird ein sehr einfaches Modell von Differenzen unterstellt, das keine korrektheitserhaltenden Editieroperationen unterstützt. Das Problem, wie Differenzen als Basis von Patches gewonnen werden und wie Referenzen korrekt aufgelöst werden, wird nicht behandelt, hierzu werden nur Möglichkeiten aufgelistet. Es bleibt völlig unklar, wie der Fehlerfall, daß ein Patch nicht vollständig ausführbar ist, sicher erkannt und wie einem Entwickler eine brauchbare Darstellung der Fehlerursachen geliefert werden kann; dies würde einen tiefen Eingriff in den Regelinterpreter oder entsprechende Debugging-Interfaces erfordern. Das Ergebnismodell kann nach der Patchanwendung beliebig inkonsistent sein. Insgesamt ist der Ansatz für die in Abschnitt 3 beschriebenen Einsatzszenarien 2 und 3 unbrauchbar.

Ein deutlich praxisorientierteres Konzept für das Patchen von Modellen stellt Könemann [Ko10] vor. Vorgeschlagen wird ein detaillierter Prozeß, wie Roh-Differenzen zu einem Patch umgestaltet werden - hierzu werden mehrere naheliegende Heuristiken kombiniert - und wie der Patch später flexibel auf ein Zielmodell angewandt wird. Dieser Prozeß setzt an vielen Stellen auf interaktive Eingriffe von Entwicklern, u.a. um die Patch-Inhalte semantisch anzureichern und die Auflösung von Referenzen zu kontrollieren bzw.

zu korrigieren. Dieser Prozeß ist sehr flexibel, andererseits mit sehr hohem Arbeitsaufwand verbunden, da jede einzelne Gruppe von elementaren Änderung separat manuell behandelt werden muß. Das in unserem Ansatz unterstellte automatisierte Liften von Roh-Differenzen deckt nur einen Teil der Fälle ab, diese aber präziser und weitaus weniger arbeitsaufwendig. Analoges gilt für die Anwendung einer Differenz: Diese kann beim Könemannschen Ansatz auf eine individuelle Reimplementierung der Intention von Änderungen hinauslaufen. Im Vergleich dazu ist der Prozeß der Anpassung an das Zielmodell bei unserem Ansatz stärker automatisiert und bietet mehr Sicherheit gegen Korrektheitsverletzungen, u.a. durch Ausnutzung zusätzlicher Informationen aus der Matching-Engine, um Operationsargumente für die anzuwendenden Editieroperationen zuverlässig zu bestimmen. Beide Ansätze können im Prinzip kombiniert werden, wobei unser Ansatz die mechanisch erkennbaren komplexen Editieroperationen sicherer und effizienter bearbeiten kann, während darüber hinausgehende komplexere Editiervorgänge mit dem Könemannschen Verfahren behandelt werden könnten.

EMF Compare [EC12] unterstützt lediglich den Anwendungsfall des 3-Wege-Mischens. In einigen Fällen läßt sich das Patchen zwar auf das 3-Wege Mischen zurückführen, grundlegende Einschränkungen, so z.B. die Notwendigkeit der Existenz einer gemeinsamen Basisversion, wurden bereits in Abschnitt 2 diskutiert. Ferner besteht hierbei keine Möglichkeit, erzeugte Patches zu editieren und so auf eine konsistenzerhaltende Teilmenge von Editierschritten zu reduzieren.

Der im Rahmen des EMF Compare-Projekts entwickelte EPatch-Ansatz [EP12] benutzt Xtext zur externen Darstellung von Patches. Die nicht-interaktive Anwendung eines Patches ist hier zwar im Gegensatz zum 3-Wege-Mischwerkzeug von EMF Compare nicht mehr an die Existenz einer gemeinsamen Basisversion gebunden, Objekte im Zielmodell werden jedoch über statische, i.d.R. auf persistenten XMI-Identifizierern basierenden Pfadnamen lokalisiert. Patches sind somit lediglich auf exakte Kopien des Ursprungsmodells anwendbar.

7 Zusammenfassung

Dieses Papier präsentiert einen neuen Ansatz zum Patchen von Modellen: ein Hauptmerkmal dieses Ansatzes sind Maßnahmen, die sicherstellen, daß die gepatchten Modelle korrekt und durch andere Werkzeuge weiterverarbeitbar sind.

Technisch unterscheidet sich unser Ansatz von früheren dadurch, daß systematisch “geliftete” Differenzen benutzt werden, die nicht mehr aus elementaren ASG-Modifikationen, sondern aus korrektheitserhaltenden Editieroperationen bestehen. Besonderer Wert wurde auf die Benutzerunterstützung bei der Anwendung und ggf. interaktiven Anpassung eines Patches auf ein Zielmodell gelegt: es werden detaillierte Beschreibungen von Fehlerursachen geliefert, ferner werden Patches als editierbare Dokumente unterstützt.

Literatur

[Ar10] Arendt, T.; Biermann, E.; Jurack, S.; Krause, C.; Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations; in: Proc. Intl. Conf. Model Driven Engineering Languages and Systems 2010, Oslo; LNCS 6394, Springer; 2010

[CRP07] Cicchetti, A.; Ruscio, D.; Pierantonio, A.: A Metamodel independent approach to difference representation; Journal of Object Technology 6(9); 2007

[CRP10] Cicchetti, A.; Ruscio, D.; Pierantonio, A.: Model Patches in Model-Driven Engineering; p.190-204 in: Models in Software Engineering - Workshops and Symposia at MODELS 2009; LNCS 6002, Springer; 2010

[CFP11] Collins-Sussman, B.; Fitzpatrick, B.W.; Pilato, C.M.: Version Control with Subversion For Subversion 1.7; http://svnbook.red-bean.com/en/1.7/svn-book.pdf; 2011

[EM12] Emanuelsson, P.: There is a strong need for diff/merge tools on models; Position paper, CVSM 2012, http://pi.informatik.uni-siegen.de/CVSM2012; 2012

[EP12] EPatch; http://wiki.eclipse.org/EMF_Compare/Epatch; 2012

[EC12] EMF Compare; http://www.eclipse.org/emf/compare/; 2012

[GK11] Gorek, G.; Kelter, U.: Abgleich von Teilmodellen in den frühen Entwicklungsphasen; p.123-134 in: Software Engineering 2011; LNI 183, GI; 2011

[HK10] Herrmannsdörfer, M.; Kögel, M.: Towards a Generic Operation Recorder for Model Evolution; p.76-81 in: Proc. 1st Intl. Workshop on Model Comparison in Practice, Malaga; ACM; 2010

[KKOS12] Kehrer, T.; Kelter, U.; Ohrndorf, M.; Sollbach, T.: Understanding Model Evolution through Semantically Lifting Model Differences with SiLift; p.638-641 in: Proc. 28th Intl. Conf. on Software Maintenance (ICSM 2012); IEEE CS; 2012

[KKT11] Kehrer, T.; Kelter, U.; Taentzer, G.: A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning; p.163-172 in: Proc. 26th Intl. Conf. on Automated Software Engineering (ASE 2011); ACM; 2011

[KKPS12] Kehrer, T.; Kelter, U.; Pietsch, P., Schmidt, M.: Adaptability of Model Comparison Tools; in: Proc. 27th Intl. Conf. on Automated Software Engineering; 2012

[KSW08] Kelter, U.; Schmidt, M.; Wenzel, S.: Architekturen von Differenzwerkzeugen für Modelle; p.155-168 in: Software Engineering 2008; LNI 121, GI; 2008

[Ko10] Könemann, P.: Capturing the Intention of Model Changes; p.108-122 in: Proc. Intl. Conf. Model Driven Engineering Languages and Systems 2010, Oslo, Part I; LNCS 6394, Springer; 2010

[KRPP09] Kolovos, D.S.; Ruscio, D.D.; Pierantonio, A.; Paige, R.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

[Me02] Mens, T.: A state-of-the-art survey on software merging; IEEE Trans. Softw. Eng.; 28(5), p.449-462; 2002

[RV08] Rivera, J.E.; Vallecillo, A.: Representing and Operating with Model Differences; p.141-160 in: Proc. TOOLS EUROPE 2008; LNBIP 11, Springer; 2008

[SZN04] Schneider, Ch.; Zündorf, A.; Niere, J.: CoObRA - a small step for development tools to collaborative environments; ICSE Workshop on Directions in Software Engineering Environments; 2004

[We11] Wenzel, S.: Unique Identification of Elements in Evolving Models: Towards Fine-Grained Traceability in Model-Driven Engineering; Dissertation, Universität Siegen; URN: urn:nbn:de:hbz:467-5243; 2010