Automatisierte Erzeugung von Transformationsregeln zur Optimierung von Modelldifferenzen

Bachelorarbeit

Manuel Ohrndorf

Siegen, 23. Februar 2012

Universität Siegen
Fakultät IV
Department Elektrotechnik und Informatik
Institut für Praktische Informatik

Erstgutachter:	Prof. Dr. Udo Kelter
Zweitgutachter:	Dipl.-Inf. Timo Kehrer

Kurzfassung: Im Zentrum des *Model-Driven Software Development* (MDSD) stehen Modelle, welche häufig durch Teams entwickelt werden. Um in einem Team mit Modellen zu arbeiten, werden Versionsverwaltungswerkzeuge benötigt. Eine Hauptaufgabe dieser Werkzeuge ist das Vergleichen von verschiedenen Versionen oder Varianten eines Modells. Um zwei Modelle miteinander zu vergleichen wird eine Differenz zwischen den beiden Modellen durch ein spezielles Differenzwerkzeug berechnet. Da es eine Vielzahl von verschiedenen Modelltypen gibt, müssen die Differenzwerkzeuge generisch arbeiten um mit allen Modelltypen umgehen zu können. Dadurch entsteht allerdings das Problem, dass die Differenz zwischen zwei Modell Versionen nur auf Basis der internen Repräsentation der Modelle angegeben werden kann. Die Differenzen auf Basis der internen Darstellung werden hier als low-level Änderungen bezeichnet. Solche low-level Änderungen sind aber für Entwickler die nur die externe Repräsentation der Modelle kennen schwer zu lesen. Diese Arbeit setzt sich damit auseinander die Lesbarkeit der Differenzen zu verbessern, indem den low-level Änderungen wieder bestimmte Editieroperationen zugeordnet werden. Dieser Prozess wird hier als Semantic-Lifting bezeichnet.

Abstract: Models are in the center of *Model-Driven Software Development* (MDSD) and they are often developed in teams. To comply with the task of sharing a model among team members we need special version management tools. One main task of a version management tool is the comparison of different model versions. We also need special model difference tools to calculate the changes between the two models. The algorithms of the difference tools have to deal with many different model types. Instead of developing an individual algorithm for each model type, the difference tools have to be generic. One problem of a generic difference algorithm is that it has to compare the models based on their internal representation. Thus, difference tools initially derive low-level changes from the internal representation of a model. But the low-level changes are often incomprehensible for normal tool users, who usually prefer model changes to be explained in terms of edit operations that are available from a user's point of view. So the primary goal here is to increase the readability of the difference by lifting the low-level changes to the level of user edit operations. We will call this process Semantic-Lifting.

Inhaltsverzeichnis

1 Einleitung

Sowohl unser Geschäfts- als auch unser Alltagsleben sind heutzutage durch Softwaresysteme geprägt. Dabei werden sowohl aus Benutzersicht als auch aus technischer Sicht hohe Anforderungen an die verwendete Software gestellt. Eine Software soll gut bedienbar, zuverlässig, sicher, vernetzbar, anpassbar und dabei auch möglichst günstig sein. Um diesen Anforderungen gerecht zu werden, werden die Software Projekte auf technischer Seite zum einen immer komplexer und größer, zum anderen muss ein solches Projekt häufig über Jahre oder Jahrzehnte hinweg gepflegt und gewartet werden. Im Laufe dieser Zeit kann es passieren, dass sich gewisse Anforderungen an das Projekt ändern oder dass die Software auf eine neu entwickelte Technologie übertragen werden soll. Nicht zuletzt spielen auch die Kosten, die in eine solche Entwicklung einfließen eine entscheidende Rolle. Daher werden immer neu Methoden benötigt, um die effektive Entwicklung und Wartung eines Softwaresystems zu ermöglichen.

Ein Konzept, welches in der Praxis immer mehr an Bedeutung gewinnt, ist das *Model-Driven Software Development* (MDSD) [MM03], [PTN+07], [GPR06]. Die Philosophie hinter diesem Softwareentwicklungskonzept ist es, möglichst viele Teile der Softwareherstellung zu automatisieren. Als Grundlage hierfür werden hinreichend formale und zugleich abstrakte Modelle verwendet, aus denen dann verschiedene Artefakte des Systems abgeleitet werden. Modelle werden dabei vorzugsweise als Diagramme angelegt, da komplexe Strukturen visuell so besser überblickt werden können. Die abgeleiteten Artefakte können zum einen ausführbare Programmteile in einer bestimmten Zielsprache wie z.B. Java oder C# sein, zum anderen können aber auch Konfigurationsdateien, Datenbankskripte, Testfälle oder Dokumentationsdateien aus den Modellen automatisch abgeleitet werden, während dieser Prozess in der klassischen Software Entwicklung eine manuelle Tätigkeit ist. Daher nehmen Modelle in MDSD ein zentrale Stellung im gesamten Entwicklungsprozess einer Software ein. [PTN+07] (S.11 - 13)

> „Das bedeutet auch, dass Modelle und einzelne Modellelemente z.B. bezüglich Versionierung genauso zu behandeln sind wie Quelltext. Verteilte Teams müssen also in der Lage sein, Modelle aus einem Versionsverwaltungssystem

auszuchecken, einzelne Modellelemente zu bearbeiten, mit früheren Versionen zu vergleichen, konkurrierende Modifikationen aufzulösen und den neuen Stand wieder in das Versionsverwaltungssystem einzuchecken.“ [PTN+07] (S.12)

Um verschiedene Versionen von Modellen zu vergleichen und um die Unterschiede zwischen den Versionen anzuzeigen, werden s.g. Differenzwerkzeuge benötigt. Differenzwerkzeuge, mit denen normalerweise Quelltexte verglichen werden, sind für strukturierte Modelle nicht zu gebrauchen [BE09], da diese auf Basis ihrer Graphrepräsentation verglichen werden müssen. Viele verfügbare Differenzwerkzeuge arbeiten generisch, so dass sie für verschiedene Modelltypen eingesetzt werden können. Ein Nachteil, der daraus resultiert, ist, dass die Modelle anhand ihrer internen Repräsentation verglichen werden müssen. Diese interne Repräsentation unterscheidet sich aber oft signifikant von der externen Darstellung der Modelle. Ein Entwickler, der mit einem bestimmten Modelltyp arbeitet, kennt vielleicht die interne Repräsentation gar nicht. Es wäre also am besten, wenn die Differenz zwischen zwei Modellen als Abfolge von Editierschritten auf Basis der externen Darstellung angezeigt würde. [KKT11] (S.1)

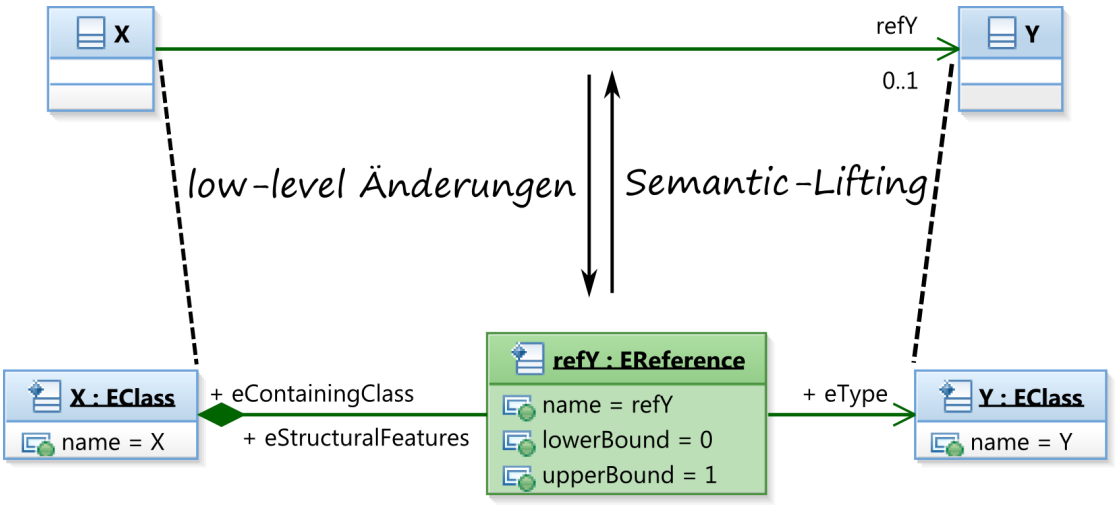


Abbildung 1.1: Semantic-Lifting

Abbildung 1.1 demonstriert dies anhand des Einfügens einer Referenz zwischen zwei Klassen eines Ecore Modells (Abbildung 2.2). Wie man sieht wird dazu in der internen Darstellung eine neue Klasse angelegt, die diese Referenz repräsentiert. Sowohl der Name als auch die Multiplizitäten der Referenzen werden als Attribute in dieser Klasse gespeichert. Zusätzlich werden noch mehrere Referenzen benötigt, welche Quelle und

Ziel der eingefügten Referenz festlegen. Eben diese s.g. **low-level Änderungen** werden durch das Differenzwerkzeug erkannt und angezeigt. Die Frage ist nun, ob es möglich ist nur anhand der low-level Änderungen zu erkennen, welche Editieroperation auf das Modell angewendet wurde, um dem Benutzer so eine Differenz auf Basis der externen Repräsentation anzuzeigen.

Das Konzept „A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning“ [KKT11] beschäftigt sich mit genau dieser Problematik. Hier wird ein Algorithmus beschrieben, der einer beliebigen Anzahl von low-level Änderungen wieder die entsprechenden Editieroperationen zuordnen kann. Dazu werden Editierregeln verwendet, welche eine Editieroperation beschreiben. Die Editierregeln werden dann dazu benutzt, um das der Editieroperation entsprechende Muster von low-level Änderungen wieder zu erkennen. Dieser Prozess wird in Anlehnung an das Konzept im Folgenden als **Semantic-Lifting** bezeichnet.

Diese Arbeit beschreibt die praktische Untersuchung, Implementierung und die daraus resultierenden Ergebnisse des Semantic-Liftings. Dazu werden in Kapitel 2 zunächst die wichtigsten Technologien für dieses Projekt besprochen. Kapitel 3 gibt einen Überblick über den im Folgenden beschriebenen Algorithmus. Hierzu wird in den Kapiteln 4, 5 und 6 die Erstellung von Editierregeln und die Generierung von daraus resultierenden Mustern zur Erkennung der Editeroperationen erläutert. In den Kapiteln 7, 8, 9, 10 und 11, wird dann dargelegt, wie mit Hilfe dieser Erkennungsmuster, das eigentlich Semantic-Lifting durchgeführt wird. In Kapitel 12 und 13 werden zum Schluss die Ergebnisse des Semantic-Liftings ausgewertet.

2 Grundlagen

Im Folgenden werden die beiden wichtigsten Technologien vorgestellt, die als Grundlage für diese Arbeit verwendet wurden. Zum einen das Eclipse Modeling Framework, welches die Umgebung für die gesamte Implementierung darstellt und zum anderen die Modell Transformationssprache Henshin als wichtigste Technolgie zur Umsetzung des Semantic-Liftings.

2.1 Eclipse Modeling Framework

Das Eclipse Modeling Framework (EMF) ist, wie der Name schon sagt, ein auf Eclipse basierendes Framework zur Generierung von Quelltext aus Modellen. EMF bildet dabei eine Brücke zwischen verschiedenen Modellierungsansätzen. Als Eingabe für EMF können z.B. UML-Diagramme, XML-Schemata oder spezielle Java Interfaces dienen. Damit EMF diese Daten weiter verarbeiten kann, wird dann aus jedem Eingabemodell ein entsprechendes Modell im EMF eigenen Modelltyp Ecore erzeugt. EMF bietet aber auch Editoren, in denen Ecore Modelle direkt erzeugt und bearbeitet werden können. Ecore ist ein s.g. plattformunabhängiges Metamodell (*engl. platform independent model (PIM)*), d.h. durch Ecore wird noch nicht festgelegt, für welche Zielplattform später Quelltext generiert werden soll und wie die konkrete Implementierung des Modells aussehen wird. Eine etwas vereinfachte Form des Ecore Metamodell ist in Abbildung 2.2 zu sehen. Ecore ist eine Implementierung des EMOF (*Essential Meta Object Facility* [MOF11]) Metamodells. EMOF beschreibt einen Ausschnitt des Metamodells der UML 2.0. Die in Ecore formulierten Modelle dienen nur dazu, die Datenstrukturen eines Programms zu beschreiben. In der gesamten UML gibt es hingegen aber auch Modelle, um das Verhalten einer Software zu beschreiben. So wie EMOF ist Ecore ein Modell, welches sich selbst definiert, d.h. das Metamodell von Ecore ist selbst ein Ecore Modell. Man spricht dann auch von einem selbstreferentiellen Metamodell.

Wie in Abbildung 2.1 dargestellt, wird neben dem Ecore Modell zur Quelltext-Generierung noch ein s.g. Generator Modell (*GenModel*) benötigt. Das Generator Modell wird weitestgehend automatisch erzeugt und erweitert das plattformunabhängige Ecore

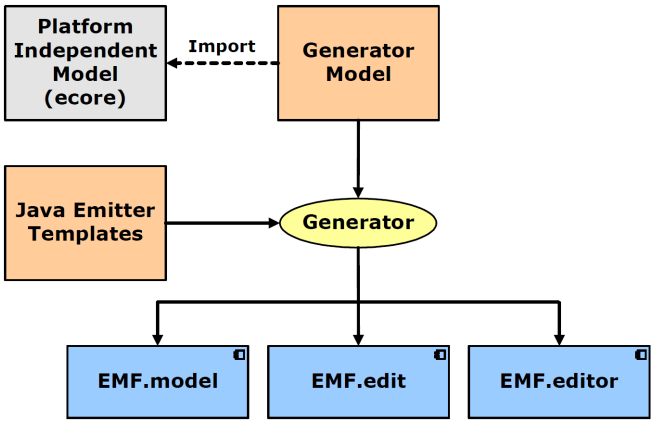


Abbildung 2.1: „EMF Toolset from 30.000 Feet“ [BG05] (S.7)

Modell um plattformspezifische Informationen. Das Generator Modell kann vom Benutzer angepasst werden und steuert direkt die Quelltext Ausgabe des darunter liegenden Generators. Innerhalb des Generators findet nun eine s.g. Modell-zu-Text Transformation statt, d.h. aus dem Generator Modell wird nun der eigentliche Quelltext generiert. Dazu werden Java Emitter Templates (JET) verwendet, diese enthalten Textmuster die vorgeben wie die Modell-Elemente zu transformieren sind. Die Zielsprache von EMF ist grundsätzlich Java. Theoretisch kann JET aber auch verwendet werden, um einen Generator zu schreiben, der aus dem plattformunabhängigen Ecore Modell z.B. SQL oder C# Quelltext erzeugt. In Abbildung 2.1 sind die drei wichtigsten Pakete zu sehen, die generiert werden können:

- **EMF.model:** Der hier erzeugte Java Quelltext ist ein s.g. plattformspezifisches Modell (*engl. platform specific model (PSM)*). Dieses stellt die Implementierung des zuvor in Ecore beschriebenen plattformunabhängigen Modells dar. Der Java Quelltext ist in der Regel sofort ausführbar, um als Datenmodell für eine drauf aufbauende Applikation eingesetzt zu werden. An gegebenen Stellen kann bzw. muss der Entwickler dann noch eigenen Code einfügen.

- **EMF.edit:** Dieses Paket stellt eine Zugriffsschicht zwischen Modell und Modell-Editor dar. Zum einen wird durch die Zugriffsschicht festgelegt, welche Befehle der Editor auf dem Modell ausführen darf, zum anderen werden die Daten des Modells für den Editor ggf. in passender Form aufbereitet. Je nach benötigter Funktionalität kann die Zugriffsschicht dann noch an die Gegebenheiten des Editors angepasst werden.

- **EMF.editor:** Hier stellt EMF einen einfachen baumbasierten Editor zum Erstellen und Bearbeiten von Modell Instanzen zur Verfügung. Der Editor wird häufig noch manuell angepasst oder auch vollständig ersetzt.

EMF bietet noch viele weitere Unterstützungen um Modelle zur Laufzeit zu verändern, diese zu Validieren oder die Daten auf Basis von XML zu Serialisieren. Eine ausführliche Einführung in die umfangreichen Funktionalitäten von EMF ist in [SBPM09], [BG05] und unter [EMF12] zu finden.

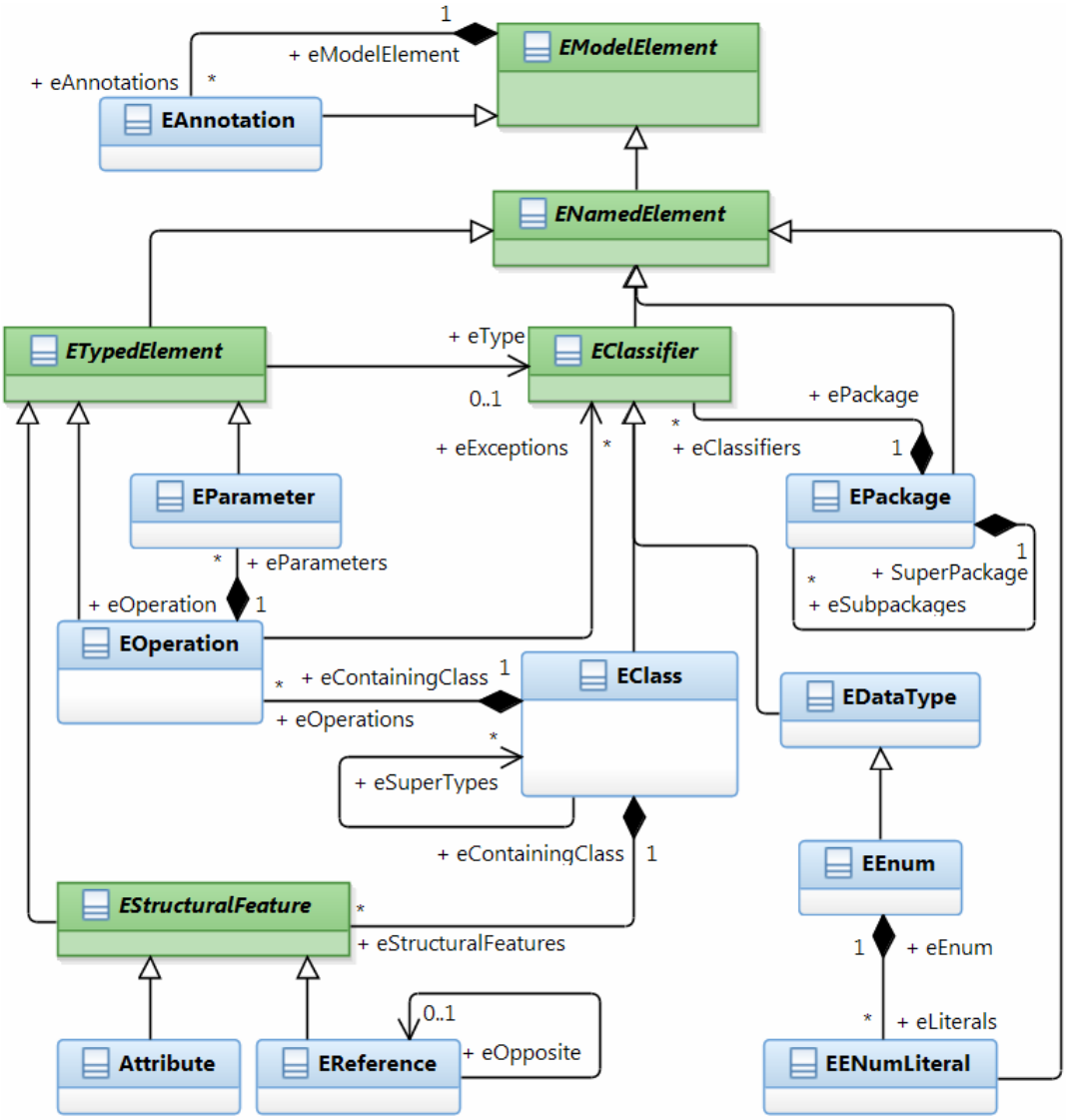


Abbildung 2.2: Ecore Metamodell (vereinfacht)

2.2 Henshin

Der Name „Henshin“ bedeutet Transformation auf Japanisch. Henshin ist eine auf EMF basierende *in-place* Transformationssprache für Ecore Modelle. Als *in-place* werden Transformationssprachen bezeichnet, die Änderungen direkt (zur Laufzeit) am Modell durchführen. Im Gegensatz dazu würde eine *out-place* Transformationssprache das Ursprungsmodell nicht verändern und eine neue Instanz des Modells mit den durchgeführten Änderungen erzeugen. Grundsätzlich kann Henshin aber ebenfalls als *out-place* Transformationssystem verwendet werden, indem man ein Modell in Henshin lädt, die Transformation durchführt und das Modell wieder an einem anderen Ort abspeichert.

Weitere Informationen über Henshin sind in [ABJ+10], [BESW10] und unter [HEN12] zu finden. Die folgenden Abschnitte geben einen kurzen Überblick über die Henshin Funktionalitäten.

2.2.1 Regeln

Henshin ist ein Graphersetzungssystem, dieses Konzept beruht auf der Graphentheorie. Ein Graph ist in Henshin typisiert und besteht aus Knoten, gerichteten Kanten und Attributen. Eine Transformation wird durch eine Regel angegeben. Diese Regel besteht aus einer linken (*Left-Hand Site* (LHS)) und einer rechten Seite (*Right-Hand Site* (RHS)). Beide Seiten der Regel sind Graphen. Der Grundgedanke bei Graph-Transformationsregeln ist ähnlich wie bei Produktionsregeln von Chomsky-Grammatiken für Texte, es wird innerhalb des s.g. Arbeitsgraphen ein Bild gesucht, welches der linken Seite entspricht. Man spricht auch von einer Abbildung (*engl. Match*) oder im Sinne der Graphentheorie von einem Graphmorphismus der linken Seite auf den Arbeitsgraphen. Wurde eine solche Abbildung gefunden, dann wird diese durch die rechte Seite ersetzt. Konnte die Transformation korrekt durchgeführt werden, dann wird die Regel als anwendbar bezeichnet. Zusätzlich muss noch der Schnitt von LHS und RHS angegeben werden. Dazu müssen in Henshin Mappings zwischen allen LHS und RHS Knoten angelegt werden, die den gleichen Knoten repräsentieren. Der Schnitt zwischen LHS und RHS ist genau der Teil der Abbildung, der beim Ausführen der Regel bewahrt (also nicht verändert) wird. Alle Teile der linken Seite, die nicht im Schnitt enthalten sind, werden also aus dem Arbeitsgraphen entfernt. Alle Teile der rechten Seite, die nicht im Schnitt enthalten sind, werden im Gegensatz dazu neu in den Arbeitsgraphen eingefügt.

Um die Abbildungen einer Regel einzuschränken, können zusätzlich noch s.g. *negativ application conditions* (NAC) formuliert werden. Eine NAC ist ebenfalls ein Graph. Allerdings darf sich dieser NAC Graph nicht auf den Arbeitsgraphen abbilden lassen,

damit die Regel angewendet werden darf. Mehrere NACs können über eine `AND` Formel logisch miteinander verbunden werden.

Für jeden Knoten können auch, entsprechend seinem Typ, Attribute angelegt werden. Jedes Attribut hat einen bestimmten Wert. Der Wert eines Attributs kann entweder ein primitiver Wert sein oder ein *JavaScript* Ausdruck. Innerhalb des Ausdrucks können Parameter verwendet werden. Diese werden einfach mit dem entsprechenden Namen für die Regel angelegt. Ein Parameter kann entweder beim Aufrufen von außen an die Regel übergeben werden oder er wird beim Abbilden der linken Seite mit einem Wert aus dem entsprechenden Objekt des Arbeitsgraphen initialisiert. Dazu muss es mindestens ein Henshin Attribut auf der linken Seite der Regel geben, welches nur den Parameter als Wert besitzt. Der Ausdruck wird dann durch die Rhino Script-Engine [RIH12] verarbeitet und ausgewertet.

Wird ein Attribut auf der rechten Seite der Regel für einen Knoten angelegt, so wird der Wert des Attributs in das Objekt des Arbeitsgraphen geschrieben, auf welches der Knoten abgebildet wurde. Wird ein Attribut hingegen auf der linken Seite angelegt, dann muss sich der Wert ebenfalls auf den Arbeitsgraphen abbilden lassen, sofern das Attribut nicht wie zuvor beschrieben zur Initialisierung eines Parameters verwendet wurde.

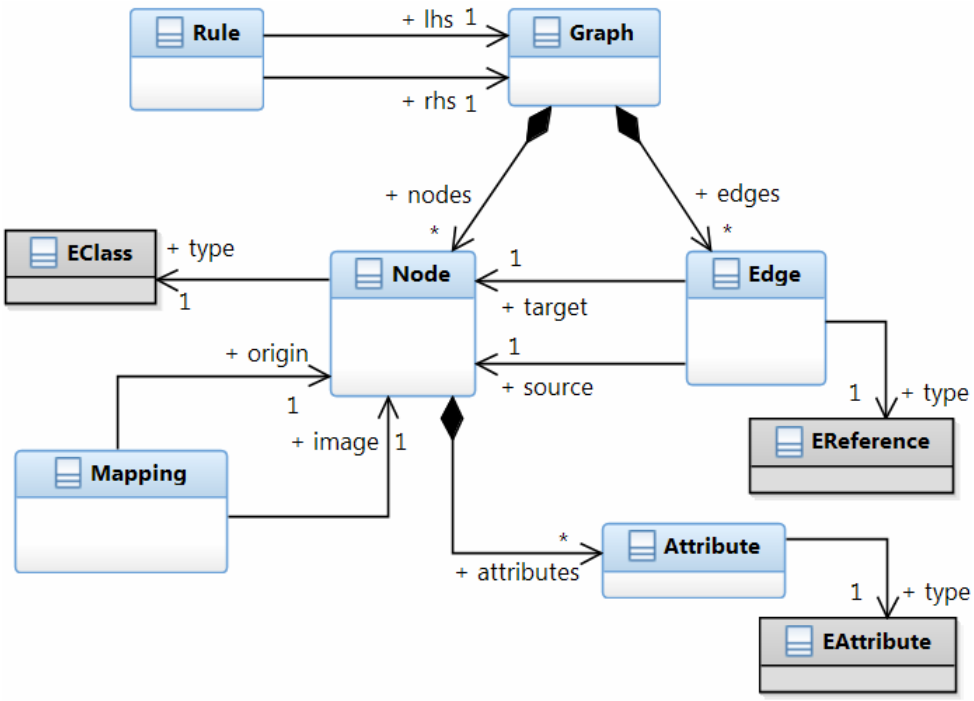


Abbildung 2.3: Henshin Metamodell für Regeln

Das Henshin Metamodell, über welches die Regeln intern dargestellt werden, ist in Abbildung 2.3 zu sehen. Das Modell basiert auf Ecore und somit können beliebige Modelle transformiert werden, die ebenfalls auf Ecore basieren.

2.2.2 Units

Mehrere Regeln werden in Henshin in Transforamtionssystemen zusammengefasst. Um den Ablauf einer Transformation mit mehreren Regeln zu steuern, werden s.g. Units verwendet. Manche Units können auch ineinander verschachtelt werden (siehe Abbildung 2.4). In diesem Zusammenhang werden Regeln dann ebenfalls als Units betrachtet. Insgesamt wird in Henshin zwischen sechs verschiedenen Units unterschieden. Eine Unit ist anwendbar, wenn sie erfolgreich ausgeführt werden konnte. Wird festgestellt, dass eine Unit nicht anwendbar ist, dann werden alle Transformationen, die bis zu diesem Zeitpunkt gemacht wurden, wieder rückgängig gemacht.

- **Independent Unit:** Führt zufällig eine der Subunits einmal aus. Eine Idependent Unit ist anwenbar, wenn mindestens eine Subunit anwendbar ist.

- **Sequential Unit:** Führt alle Subunits in gegebener Reihenfolge aus. Eine Sequential Unit ist anwenbar, wenn alle Subunits anwendbar sind.

- **Counted Unit:** Führt die Subunit mit der angegebenen Häufigkeit aus. Mit einer Häufigkeit von -1 wird die Subunit solange ausgeführt, bis sie nicht mehr anwendbar ist. Eine Counted Unit mit einer Häufigkeit von -1 ist immer anwendbar, ansonsten ist die Counted Unit anwendbar, wenn sie mit der angegebenen Häufigkeit ausgeführt werden konnte.

- **Conditional Unit:** `IF THEN ELSE` Konstrukt für Units. Eine Conditional Unit ist anwendbar, wenn (je nach Anwendbarkeit der `IF` Subunit) die `THEN` Subunit bzw. die `ELSE` Subunit anwendbar ist.

- **Priority Unit:** Führt die erst mögliche anwendbare Subunit mit der höchsten Priorität einmal aus.

- **Amalgamation Unit:** Eine Amalgamation Unit besteht aus einer Kernregel und beliebig vielen Multiregeln. Die Kernregel wird dann genau einmal ausgeführt, während die Multiregeln so oft wie möglich ausgeführt werden. Damit die Amalgamation Unit anwendbar ist, muss mindestens die Kernregel anwendbar sein. Die

Kernregel muss dabei in jede Multiregel eingebettet werden. Dies wird gekennzeichnet durch Mappings zwischen den Knoten der Kernregel und den entsprechenden Knoten der Multiregel.

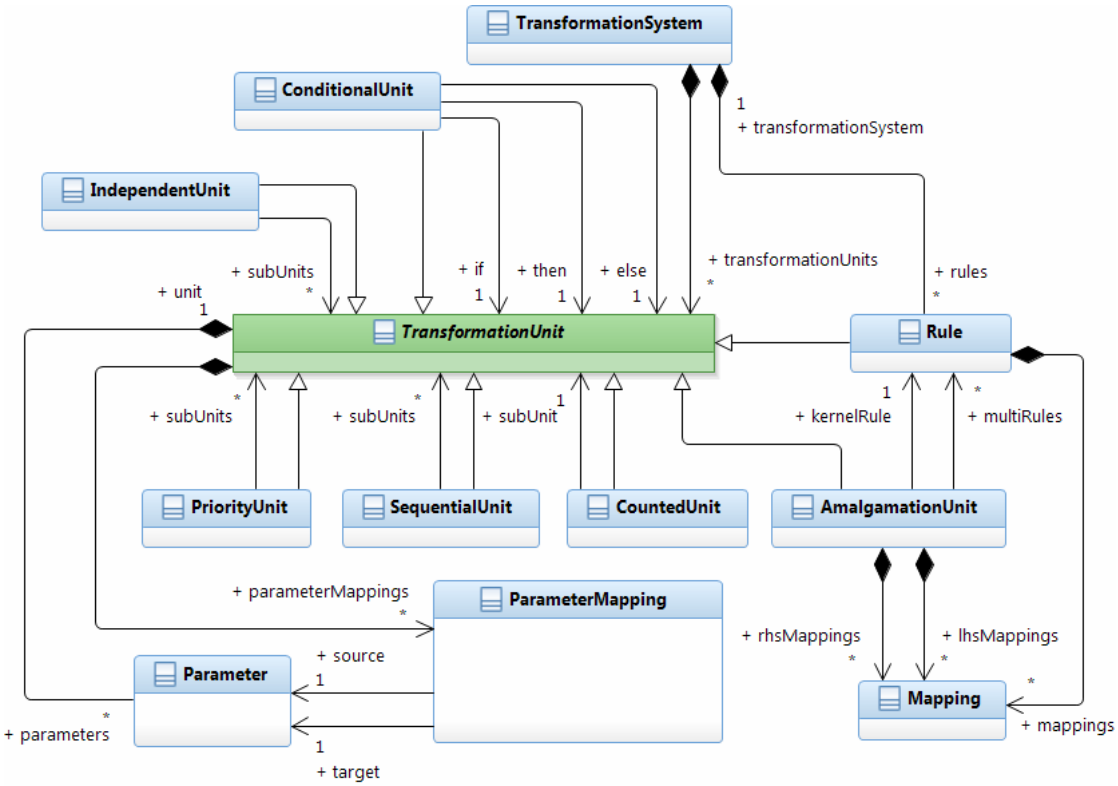


Abbildung 2.4: Henshin Metamodell für Units

2.2.3 Editoren

Henshin bietet zwei Editoren zum Entwickeln von Transformationssystemen. Der erste Editor (Abbildung 2.5) basiert auf dem Standard EMF-Editor und bietet eine baumbasierte Ansicht des Transformationssystems. Der zweite Editor (Abbildung 2.6) ist speziell dazu gemacht, um Henshin Regeln zu entwerfen. Die Regeln werden hier in einer integrierten Darstellung angezeigt. D.h. alle Elemente (Knoten, Kanten, Attribute), die nur auf der linken Seite der Regel vorkommen, werden mit dem Stereotyp `«delete»` gekennzeichnet; Alle Elemente, die nur auf der rechten Seite der Regel vorkommen, werden mit `«create»` gekennzeichnet und alle Elemente, die im Schnitt von LHS und RHS enthalten

sind, werden mit «preserve» gekennzeichnet. Der Graph der Regel lässt sich also intuitiv so lesen, dass alle «delete» Anteile gelöscht, alle «create» Anteile hinzugefügt und alle «preserve» Anteile bewahrt werden, wenn die Regel auf den Arbeitsgraphen angewendet wird. NAC Anteile werden in dieser Darstellung durch den Stereotyp «forbid» gekennzeichnet. Im Beispiel in Abbildung 2.5 und 2.6 wird ein EAttribute von einer EClass in ein über eine EReference benachbarte EClass verschoben.

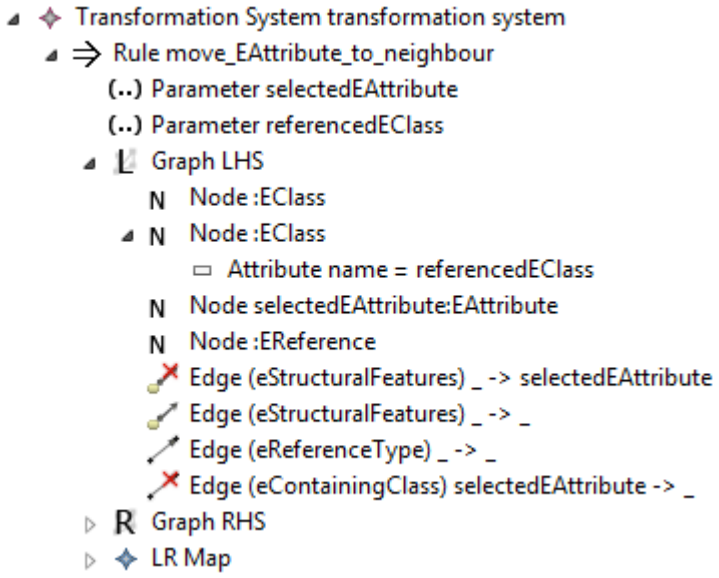


Abbildung 2.5: Baumbasierter Henshin Editor

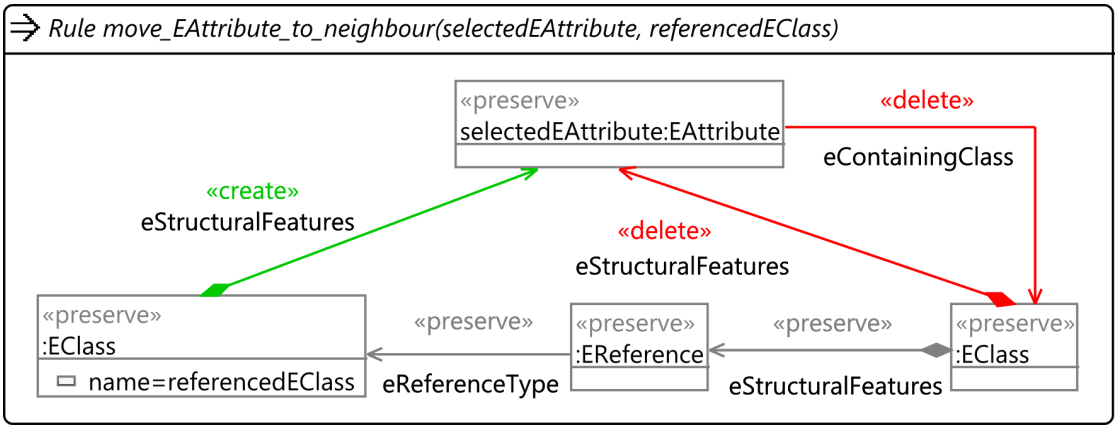


Abbildung 2.6: Visueller graphbasierter Henshin Editor

3 Differenz Pipeline

Die folgenden Abschnitte dokumentieren die Implementierung des in [KKT11] beschriebenen Konzepts des Semantic-Liftings. Die einzelnen Teile des Konzepts werden dabei in Bezug auf ihre technische Umsetzung nochmal im Detail untersucht. Es wird dabei auch auf aufgetretene Probleme und ggf. auf deren Lösungsansatz eingegangen.

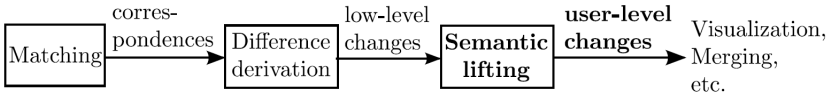


Abbildung 3.1: Differenz Pipeline [KKT11]

Die Architektur des Semantic-Lifting Algorithmus lässt sich als Pipeline beschreiben, wie in Abbildung 3.1 dargestellt. Als Eingabe für diese Pipeline dienen immer zwei Versionen eines Modells. Diese beiden Versionen werden im Folgenden immer als **Modell A** und **Modell B** bezeichnet, wobei Modell A als die Ausgangsversion des Modells betrachtet wird, während Modell B eine spätere Version des Modells darstellt, in der im Vergleich zu Modell A verschiedene Teile hinzugefügt oder entfernt wurden. Als technische Grundlage für die Implementierung dient das Eclipse Modeling Framework (EMF). Daher können nur Modelle verglichen werden, die auf dem EMF eigenen Metamodell Ecore basieren. Im Rahmen dieses Proof-Of-Concepts verwenden wir Ecore direkt als Metamodell für die zu vergleichenden Modelle. Grundsätzlich können aber auch z.B. auf Ecore basierende UML Klassendiagramme oder Zustandsautomaten als Eingabe dienen, wie Abbildung 3.2 illustriert. In allen Fällen bildet Ecore einen selbstreferentiellen Abschluss der Abstraktionsebenen als Meta-Metamodell.

Die Modelle A und B werden dann in den folgenden Schritten weiter verarbeitet. Der erste Schritt auf dem Weg zur semantisch gelifteten Differenz ist das Matching (Abschnitt 3.1). In dieser Phase wird versucht den Elementen aus Modell A die entsprechend gleichen Element aus Modell B zuzuordnen. Den nächsten Schritt bildet die Differenz-Ableitung (Abschnitt 3.2), in der die später zu verarbeitende technische Differenz aus den Matchings erzeugt wird. Im letzten Schritt erfolgt das Semantic-Lifting (Abschnitt 3.3), hier wird aus der technischen Differenz eine semantisch geliftete Differenz berechnet.

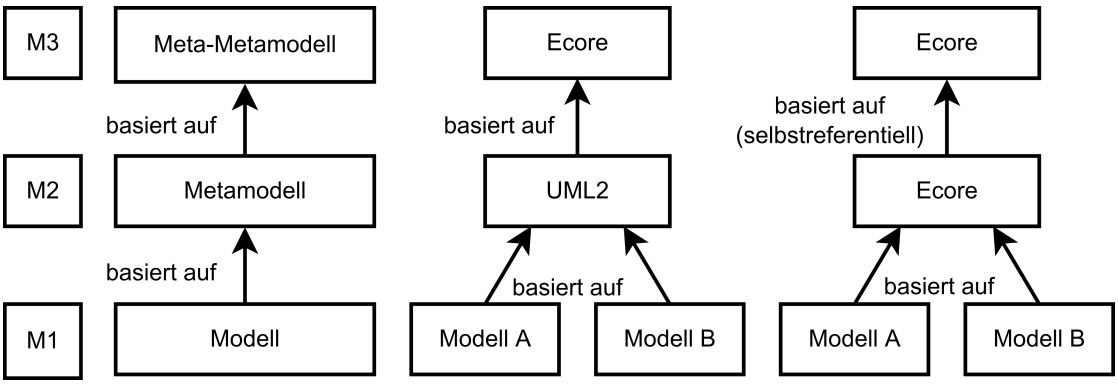


Abbildung 3.2: Metaebenen

3.1 Matching

Als Matching wird in diesem Zusammenhang der Prozess bezeichnet, in dem die beiden Modelle A und B miteinander verglichen werden. Dazu werden zustandsbasierte Vergleichsalgorithmen verwendet, welche eine symmetrische Differenz auf der Basis des aktuellen Zustands der beiden Modelle berechnen. Ziel des Matchings ist es herauszufinden welche Elemente in beiden Modellen übereinstimmen. Diese Übereinstimmungen werden als Korrespondenzen (*engl. Correspondence*) zwischen Modellen A und B bezeichnet. [KKT11] (S.3)

$$A \Delta B := \{x \mid (((x \in A) \wedge (x \notin B)) \vee ((x \in B) \wedge (x \notin A)))\}$$

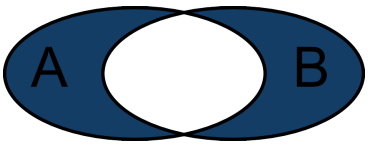


Abbildung 3.3: Symmetrische Differenz

Die Korrespondenzen geben zunächst die Schnittmenge zwischen den Modellen an. Für die Berechnung der korrespondierenden Elemente gibt es grundsätzlich verschiedene (zustandsbasierte) Techniken und Algorithmen:

- **ID-basiert:** In diesem Ansatz wird davon ausgegangen, dass jedes Modell-Element eine eindeutige statische ID besitzt. D.h. diese ID wird beim Erzeugen des Elements vergeben und darf sich auch im weiteren Entwicklungsverlauf des Modells nicht verändern. Der Hauptvorteil dieses Ansatzes besteht darin, dass keine spezielle

Konfiguration für unterschiedliche Modelltypen nötig ist. Außerdem lassen sich Korrespondenzen sehr effizient berechnen. Dieser Ansatz funktioniert allerdings nicht für Modelle, die unabhängig voneinander erzeugt wurden. Außerdem muss die Vergabe von IDs durch die verschiedenen Entwicklungsumgebungen gewährleistet sein. [KPRP09] (S.2)

- **Signaturbasiert:** In diesem Ansatz wird eine Signatur zum vergleichen der Modell-Elemente verwendet. Im Gegensatz zu einer ID ist die Signatur nicht statisch. Eine Signatur wird dynamisch aus den Werten der Eigenschaften eines Elements berechnet. Daher können auf diese Weise auch Modelle verglichen werden, die unabhängig voneinander entwickelt wurden. Dieser Ansatz ist allerdings mit deutlich mehr Aufwand verbunden, da für jeden Modell-Element-Typ eine Berechnungsfunktion für die jeweilige Signatur angelegt werden muss. [KPRP09] (S.3)

- **Ähnlichkeitsbasiert:** In diesem Ansatz werden Modelle als typisierte, attributierte Graphen behandelt. Die einzelnen Modell-Elemente werden dabei durch die Summe ihrer Eigenschaften identifiziert. Dabei sind nicht alle Eigenschaften gleich wichtig. Hierzu wird in den meisten Algorithmen eine Konfiguration benötigt die das relative Gewicht der einzelnen Eigenschaften angibt. Herauszufinden welche Gewichtung der Eigenschaften für eine Modellierungssprache ein möglichst optimales Ergebnis liefert, ist allerdings häufig ein „trial-and-error“ Prozess. Außerdem können generische Ansätze keine Rücksicht auf die Semantik der Modellierungssprache nehmen, welche die Genauigkeit verbessern und die Anzahl der individuellen Vergleiche verringern würde. [KPRP09] (S.3)

- **Sprachspezifisch:** Ein sprachspezifischer Matching Algorithmus wurde speziell für eine Modellierungssprache implementiert. Da diese Algorithmen die Semantik ihrer Modellierungsprache kennen ist hier in der Regel auch kein Semantic-Lifting notwendig. Daher wird hier auch nicht näher auf solche Algorithmen eingegangen. Der Nachteil dieses Ansatzes bestehen im hohen Entwicklungsaufwand und der auf die Modellierungssprache eingeschränkten Benutzbarkeit.

Zur Berechnung der Korrespondenzen für das Semantic-Lifting kann grundsätzlich ein beliebiger Algorithmus verwendet werden. Die Korrespondenzen müssen nur immer in die interne Darstellung konvertiert werden. (Siehe `Correspondeces` Abbildung 3.4.) Ein Vergleich verschiedener Ansätze ist in [KPRP09] zu finden. Im folgenden wird nun auf die im Rahmen dieser Arbeit verwendeten Algorithmen eingegangen.

3.1.1 SiDiff

SiDiff ist ein an der Universität-Siegen entwickelter Metamodell unabhängiger Ansatz, um Modelle zu vergleichen. Der Algorithmus ist hauptsächlich darauf ausgelegt Modell-Elemente nach ihrer Ähnlichkeit zu vergleichen (ähnlichkeitsbasiert), unterstützt aber auch Ansätze zum ID-basierten oder signaturbasierten Modellvergleich. Der Hauptvorteil von SiDiff besteht darin, dass dem Benutzer eine sehr frei konfigurierbare Umgebung geboten wird, die auf jeden Modelltyp angepasst werden kann. Der Modelltyp muss sich lediglich als graphähnliche Struktur darstellen lassen. Der generische Algorithmus muss für jeden Modelltyp konfiguriert werden. Eine Konfiguration besteht aus einer Transformation vom Originaldokument in die interne Repräsentation, einer Definition für die Gewichtung der Eigenschaften der zu vergleichenden Modell-Elemente für die Berechnung der Ähnlichkeiten und einer Spezifizierung der auszugebenden Daten. [SiD11]

3.1.2 Named-Element-Matcher

Der Named-Element-Matcher wurde im Rahmen diese Projekts als Testwerkzeug verwendet, er vergleicht die beiden Modelle A und B einfach anhand der Namen der einzelnen Elemente. Der Matcher eignet sich vor allem für selbst konstruierte Testfälle, da das Ergebnis des matchings relativ gut abzusehen ist. Dieser Algorithmus lässt sich als ein signaturbasierter Algorithmus einordnen. Als Signatur wird in diesem Fall eben nur der Namen benutzt.

3.2 Differenz-Ableitung

Die Differenz-Ableitung ist der zweite Schritt zur Berechnung einer symmetrischen Differenz zwischen Modell A und B. In diesem Schritt werden die Objekte identifiziert, die nicht in der Schnittmenge der beiden Modell sind. Die so berechnete Differenz wird im Folgenden als **technische Differenz** bezeichnet. Eine technische Differenz teilt sich zunächst in zwei Klassen ein: `Correspondence` und `Change`. Die zuvor im Matching berechneten Korrespondenzen der Schnittmenge werden durch Objekte der Klasse `Correspondence` dargestellt. Die Summe aller `Correspondence` gibt damit die gemeinsamen Teile von Modell A und Modell B an. Die Teile, die sich von Modell A zu Modell B unterscheiden, werden durch Objekte der Klasse `Change` (Änderung) angegeben. Bei Änderungen an einem Modell unterscheiden wir dabei noch zusätzlich zwischen den drei Grundelementen eines Modells: Objekte, Referenzen und Attribute. Objekte und Referenzen können in ein Modell sowohl eingefügt als auch entfernt werden. Ausgehen davon,

dass wir keine mehrwertigen Attribute betrachten, sind hingegen einwertige Attribute durch die Metaklassen fest vorgegeben und können daher nur einen neuen Wert zugewiesen bekommen. Zu diesem Zweck werden folgende Klassen definiert:

- `Add-Object`: In Modell B existiert ein Objekt, welches nicht in Modell A existiert. Das Objekt wurde also in Modell B hinzugefügt.
- `Remove-Object`: In Modell A existiert ein Objekt, welches nicht in Modell B existiert. Das Objekt wurde also in Modell B entfernt.
- `Add-Reference`: In Modell B existiert eine Referenz, welche nicht in Modell A existiert. Die Referenz wurde also in Modell B hinzugefügt.
- `Remove-Refernce`: In Modell A existiert eine Referenz, welche nicht in Modell B existiert. Die Referenz wurde also in Modell B entfernt.
- `Attribute-Value-Change`: Für alle Objekte aus Modell A und Modell B, für die eine Korrespondenz existiert, wird überprüft, ob sich der Wert eines Attributs verändert hat und ggf. ein Attribute-Value-Change erzeugt. Es werden keine Attribute-Value-Changes für neue initialisierte Attribute eines Add-Objects angelegt.

  Attribute, deren Metatyp (`EAttribute`) folgende Eigenschaften haben, können dabei vernachlässigt werden, da sich aus diesen Attributen keine direkten Änderungen des Modells ableiten lassen:

  - `changeable = false` → Der Wert kann von außen nicht verändert werden. D.h. es wird keine `setXX()` Methode generiert. [SBPM09] (S. 108)
  - `transient = true` → Der Wert wird bei der Serialisierung übergangen und nicht mit abgespeichert. [SBPM09] (S. 108)
  - `derived = true` → Gibt an, dass dieser Wert aus anderen Informationen abgeleitet wird. Hat aber keinen Einfluss auf die Code Generierung. [SBPM09] (S. 108)

Das in Ecore Implementierte Differenzmodell ist in Abbildung 3.4 zu sehen. Es bildet die Basis für das spätere Semantic-Lifting. Die in dieser Phase berechneten `Changes` werden im Folgenden als **low-level Änderungen** bezeichnet, da die durchgeführten Änderungen am Modell hier noch auf einer rein technischen Ebene angegeben werden.

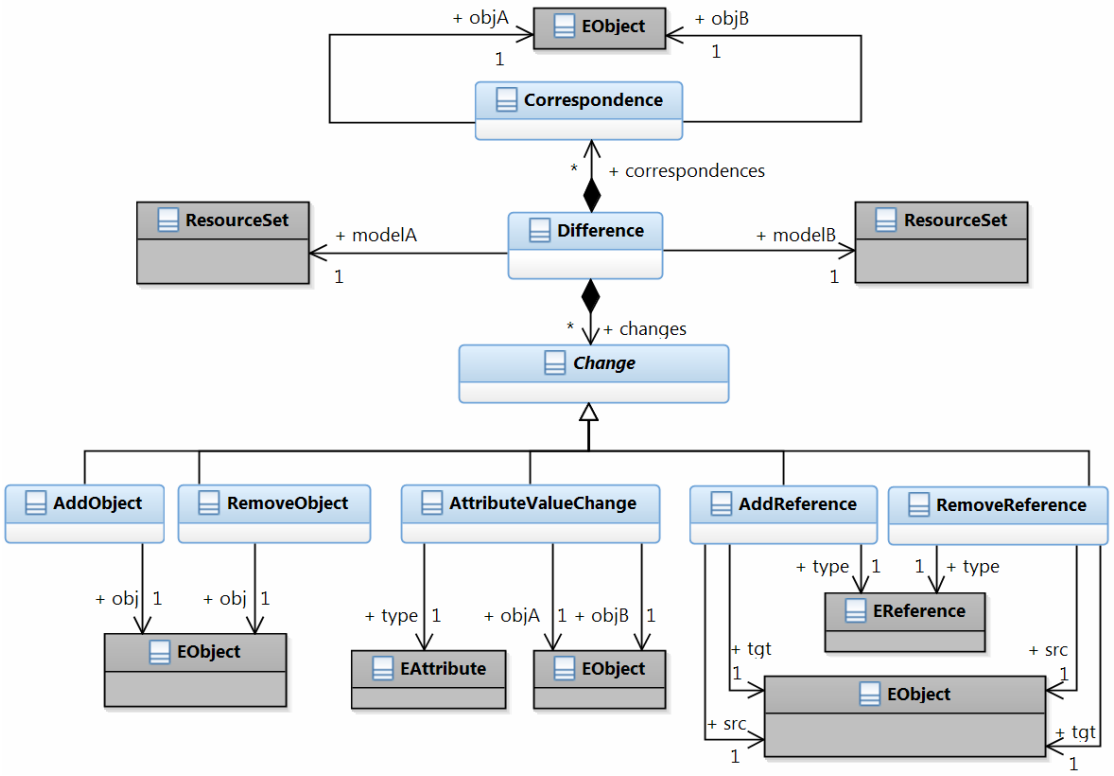


Abbildung 3.4: Differenzmodell

3.3 Semantic-Lifting

Als Eingabe für das Semantic-Lifting dient die zuvor berechnete technische Differenz der Modelle A und B. Die technische Differenz enthält ausschließlich low-level Änderungen, welche angeben, was sich zwischen Modell A und Modell B verändert hat. Da low-level Änderungen auf Basis des Metamodells angegeben werden, sind diese aber für normale Benutzer oft sehr unverständlich. Auch um einen schnellen Überblick über die Veränderungen eines Modells zu bekommen, eignet sich diese Form der Repräsentation nicht besonders gut, da die low-level Änderungen unstrukturiert in der Differenz liegen. Aus Benutzersicht wäre es wünschenswert die low-level Änderungen so zu strukturieren, dass sie den Bearbeitungsprozess eines Editors wiedergeben. Genau dieses Ziel verfolgt das Semantic-Lifting, die technischen Änderungen wieder als Editieroperation auf einer Benutzer verständlichen Ebene darzustellen.

Um eine Editieroperation darzustellen, werden die low-level Änderungen neu strukturiert. Jede angewendete Editieroperation lässt sich auf ein bestimmtes Muster in der

Differenz zurückführen. Dieses Muster besteht sowohl aus low-level Änderungen als auch aus einem bestimmten Kontext. Der Kontext einer Editieroperation gibt zum einen an, auf welche Elemente die Editieroperation angewendet werden soll, zum andern können bestimmte Umstände vorgegeben sein, damit der Vorgang ausgeführt werden darf. Welche Änderungen in welchem Kontext von einer Editieroperation ausgeführt werden, wird durch die s.g. **Editierregel** vorgegeben. Aus dieser Editierregel kann dann das Muster abgeleitet werden, das die Editieroperation in einer Differenz wieder erkennt. Dieses Muster wird als **Erkennungsregel** bezeichnet. Das Muster einer Erkennungsregel überprüft die Differenz sowohl auf auftretende low-level Änderungen als auch auf den Kontext der entsprechenden Editieroperation. Nachdem also eine bestimmte Editieroperation in der Differenz erkannt wurde, können die entsprechenden low-level Änderungen der Editeroperation wieder zugeordnet werden. Um diese semantische Zuordnung von Editieropration zu low-level Änderungen in der Differenz zu speichern, wird eine neu Klasse in unser Differenzmodell eingeführt. Wie in Abbildung 3.5 zu sehen ist, gruppiert ein s.g. **Semantic-Change-Set** mehrere zu einer Editieroperation gehörenden low-level Änderungen (`Change`). Der Name des Semantic-Change-Sets entspricht der assoziierten Editieroperation. Dabei ist darauf zu achten, dass am Ende des Semantic-Lifitngs jede low-level Änderung nur in einem Semantic-Change-Set enthalten ist. Im Semantic-Lifting Konzept wird dies wie folgt formuliert:

> „The objective of semantically lifting a model difference is thus to partition the set of low-level changes into disjoint subsets, each subset containing the changes belonging to exactly one edit operation. These subsets must be disjoint since each low-level change results from the application of exactly one edit operation. We call these subsets **semantic change sets**.“ [KKT11] (S.5)

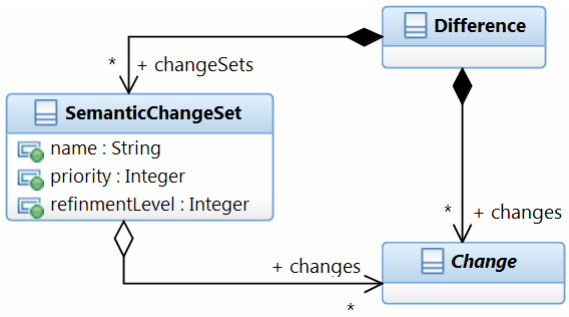


Abbildung 3.5: Semantic-Change-Set

4 Erstellen der Editierregeln

Um Modelle bearbeiten zu können, benötigen wir darauf ausgelegte Werkzeuge. In der Regel werden dazu grafische Editoren oder Transformationstechniken benutzt. Welche Technik zur Bearbeitung eines Modells verwendet wurde und welche Semantik hinter den einzelnen Veränderungen im Modell steckt, ist zum Zeitpunkt der Differenzberechnung nicht mehr direkt nachzuvollziehen. Das Problem ist, dass dem generischen Differenzberechner keine Informationen über die Semantik der möglichen Editieroperationen vorliegt. Um die Semantik wiederherzustellen, müssen wir allerdings wissen, welche Editieroperationen für einen bestimmten Modelltyp existieren und wie diese aussehen. Hierzu werden s.g. Editierregeln verwendet. Die Editierregeln sind die Grundlage für das spätere Semantic-Lifting. Eine Editierregel beschreibt eine Editieroperation, die vom Entwickler auf einem Modell eines bestimmten Typs ausgeführt werden kann. Dabei hat jeder Modelltyp auf Basis seines Metamodells und der Semantik andere Editieroperationen. Daher müssen auch die Editierregeln für jeden Modelltyp einzeln angegeben werden.

4.1 Identifikation von Editierregeln

Zunächst stellt sich an dieser Stelle die Frage, welche Editierregeln benötigt werden, um das vollständige Bearbeiten eines Modells zu ermöglichen. Grundsätzlich können hier natürlich verschiedene Ansätze gewählt werden. Eine Möglichkeit wäre es, falls vorhanden, den Editor für den bestimmten Modelltyp zu analysieren und festzustellen, welche Editieroperationen zur Verfügung gestellt werden. Eine weitere Möglichkeit besteht darin, das eigentliche Metamodell und die damit verbundenen Constraints zur Erstellung der Editierregeln zu verwenden.

Da das Semantic-Lifting ein generischer Ansatz ist, kann zu diesem Zeitpunkt noch keine Aussage über die später in der Praxis verwendeten Metamodelle getroffen werden. Die eigentliche Entscheidung, welche Editierregeln benötigt werden und wie diese aussehen, bleibt damit zum Teil auch dem Entwickler überlassen, der die Semantic-Lifting-Engine für ein bestimmtes Metamodell konfiguriert. Dabei sollte bedacht werden, dass eben nicht immer eindeutig klar ist, auf welche Art das Modell bearbeitet wurde. Unterschiedliche

Editoren können für ein und dasselbe Metamodell unterschiedliche Editieroperationen haben.

Die Editierregeln müssen also alle Editieroperationen nachbilden, die grundlegend möglich wären. Trivial aber entscheidend an dieser Stelle ist, dass später nur solche Editieroperationen geliftet werden, für die auch eine Editierregel oder eine entsprechende Zerlegung von mehreren Editierregeln existiert. Aufgrund dieser Überlegungen erschien eine Analyse des Metamodells zur Ableitung der Editierregeln am sinnvollsten. An Hand des Metamodells lässt sich am besten schematisch und formal nachvollziehen, welche Editierregeln benötigt werden. Allgemein formuliert werden zunächst die Editierregeln benötigt, mit denen für ein Modell neue Elemente erzeugt, bestehende Elemente entfernt oder ggf. abgeändert werden können. Ziel ist es eine Basismenge an Editierregeln zu erzeugen, die sich an folgende Definition anlehnt. Statt *„atomic pattern“* wird hier im Folgenden der Begriff der **Atomic-Editierregel** verwendet:

> „Atomic patterns constitute basic elements for composing modeling patterns. They are governed by two characteristics: On the one hand, they maintain model consistency, of course. On the other hand, they cannot be split into smaller modeling patterns keeping model consistency. Therefore, they form the smallest units of consistent model changes.
>
> By using atomic patterns only, it is possible to create all eligible instances of a specific modeling language. E.g., for each model element defined by the modeling language, at least one particular atomic modeling pattern exists performing the minimum of required modeling steps to create this model element. Otherwise this model element is no eligible element of the modeling language, e.g. due to prohibition by constraints.“ [Wie10] (S.11)

4.2 Editierregeln für Ecore

Betrachtet man das Ecore Metamodell (Abbildung 2.2), so lassen sich 69 Atomic-Editierregel identifizieren. Abstrakte Klassen (nicht deren Attribute und Referenzen), abgeleitete Referenzen und nicht veränderbare Attribute müssen nicht betrachtet werden. Die Regeln sollen wie in der Definition beschrieben, die Konsistenz des Modells aufrecht erhalten. Man muss bedenken, dass die Ecore Editoren auch Editieroperationen zulassen, die das Modell inkonsistent werden lassen. Zum Beispiel können Klassen ohne Namen oder Attribute ohne Typ angelegt werden. Die Atomic-Editierregeln lassen sich wie unten aufgelistet in mehrere Kategorien einteilen. Die Aufteilung in diese Kategorien dürfte

sich aber in der Regel auch auf jeden anderen Modelltyp übertragen lassen.

- **11 × Add <Class>:** Hinzufügen eines Modell-Elements.
- **11 × Remove <Class>:** Entfernen eines Modell-Elements.
- **11 × Move <Class>:** Verschieben eines bestehenden Modell-Elements.
- **5 × Add Reference <Class> <Reference>:** Hinzufügen einer optionalen Referenz zu einem bestehenden Modell-Element.
- **5 × Remove Reference <Class> <Reference>:** Entfernen einer optionalen Referenz zu einem bestehenden Modell-Element.
- **3 × Change Reference <Class>:** Verändern einer benötigten Referenz eines bestehenden Modell-Elements.
- **23 × Attribute Value Change <Class> <Attribute>:** Ändern eines Attributs eines bestehenden Modell-Elements.

Regeln, die nicht in die Kategorie der Atomic-Editierregel fallen, werden im Folgenden als **Advanced-Editierregel** bezeichnet. Eine Advanced-Editierregel kann im Prinzip als eine Zusammensetzung bzw. Spezialisierung von Atomic-Editierregeln betrachtet werden.

4.3 EMF-Refactor

Um die Editierregeln zu formulieren werden Henshin Transformationssysteme (TS) verwendet. Diese Transformationssysteme müssen bestimmten Konventionen entsprechen, damit sie später für das Semantic-Lifting weiterverarbeitet werden können. Die Konventionen entsprechen denen von EMF-Refactor Regeln. EMF-Refactor ist ein Werkzeug um benutzerdefinierte Refactorings für Ecore basierte Modelle anzulegen. Die einmal angelegten Refactorings können dann über den baumbasierten EMF-Editor auf das Modell angewendet werden. Der Unterschied zwischen einem Refactoring und einer Editieroperation besteht darin, dass ein Refactoring nur die Struktur des Modells verändert, nicht aber das Verhalten wie bei einer Editieroperation. Der Aufbau von Editierregeln für Editieroperationen unterscheidet sich aber nicht von dem für Refactorings, daher lässt sich das Konzept problemlos übernehmen. EMF-Refactor definiert drei Henshin Transformationssysteme, aus denen ein Refactoring besteht.

- **Initial-Check-TS** (*Precondition*)**:** Hier werden grundlegende Tests durchgeführt, ob das Refactoring angewendet werden darf. Dazu wird für jede Situation, in der das Refactoring nicht angewendet werden darf, eine Regel formuliert. In diesen Regeln wird genau die Situation dargestellt, die nicht auftreten darf. Das Refactoring darf nur dann ausgeführt werden, wenn keine Initial-Check-Regel zutrifft, bzw. anwendbar ist. Dazu wird eine Independent-Unit verwendet, die alle Regeln enthält. Eine Independent-Unit ist dann anwendbar, wenn genau eine Regel der Unit anwendbar ist. D.h. die Independent-Unit darf zum ausführen des Refactorings nicht anwendbar sein. Theoretisch können auch mehrere Units verschachtelt werden. Um dem EMF-Refactor kenntlich zumachen, welche Unit beim Initial-Check ausgeführt werden soll, muss diese Unit mit dem Namen *mainUnit* gekennzeichnet werden.

- **Final-Check-TS** (*Precondition*)**:** Der Final-Check unterscheidet sich strukturell nicht vom Initial-Check. Auch hier muss es eine Unit mit Namen *mainUnit* geben, die nicht anwendbar sein darf, damit das Refactoring durchgeführt werden kann. Im Unterschied zum Initial-Check wird der Final-Check durchgeführt, nachdem die für das Refactoring benötigten Daten vom Benutzer abgefragt wurden. Hier wird überprüft, ob die eingegebenen Daten für das Refactoring verwendet werden können. Zum Beispiel könnte beim Erstellen einer Klasse überprüft werden, ob eine Klasse mit dem gleichen Namen schon existiert.

- **Execute-TS:** In der Execute Phase wird das eigentliche Refactoring ausgeführt. Die in diesem Transformationssystem enthaltenen Regeln geben an, welche Änderungen an dem Modell durchgeführt werden sollen. Auch hier muss wieder eine *mainUnit* spezifiziert werden.

Für das Semantic-Lifting ist hauptsächlich das Execute-TS von Bedeutung. Das Refactoring für eine bestimmte Modellierungssprache wird hier auf Basis des Metamodells angegeben. Daher lassen sich aus den Regeln die eigentlichen low-level Änderungen ablesen, die durch das Semantic-Lifting wieder erkannt werden sollen. Im Fall des Initial- und Final-Check gehen wir zunächst einmal davon aus, dass, wenn ein angewandtes Refactoring erkannt wurde, auch die Vorbedingungen (*engl. precondition*) für dieses Refactoring erfüllt waren.

Damit die Henshin Transformationssysteme durch die Semantic-Lifting-Engine verarbeitet werden können, müssen diese den EMF-Refactor Konventionen entsprechen. Genauere Angaben und Beispiele sind auf [ERe12] zu finden.

5 Generieren der Erkennungsregeln

Ziel des Semantic-Liftings ist es, die low-level Änderungen der technischen Differenz wieder einer bestimmten Editieroperation zuzuordnen. Die Änderungen einer Editieroperation werden durch die Editierregeln vorgegeben. Das Auslesen der Editieroperation aus der Differenz wird durch die s.g. Erkennungsregeln erledigt. Eine Erkennungsregel ordnet den low-level Änderungen wieder eine Editieroperation zu und gruppieren diese in s.g. Semantic-Change-Sets. Die Erkennungsregeln sind, genau so wie die Editierregel, Henshin Transformationssysteme. Alle Informationen, die für die Erkennung der Editieroperation benötigt werden, sind in der Editierregel vorhanden und deren Semantik ist wohldefiniert. Für jeden Teil einer Editierregel lässt sich eindeutig nachvollziehen, welche Änderungen dadurch in der Differenz auftreten werden. Daher können die Erkennungsregeln vollständig automatisch aus den Editierregeln generiert werden. Diese Aufgabe übernimmt der **Erkennungsregel Generator**.

> „Change set recognition rules are getting complex very quickly. However, they are very schematic and can be automatically generated from their corresponding edit rule.“ [KKT11] (S.6)

5.1 Implizite Kanten

Bevor die eigentliche Generierung der Erkennungsregel beginnen kann, muss die Editierregel ggf. noch um bestimmte Kanten erweitert werden. In Ecore kann einer Referenz eine entgegen gerichtete Referenz (`eOpposite`) zugeordnet werden. In den meisten Fällen reicht es in Henshin aus, wenn eine Referenz nur in eine Richtung angegeben wird. Die entgegen gerichtete Referenz wird dann, z.B. beim erzeugen einer Referenz, automatisch mit eingefügt.

Im Beispiel in Abbildung 5.1 wird eine neue Klasse in ein Paket eingefügt. Beim einfügen neuer Objekte in ein Ecore Modell wird typischerweise eine Referenz sowohl zwischen Container und Objekt als auch zwischen Objekt und Container erzeugt. Angegeben wurde in diesem Fall aber nur die Referenz `eClassifiers` vom Container zum Objekt. Diese

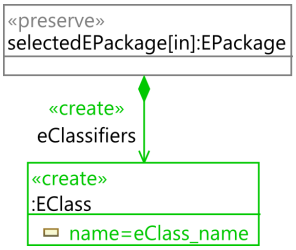


Abbildung 5.1: Ecore Klasse einfügen

Referenz besitzt aber eine entgegen gerichtete Referenz `ePackage` vom Objekt zum Container. Eben genau diese low-level Änderungen werden nach Anwendung dieser Regel in der technischen Differenz auftreten: 2 × Add-References und 1 × Add-Object.

Das Problem solcher **impliziten Kanten** kann wie schon angedeutet in Ecore häufiger auftreten. Damit alle Änderungen durch die Erkennungsregel wieder vollständig zugeordnet werden können, müssen zunächst alle impliziten Kanten in den Transformationsgraphen eingefügt werden. Dazu werden folgende Schritte für alle Kanten im Graphen ausgeführt:

1. Besitzt der Typ (`EReference`) der aktuellen Kante X eine entgegen gerichtete Referenz (`eOpposite`)?
2. Ist dies der Fall, dann überprüfe, ob eine Kante Y existiert, die in entgegengesetzter Richtung zur Kante X verläuft und vom entsprechenden `eOpposite` Typ ist.
3. Wurde Kante Y nicht gefunden, dann füge sie dem Graphen hinzu.

Nachdem alle impliziten Kanten eingefügt wurden, kann mit der eigentlichen Erkennungsregel Generierung begonnen werden.

5.2 Parameter

Die Parameter der Editierregel werden zu Beginn vollständig in die Erkennungsregel übernommen. Sie lassen sich dabei in zwei Kategorien einteilen:

- **Regel-Parameter:** Parameter wird innerhalb der Regel durch ein Attribut eines Knoten initialisiert.
- **Unit-Parameter:** Parameter wird von außen an die Regel übergeben. D.h es wird ein benutzerdefinierter Wert vor Aufruf der Regel gesetzt. Dies ist im Fall einer

EMF-Refactor Regel immer ein Parameter, der auf einen Parameter der *mainUnit* gemappt wurde.

Diese Unterscheidung spielt innerhalb der Erkennungsregel beim vergleichen der Attributwerte eine Rolle. Der ursprüngliche Wert eines Unit-Parameters, der der Editierregel übergeben wurde, ist in der Erkennungsregel nicht mehr bekannt. Problematisch wird dies dann, wenn der Wert des Unit-Parameters durch einen bestimmten Ausdruck mit anderen Parametern oder Werten verrechnet wurde (z.B. `+,-,*,/`). Um den Wert des Unit-Parameter wieder zu extrahieren, würde man die entsprechende Umkehrfunktion zu diesem Ausdruck benötigen, bzw. man müsste diesen automatisch berechnen. In Henshin ist es allerdings nur möglich, Attributwerte auf einfache Gleichheit zu überprüfen. Reguläre Ausdrücke oder ähnliches, um z.B. eine String Konkatenation zu überprüfen, sind nicht möglich. Ausdrücke in denen Unit-Parameter verrechnet werden, werden daher bei der Generierung bisher übergangen.

5.3 Generierungsmuster

Um aus einer Editierregel eine Erkennungsregel zu generieren, werden jeweils die einzelnen Elemente eines Henshin Transformationssystems betrachtet. Ein Transformationssystem besteht aus einem linken (*Left-Hand Site* (LHS)) und einem rechten Graph (*Right-Hand Site* (RHS)). Dieser besitzt wiederum: Knoten (*engl. Node*), Kanten (*engl. Edge*) und Attribute (*engl. Attribute*). Alle diese Elemente können sowohl nur auf der linken (`«delete»`) als auch nur auf der rechten Seite (`«create»`) oder auch auf beiden Seiten (`«preserve»`) der Regel vorkommen. Diese Stereotype werden im Folgenden verwendet, um die Zuordnung eines Henshin Elements zu einer bestimmten Seite der Regel auszudrücken. In der grafischen Henshin Notation dargestellt ergeben sich damit die folgenden Muster, um Elemente einer Editierregel in eine Erkennungsregel zu transformieren. Die Editierregel Muster sind jeweils links und die Erkennungsregel-Muster jeweils rechts dargestellt. Das fiktive Metamodell in Abbildung 5.2 dient als Beispiel für die abgebildeten Muster.

Abbildung 5.2: Fiktives Metamodell

- **Remove-Object-Pattern:** Für jeden Knoten, der nur auf der linken Seite der Regel vorkommt, wird das Muster in Abbildung 5.3 erzeugt. Ein Remove-Object zeigt auf ein Objekt aus Modell A vom entsprechenden Typ.

Abbildung 5.3: Remove-Object-Pattern

- **Add-Object-Pattern:** Analog zum Remove-Object-Pattern wird das Add-Object-Pattern für jeden Knoten angelegt, der nur auf der rechten Seite der Regel vorkommt. Das referenzierte Objekt ist in diesem Fall aus Modell B.

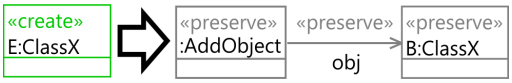


Abbildung 5.4: Add-Object-Pattern

- **Correspondence-Pattern:** Innerhalb der Differenz werden durch dieses Muster genau die Teile ausgewählt, die sich von Modell A zu Modell B nicht verändert haben. Dazu wird durch einen **Correspondence-Knoten** ein Objekt aus Modell A mit dem entsprechenden Objekt aus Modell B verbunden. Siehe Abbildung 5.5. Dieses Muster tritt immer dann auf, wenn ein Knoten auf beiden Seiten der Editierregel vorkommt (`«preserve»`). Knoten, die Objekte aus Modell A bzw. Modell B repräsentieren, werden im Folgenden als **Modell A Knoten** bzw. **Modell B Knoten** bezeichnet.

- **Preserved-Reference-Pattern:** Genau so wie `«preserve»` Knoten in der Editierregel durch `«preserve»` Kanten verbunden werden, werden auch die entsprechend Modell A und B Knoten der Correspondence-Patterns verbunden.

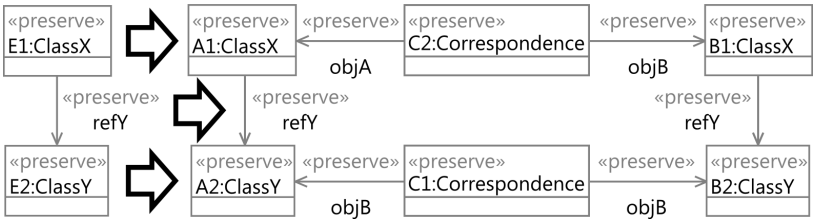


Abbildung 5.5: Correspondence- & Preserved-Reference-Pattern

- **Remove-Reference-Pattern:** Als nächstes werden nun die Kanten betrachtet, die nur auf der linken Seite (`«delete»`) der Regel vorkommen. Eine Remove-Reference verbindet durch Quelle (`src`) und Ziel (`tgt`) immer zwei Modell A Knoten. D.h., werden die Muster beim generieren in der hier beschriebenen Reihenfolge abgearbeitet, so wurden die Knoten $A1$ und $A2$ bereits durch Correspondence-Patterns oder durch Remove-Object-Patterns erzeugt, da sich die Muster an dieser Stelle überschneiden. Welche Muster sich überschneiden, hängt eben davon ab, ob die Knoten der Editierregel `«preserve»` oder `«delete»` Knoten sind. Der Fall eines `«create»` Knoten in Verbindung mit einer `«delete»` Kante ist nicht möglich und muss daher nicht betrachtet werden.

  Neben Quelle und Ziel der Editierregel Kante muss in der Erkennungsregel noch der Typ betrachtet werden. Dazu wird das Meta-Metamodell, also Ecore, verwendet und der Typ der Remove-Reference auf eine `EReference` mit dem entsprechenden Namen geprüft. Grundsätzlich könnte es vorkommen, dass eine Referenz mit dem gleichen Namen im Metamodell mehrfach existiert. Der Kontext für die Referenz wird aber eindeutig durch die Quelle (`src`) der Remove-Reference vorgegeben. Ein solcher Knoten zur Überprüfung des Typs wird im Folgenden als **Typknoten** bezeichnet.

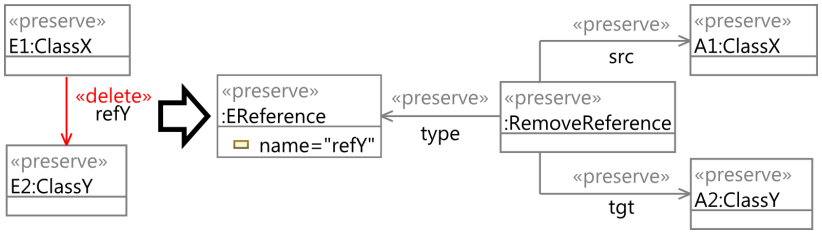


Abbildung 5.6: Remove-Reference-Pattern

- **Add-Reference-Pattern:** Für das Add-Reference-Pattern werden analog zum Remove-Reference-Pattern die Kanten betrachtet, die nur auf der rechten Seite (`«create»`) der Editierregel vorkommen.

  Innerhalb der Editierregel können `«create»` Kanten nur zwischen `«create»` oder `«preserve»` Knoten auftreten. Damit zeigen die `src` und `tgt` Kanten des Add-Reference Knoten immer auf Modell B Koten. Entsprechend dem Remove-Reference-Pattern wurden die Modell B Knoten $B1$ und $B2$ bereits von Correspondence-Patterns oder von Add-Object-Patterns erzeugt.

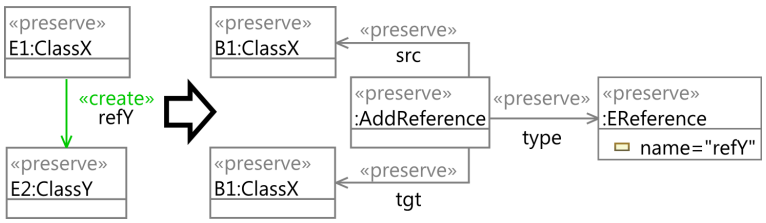


Abbildung 5.7: Add-Reference-Pattern

- **Attribute-Value-Change-Pattern:** Dieses Muster wird nur auf Attribute angewendet, die Teil eines `«preserve»` Knoten sind. Für Attribute, die beim neu Anlegen von Objekten gesetzt werden, werden bei der Differenz-Ableitung keine Attribute-Value-Changes erzeugt. D.h. `«create»` Knoten und deren Attribute müssen nicht beachtet werden.

  Daher verbindet das Attribute-Value-Change-Pattern immer ein Objekt aus Modell A mit dem entsprechenden Objekt aus Modell B. Das bedeutet für diesen Fall, dass die Knoten $A1$ und $B1$ bereits durch ein Correspondence-Pattern angelegt wurden. Der Typ des Attributs wird, ähnlich wie der Referenztyp beim Remove-Reference-Pattern, durch einen `EAttribute` Typknoten mit Hilfe des Namens überprüft.

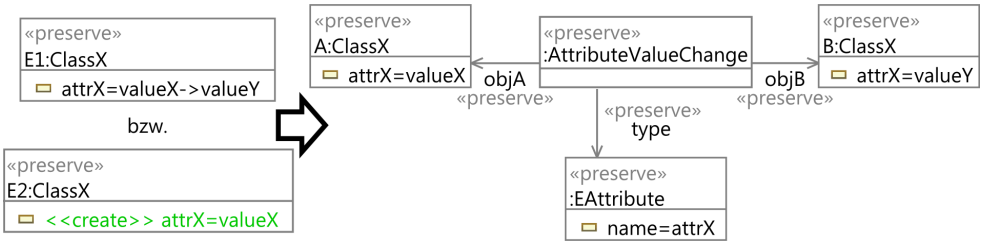


Abbildung 5.8: Attribute-Value-Change-Pattern

Zusätzlich wird in der Erkennungsregel noch der Wert, der in der Editierregel angegebenen Attribute überprüft. Handelt es sich um feste primitive Werte, also z.B. Zeichenketten oder Zahlen, kann damit im ersten Fall überprüft werden, ob sich der Wert im Modell A Objekt von *valueX* zu *valueY* im Modell B Objekt verändert hat. Im zweiten Fall tritt das Attribut nur auf der rechten Seite der Regel (`«create»`) auf, hier würde das Attribut nur für den Modell A Knoten (Knoten $A1$) angelegt und überprüft.

Handelt es sich hingegen bei den Henshin Attributwerten um Parameter, so würde dadurch bei mehrfachem Auftreten eines Parameters in verschiedenen Attributen

überprüft, ob die jeweiligen Werte gleich sind. Neben einzelnen primitiven Werten und Parametern können für Henshin Attribute auch komplexere *JavaScript* Ausdrücke mit Parametern, Werten und Operatoren angelegt werden. Für solche Ausdrücke gelten die zuvor im Abschnitt 5.2 beschriebenen Einschränkungen für Unit-Parameter. Dies gilt auch für alle folgenden Attribute-Value-Patterns.

- **Add-Attribute-Value-Pattern:** Wie bereits erwähnt werden für neu angelegte Objekte keine Attribute-Value-Changes berechnet. Das Attribut wird aber wieder für das entsprechende Modell B Objekt geprüft.

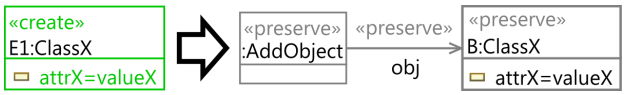


Abbildung 5.9: Add-Attribute-Value-Pattern

- **Remove-Attribute-Value-Pattern:** Dieses Muster wird analog zum Add-Attribute-Value-Pattern für aus Modell A entfernte Objekte angelegt.

Abbildung 5.10: Remove-Attribute-Value-Pattern

- **Preserved-Attribute-Value-Pattern:** Beim Preserved-Attribute-Value-Pattern wird überprüft, dass sich ein Wert von Modell A zu Modell B nicht ändert. Dazu wird das gleiche Attribut sowohl für den Modell A Knoten als auch für Modell B Knoten angelegt. Ein `«delete»` Attribut in einem `«preserve»` Knoten verhält sich dabei in Henshin ganuso wie ein `«preserve»` Attribut.

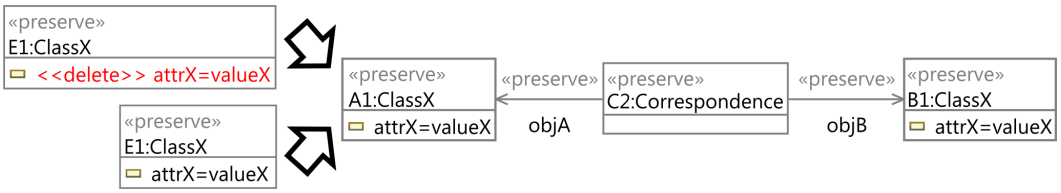


Abbildung 5.11: Preserved-Attribute-Value-Pattern

- **Semantic-Change-Set-Pattern:** Abschließend muss noch der **Differenz-Knoten** und der **Semantic-Change-Set-Knoten** angelegt und mit allen **Änderungsknoten** (Add/Remove-Object-Knoten, Add/Remove-Reference-Knoten und

Attribute-Value-Change-Knoten) verbunden werden. Für das Semantic-Change-Set werden noch die drei folgenden Attribute angelegt und initialisiert: `name`, `priority` und `refinementLevel`. Der Name des Semantic-Change-Sets entspricht der ursprünglichen Editierregel. Auf die Attribute `priority` und `refinementLevel` wird später im Rahmen des Abschnitts 9 Post-Processing noch genauer eingegangen.

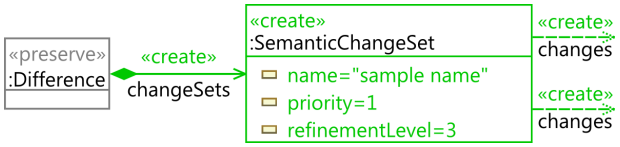


Abbildung 5.12: Semantic-Change-Set-Pattern

5.3.1 Traces

Grundsätzlich ist die Reihenfolge, in der die Muster abgearbeitet werden, voneinander unabhängig. Da die einzelnen Muster aber über die Modell A und B Knoten zusammenhängen, bietet es sich an zunächst die Remove/Add-Object- und Correspondence-Patterns zu erzeugen. Damit eine korrekte Zuordnung der sich überschneidenden Knoten zwischen den einzelnen Muster möglich ist, muss abgespeichert werden, welcher Modell A und B Knoten der Erkennungsregel welchem Knoten der Editierregel entspricht. Diese Zugehörigkeit von Knoten wird im Folgenden als **Trace** bezeichnet.

5.3.2 Typknoten

Eine weitere Überschneidung kann bei den Typknoten der Remove-Reference-Patterns und Add-Reference-Patterns oder beim Attribute-Value-Change-Pattern auftreten. Da der Henshin Graph injektiv auf den Arbeitsgraphen abgebildet wird, darf auch jeder Typknoten, der ein bestimmtes Element des Metamodells darstellt, nur einmal erzeugt werden.

5.3.3 Generierungsbeispiel

Die Muster müssen auf alle Elemente in einer Editierregel angewandt werden, um die zugehörige Erkennungsregel zu generieren. In Abbildung 5.13 ist die generierte Erkennungsregel (unten) zu der Editierregel *add empty EClass* (oben) abgebildet. In diesem Fall lassen sich vier Generierungsmuster identifizieren:

- 1 × Correspondence-Pattern markiert mit (1).
- 2 × Add-Reference-Pattern. Die Referenz `eClassifiers` wurde mit (2) markiert. Die Referenz `ePackage` ist eine implizite Kante und daher nicht in der Editierregel zu sehen.
- 1 × Add-Object-Pattern markiert mit (3).

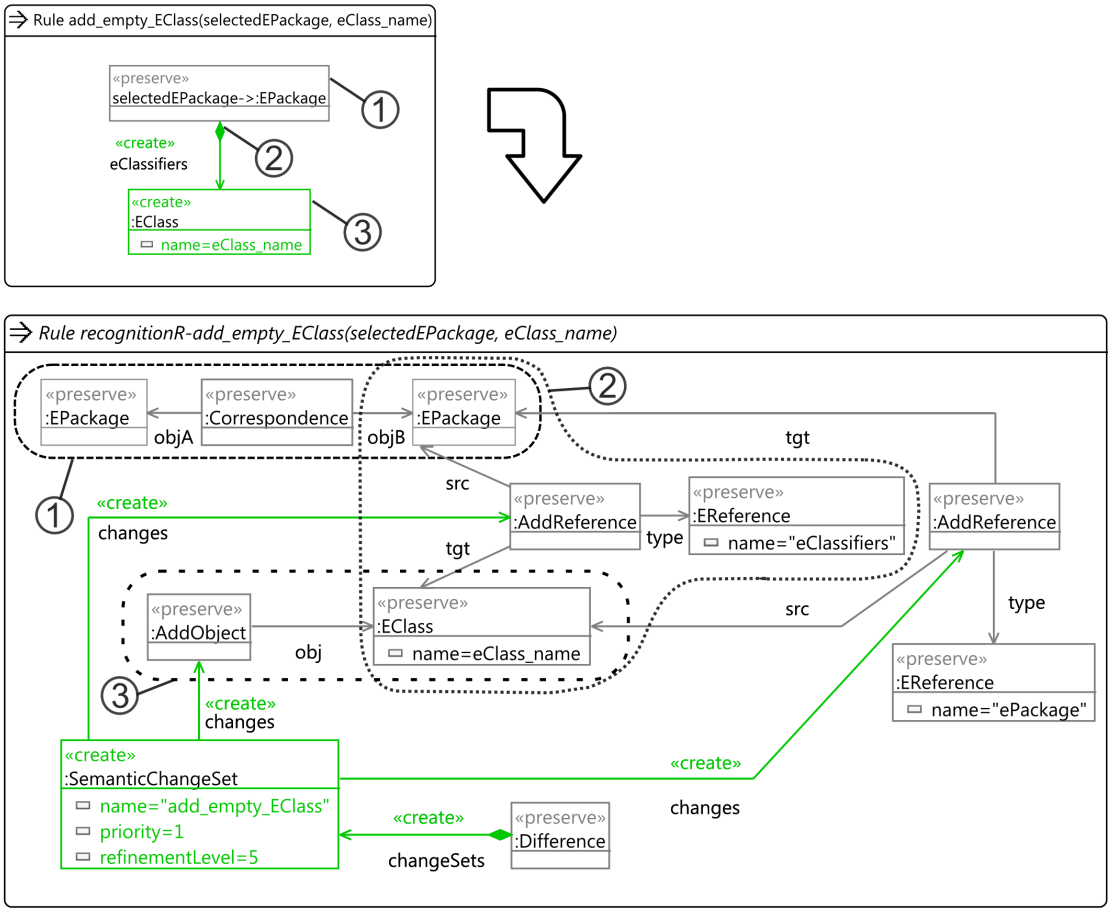


Abbildung 5.13: Add `EClass` Erkennungsregel

5.4 Amalgamation Units

Eine Amalgamation Unit besteht immer aus genau einer Kernregel und beliebig vielen Multiregeln. Um eine Amalgamation Unit von einer Editierregel in eine Erkennungsregel

zu transformieren, muss die Kernregel und jede einzelne Multiregel nach dem zuvor beschriebenen Verfahren in eine Erkennungsregel transformiert werden. `editR2RecognR()` im Pseudocode in Abbildung 5.14

```
function editAU2RecognAU(AmalgamationUnit eau) {
  AmalgamationUnit rau = new AmalgamationUnit();
  rau.kernel = editR2RecognR(eau.kernel);
  for each Rule em ∈ eau.multi {
    rau.multi.add(editR2RecognR(em));
    embedKernel(rau, em);
  }
  return rau;
}
```

Abbildung 5.14: Amalgamation Unit Generierung [KKT11] (S.7)

Der nächste Schritt ist dann das Erzeugen der Mappings zwischen Kernregel und Multiregeln. `embedKernel()` im Pseudocode in Abbildung 5.14. D.h. für jeden Knoten einer Multi-Erkennungsregel wird ein Mapping auf einen Knoten der Kern-Erkennungsregel angelegt. Das Mapping zwischen einer Multi-Erkennungsregel und einer Kern-Erkennungsregel lässt sich nur dann vollständig angeben, wenn die entsprechende Multi-Editierregel die Kern-Editierregel als Teilgraph enthält. Dies gilt sowohl für Knoten als auch für Kanten.

Das Problem beim Anlegen der Mappings besteht darin, dass aus einem Knoten einer Editierregel immer mehrere Knoten in der Erkennungsregel resultieren. Für alle Kanten besteht das gleiche Problem. Außerdem existieren für Kanten keine expliziten Mappings. Es wäre also rein technisch nicht unmöglich in einer Multiregel eine Kante anzulegen, die nicht in der Kernregel vorkommt. Dies würde aber bei der Generierung zu einigen technischen Problemen führen und ist deshalb zu vermeiden. Grundsätzlich wird dadurch die Funktionalität der Amalgamation Unit auch nicht eingeschränkt. Dies lässt sich nachvollziehen, wenn man bedenkt, dass beim Ausführen einer Amalgamation Unit die Multiregel im Prinzip mit der Kernregel (mehrfach) verschmolzen wird. Es spielt daher keine Rolle, ob eine Kante der Multiregel zuvor in der Kernregel vorhanden war oder nicht.

Um die Mappings der Erkennungsregel zu erzeugen, werden die Mappings der Editierregel zunächst einem bestimmten Generierungsmuster zugeordnet. Mappings von `«preserve»` Knoten werden dem Correspondence-Pattern, Mappings von `«delete»` Knoten werden dem Remove-Object-Pattern und Mappings von `«create»` Knoten werden dem Add-Object-Pattern zugeordnet. Zu jedem Mapping der Editierregel können nun

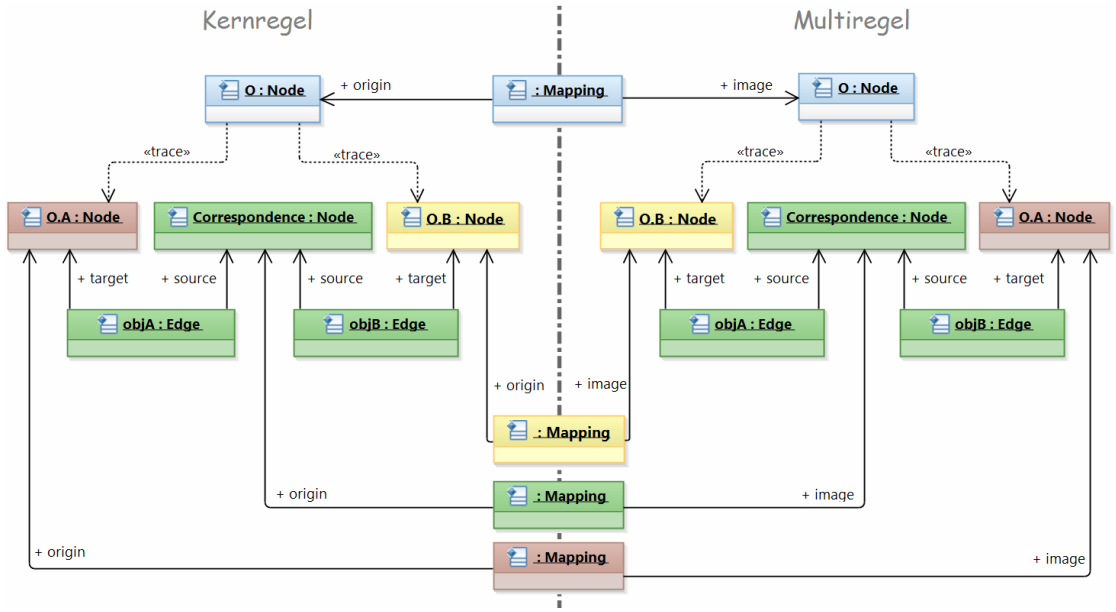


Abbildung 5.15: Amalgamation Mapping (LHS bzw. RHS Correspondence-Pattern)

über die Traces (Abschnitt 5.3.1) die entsprechenden Modell A und B Knoten der Erkennungsregel zugeordnet und das entsprechende Mapping angelegt werden. Anschließend werden noch die zu den jeweiligen Mustern gehörenden Änderungsknoten bzw. Correspondence-Knoten gemappt.

Abbildung 5.15 demonstriert diesen Vorgang für zwei `«preserve»` Knoten *O* einer Amalgamation Editierregel. In der Abbildung wird nur eine der beiden Seiten der Regel dargestellt. Für alle `«preserve»` Knoten unterscheidet sich aber die LHS nicht von der RHS. Wie zu sehen ist, sind die Knoten *O* über ein Mapping verbunden. Von beiden Knoten sind dann über die Traces die entsprechenden Erkennungsregel Knoten zu erreichen, die aufeinander gemappt werden müssen. Die beiden Correspondence-Knoten sind über die Kanten `objA` und `objB` zu erreichen und müssen ebenfalls gemappt werden.

Die Typknoten des Add/Remove-Object und Attribute-Value-Change-Patterns lassen sich relativ leicht zuordnen, da diese per Defintion in der Erkennungsregel eindeutig sein müssen. Somit können immer die gleichen Typknoten aus Kern- und Multiregel aufeinander gemappt werden.

Der Änderungsknoten eines Add-Reference, Remove-Reference oder Attribute-Value-Change-Patterns kann immer dann zwischen Kern- und Multiregel gemappt werden, wenn die Typknoten (`type`) und Modell Knoten (`src, tgt`) des Musters ebenfallls aufeinander gemappt wurden.

Damit später nur ein Semantic-Change-Set für die Amalgamation Unit erzeugt wird, müssen auch die Semantic-Change-Set- und Differenzknoten gemappt werden.

5.5 Generierung als Higher-Order-Transformation

Neben der in Java implementierten Version der Erkennungsregel Generierung wurde der Algorithmus auch als Henshin Transformation angelegt. Da Henshin selbst auf Ecore basiert, ist es technisch problemlos möglich ein Henshin Transformationssystem wieder durch ein anderes Henshin Transformationssystem zu transformieren. Man spricht in diesem Fall auch von einer Higher-Order-Transformation (HOT) [TJF+09]. Dies wird in Abbildung 5.16 illustriert. Hier ist zu sehen, dass sowohl Eingabe (Editierregel) als auch Ausgabe (Erkennungsregel) der HOT *editR2recognitionR* ein Henshin Transformationssystem ist.

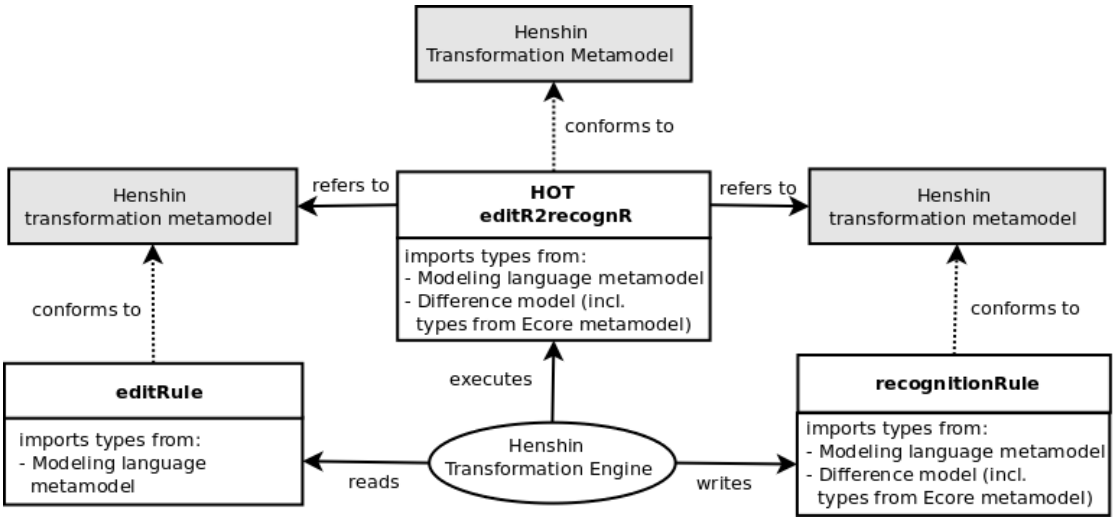


Abbildung 5.16: Higher-Order-Transformation: Editierregel → Erkennungsregel

Um die einzelnen Regeln möglichst noch überschaubar und grafisch darstellbar zu halten, wurde der Algorithmus in fünf Phasen unterteilt. Dies hat außerdem den Vorteil, dass die Regeln nachvollziehbar und besser wartbar bleiben. Die sequentielle Abarbeitung dieser Phasen ist in Abbildung 5.17 in Form von Henshin Units zu sehen[1].

[1]Das vollständige Transformationssystem ist verfügbar unter: http://pi.informatik.uni-siegen.de/Projekte/sidiff/pipeline/semantic-lifting/hot/index.htm

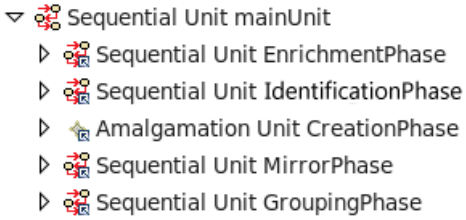


Abbildung 5.17: Henshin Main-Unit

Das Problem besteht darin, dass in den meisten Fällen für die Generierung der Muster immer die rechte und linke Seite der Editierregel betrachtet werden muss und aus diesen Informationen dann wieder ein rechter und linker Teil der Erkennungsregel resultiert. Auf Grund der Beschaffenheit des Henshin Metamodells 2.3 müssten bei einer direkten Generierung der beschriebenen Muster häufig sehr viele Objekte gleichzeitig betrachtet werden.

Grundsätzlich ist es in Henshin schwierig, Informationen innerhalb des Transformationssystems temporär zwischen zu speichern. Um die Information zwischen den verschiedenen Regeln weiter zu reichen, wird ein für diesen Zweck angelegtes Ecore Modell als Hilfsstruktur verwendet, das s.g. **Lifting-Modell**. Wie in Abbildung 5.18 zu sehen ist, gibt es eine zentrale Klasse `Edit2Recognition`, die alle benötigten Information während der Transformation verwaltet. Als Startkonfiguration für die Transformation *editR2recognitionR* wird ein Objekt der Klasse `Edit2Recognition` mit der Editierregel und einer leeren Erkennungsregel initialisiert.

5.5.1 Enrichment-Phase

Die erste Regel (`Create-Implicite-Edge`) des Transformationssystems wird direkt auf die Editierregel angewandt. Hierdurch werden die zuvor beschriebenen impliziten Kanten (Abschnitt 5.1) in die Editierregel eingefügt.

Der nächst Schritt dient dazu, den Umgang mit dem Transformationssystem der Editierregel zu erleichtern. Für `«preserve»` Knoten werden explizit Mappings für die Graphen angelegt, für Kanten ist dies nicht der Fall. Um später einfacher herausfinden zu können, ob eine Kante `«preserve»` ist, wird durch die Regel `Map-Preserved-Edge` für jede Kante, die sowohl auf der rechten als auch auf der linken Seite einer Regel vorkommt, ein Edge-Mapping angelegt. Das Edge-Mapping wird durch das Lifting-Modell abgespeichert und ist somit für alle weiteren Regeln verfügbar.

Wie in Abbildung 5.19 zu sehen ist, müssen erst alle impliziten Kanten angelegt wer-

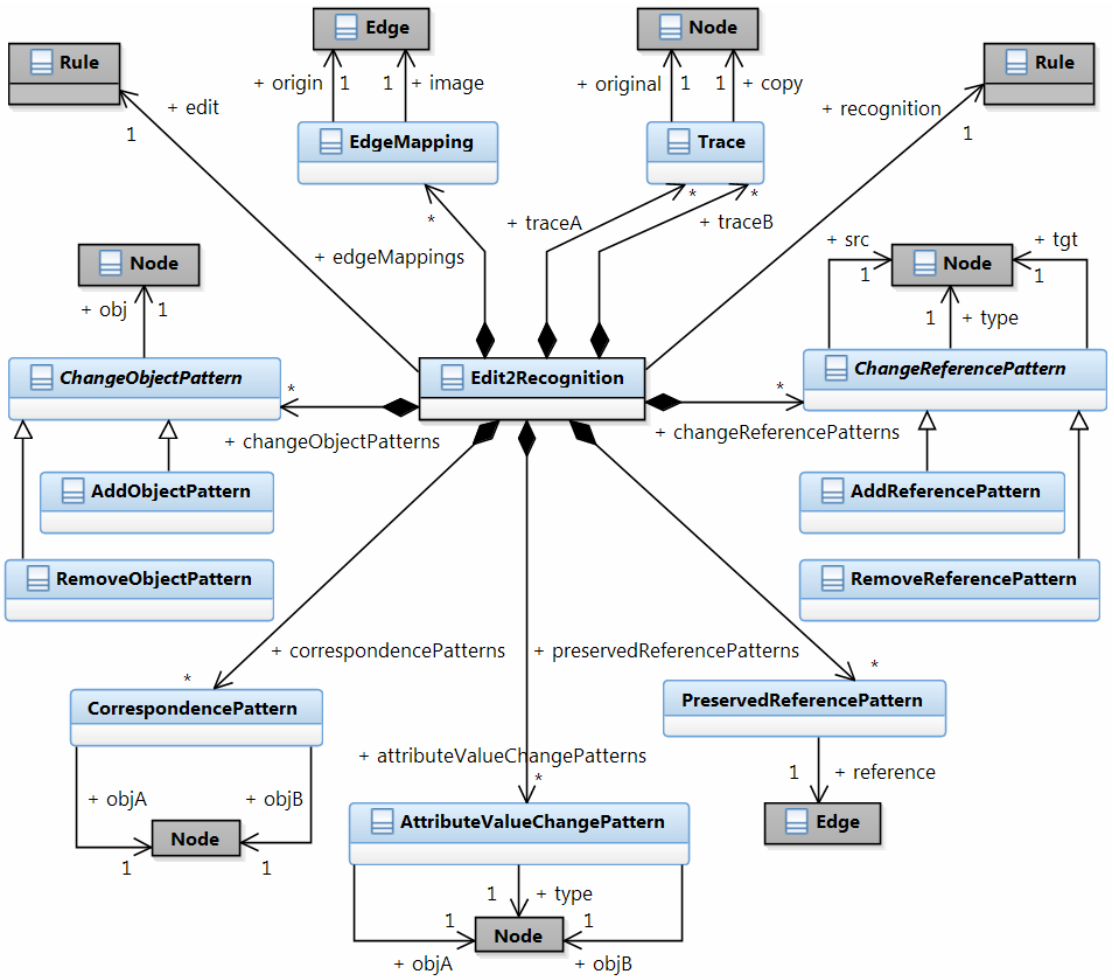


Abbildung 5.18: Lifting-Modell

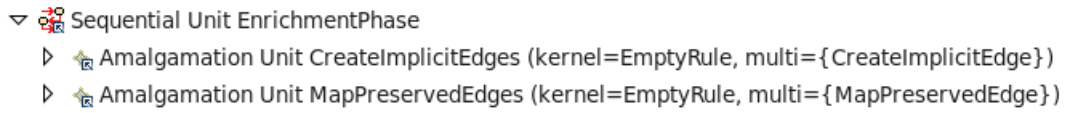


Abbildung 5.19: Henshin Enrichment-Unit

den bevor das Mapping der Kanten gestartet werden kann. Durch die Verwendung von Amalgamation Units werden die beiden Regeln jeweils parallel auf alle Matches angewandt. Die leere Kernregel bewirkt, dass die Mulitregeln für jeden gefundenen Match einmal in die temporär angelegte Amalgamation Regel kopiert werden.

5.5.2 Identification-Phase

In der ersten Phase werden die Informationen für die einzelnen Generierungsmuster gesammelt. Ziel ist es zunächst einmal nur die wirklich benötigten Informationen abzuspeichern, um die einzelnen Regeln möglichst klein zu halten. Im Lifting-Modell gibt es daher eine Klasse für jedes Generierungsmuster (Abschnitt 5.3). Genauso existiert eine Matching-Regel für jedes Muster. Für jeden Match, der in der Editierregel gefunden wird, wird dann ein Objekt der entsprechenden Klasse im Lifting-Modell angelegt.

Da sich die Modell A und B Knoten in den verschiedenen Mustern überschneiden, muss es eine Möglichkeit geben einen Erkennungsregel Knoten einem entsprechenden Knoten der Editierregel zuzuordnen. Diese Aufgabe wird durch die bereits erwähnten Traces (Abschnitt 5.3.1) erledigt. Ein Trace ordnet immer einem Knoten der Editierregel (`original`) einen Knoten der Erkennungsregel (`copy`) zu. Die Traces werden nach Modell A und B Knoten unterschieden im Lifting-Modell abgespeichert.

Das anlegen aller Modell A und B Knoten und Traces erfolgt in der ersten Unterphase der Identification-Phase (Abbildung 5.20 IdentificationPhase01 (links)). Die Überschneidenden Muster werden erste in der zweiter Unterphase (Abbildung 5.20 Identification-Phase02 (links)) angelegt.

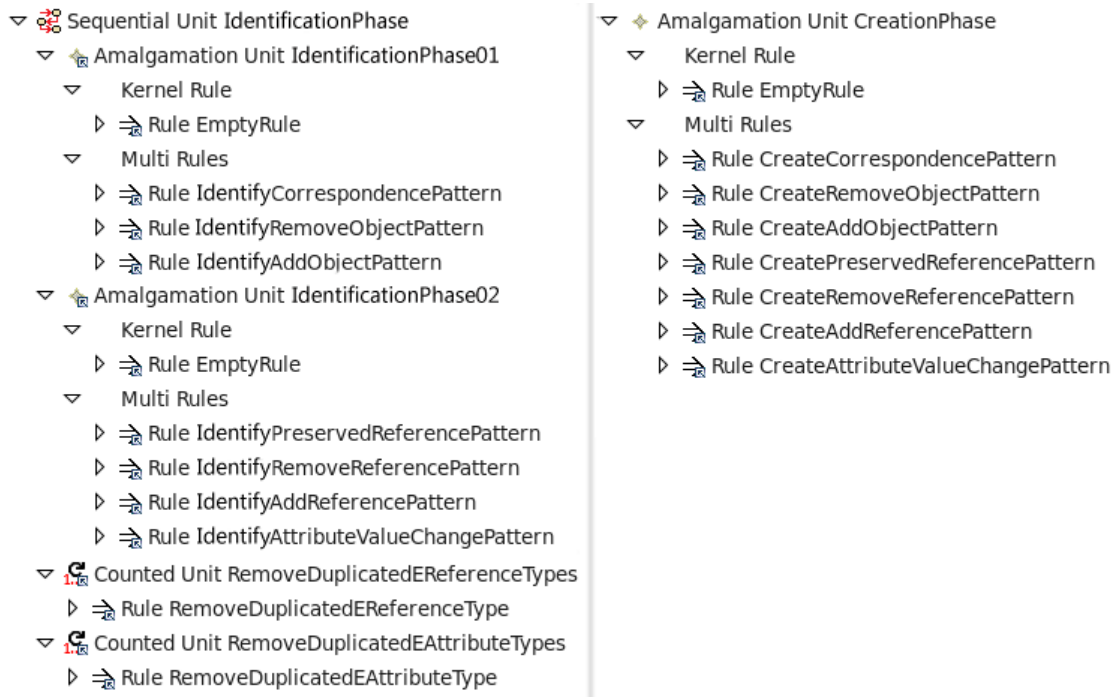


Abbildung 5.20: Henshin Identification- und Create-Unit

Die Typknoten `ChangeReferencePattern.type, AttributeValueChangePattern.type` werden beim ausführen der Amalgamation Unit der Muster Remove/Add-Reference und Attribute-Value-Change-Pattern parallel angelegt. Wie in Abschnitt 5.3.2 beschrieben, müssen diese Knoten aber in der Erkennungsregel eindeutig sein. Bis zu diesem Zeitpunkt ist es noch möglich, dass ein Typknoten mehrfach in den im Lifting-Modell angelegten Muster Objekten vorkommt. Um dieses Problem zu beheben, werden durch die Regeln `Remove-Duplicated-Types` nachträglich alle duplizierten Typknoten aus den Muster Objekten entfernt und alle `type` Kanten auf eine einzelne Instanz gesetzt. Dazu werden die beiden Regeln, wie in Abbildung 5.20 zu sehen, als Counted-Unit (`count = -1`) solange ausgeführt bis kein Duplikat mehr gefunden wird.

Betrachtet man eine Erkennungsregel, so sind der linke und rechte Graph der Regel, abgesehen vom Semantic-Change-Set, identisch (`«preserve»`). Es reicht daher aus zunächst nur einen einfachen Graphen zu erzeugen. Dieser Graph kann dann später einer Seite der Regel zugeordnet und auf die andere Seite gespiegelt bzw. kopiert werden.

5.5.3 Create-Phase

In der nächsten Phase werden für alle gesammelten Muster im Lifting-Modell die noch fehlenden Änderungsknoten (`Correspondence, Add/Remove-Object, Add/Remove-Reference`) und die dazu gehörigen Kanten eingefügt. Außerdem werden sämtliche Knoten und Kanten einem Graphen und der Graph der linken Seite der Regel zugeordnet. Wie in Abbildung 5.20 (rechts) zu sehen, ist dieser Prozess für alle Muster voneinander unabhängig und kann damit parallel ausgeführt werden.

5.5.4 Mirror-Phase

Da bisher nur der linke Teil der Erkennungsregel angelegt wurde, muss im nächsten Schritt der linke Graph vollständig in den rechten Graph kopiert werden. Zusätzlich müssen zwischen allen Knoten der linken und rechten Seite Mappings angelegt werden. Hierzu gibt es je eine Regel für Knoten, Kanten und Attribute. Die Abarbeitung der Regeln ist in Abbildung 5.21 (links) zu sehen.

5.5.5 Grouping-Phase

Nachdem der `«preserve»` Teil der Erkennungsregel erzeugt wurde, muss abschließend noch das Semantic-Change-Set erzeugt und anschließend mit allen low-level Änderungsknoten verbunden werden. Siehe Abbildung 5.21 (rechts).

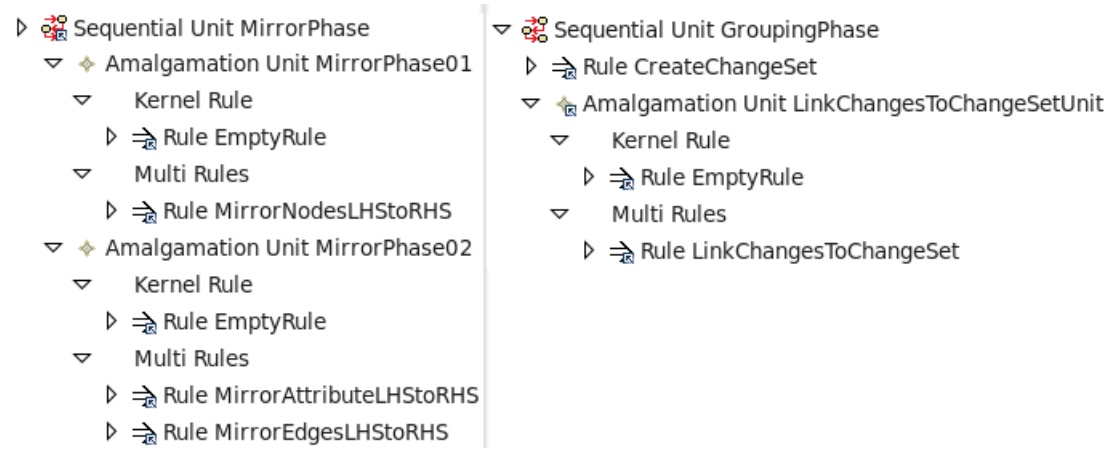


Abbildung 5.21: Henshin Mirror- und Grouping-Unit

5.6 Vergleich von HOT und Java Implementierung

Die Implementierung des Erkennungsregel Generators unter Java[2] wurde unabhängig von der Henshin Implementierung geschrieben. Ein Vergleich beider Ansätze zeigt aber einige Parallelen auf. Wie in Tabelle 5.1 zu sehen ist, ist die Aufteilungen der einzelnen Funktionalitäten sehr ähnlich. Ein technischer Unterschied zwischen einem prozeduralen Aufruf in Java und dem Ausführen einer Regel in Henshin besteht darin, dass eine Henshin Regel keine komplexen Datentypen durch einen Rückgabewert an andere Regeln weiterreichen kann.

Ein Entwurfsmuster, das sich in der Henshin-basierten Realisierung beobachten lässt, ist die sequentielle Anreicherung des Lifting-Modells, das als Hilfsstruktur verwendet wird. Die Hilfsstruktur wird verwendet, um ähnlich wie bei prozeduralen Aufrufen in Java komplexe Datentypen durch eine Regel zurückzugeben. Dies lässt sich am Beispiel der einzelnen Generierungsmuster beobachten. Ein Muster wird hier in zwei aufeinander folgenden Schritten erzeugt. Zunächst wird die Editierregel auf das Vorkommen eines bestimmten Musters geprüft und erst im nächsten Schritt werden die daraus resultierenden Muster in der Erkennungsregel angelegt. Dieses Vorgehen findet sich genauso in der Java Implementierung wieder. Alle Correspondece-Patterns werden in Java mit dem Aufruf der Funktion `createCorrespondencePatterns()` erzeugt. Um alle Teile einer Editierre-

[2]Die Quellcodes sind frei verfügbar unter: http://pi.informatik.uni-siegen.de/Projekte/sidiff/pipeline/semantic-lifting/hot/index.htm

gel zu identifizieren, auf die das Muster angewendet werden soll, wird dann zunächst die Funktion `getLHSIntersectRHSNodes()` aufgerufen, welche alle `«preserve»` Knoten der Editierregel zurückliefert. In Henshin wird die gleiche Funktionalität durch die Regeln `Identify-Correspondence-Pattern` zum Identifizieren aller `«preserve»` Knoten und `Create-Correspondence-Pattern` zum Anlegen aller Muster ausgeführt.

Tabelle 5.1: Vergleich von HOT und Java Implementierung

Phase	**Henshin Regel**	**Java Methode / Utility**
EP:	CreateImplicitEdge	createImplicitEdges()
	MapPreservedEdge	isEdgeMapped()
IP:	MatchCorrespondencePattern	getLHSIntersectRHSNodes()
	IdentifyAddObjectPattern	getRHSMinusLHSNodes()
	⋮	⋮
	RemoveDuplicatedEReferenceTypes	-
	RemoveDuplicatedEAttributeTypes	-
CP:	CreateCorrespondencePattern	createCorrespondencePatterns()
	CreateAddObjectPattern	createAddObjectPatterns()
	⋮	⋮
MP:	MirrorNodesLHStoRHS	-
	MirrorAttributeLHStoRHS	-
	MirrorEdgesLHStoRHS	-
GP:	CreateChangeSet LinkChangesToChangeSet	createChangeSet()
	-	ModelHelper.java
	-	ModelHelperEx.java
	-	NodePair.java

Die beiden Regeln `Remove-Duplicated-EReference-Types` und `Remove-Duplicated-EAttribute-Types` haben keine entsprechenden Java Methoden. Der Unterschied in den Implementierungen besteht darin, dass die Muster, in denen die Typknoten erkannt bzw. erzeugt werden in Henshin parallel und in Java sequentiell abgearbeitet werden. Daher kann in Java die Existenz eines Typknoten direkt beim Anlegen des Musters überprüft werden (Zeile 258, 340 und 405 in `EditRule2RecognitionRule.java`). In Henshin kann dies durch die Parallelität erst nachträglich erledigt werden. Grundsätzlich wäre es aber

auch möglich, den gleichen Ablauf in Henshin über ein `If then else` Konstrukt zu realisieren.

Während in Henshin alle `«preserve»` Anteile über die Regeln `Mirror-Nodes-LHS-to-RHS`, `Mirror-Attribute-LHS-to-RHS` und `Mirror-Edges-LHS-to-RHS` abgearbeitet werden, wird die gleiche Funktionalität in Java über Hilfsfunktionen und Hilfsstrukturen geregelt. Die Utility Funktion `createPreservedNode()` der Klasse `HenshinRuleAnalysisUtilEx` und die Hilfsstruktur `NodePair` werden dem Programmierer zur Verfügung gestellt um die LHS und RHS Verwaltung von `«preserve»` Knoten transparent zu handhaben.

Dieser Vergleich zeigt, dass Entwurfsmuster und Techniken der prozeduralen Programmierung auch im Bereich der Modelltransformation hilfreich eingesetzt werden können. Übertragen lassen sich diese durch die Aufteilung der verschiedenen Funktionalität auf einzelne Regeln und durch die Verwendung von Hilfsstrukturen, um komplexe Datentypen zu handhaben. Die Technik von LHS zu RHS zu spiegeln, um `«preserve»` Knoten anzulegen, ist relativ speziell, lässt sich aber auch auf ähnliche Mustererkennungsprobleme übertragen.

6 Verwaltung der Erkennungsregeln

Eine Sammlung von Erkennungsregeln werden in einer Regelbasis zusammengefasst und verwaltet. Eine Regelbasis kann aus beliebig vielen Erkennungsregeln bestehen, gehört aber immer nur zu einem Dokumenttyp. Der Dokumenttyp gibt das Metamodell an, für welches eine Erkennungsregel bzw. Regelbasis entwickelt wurde.

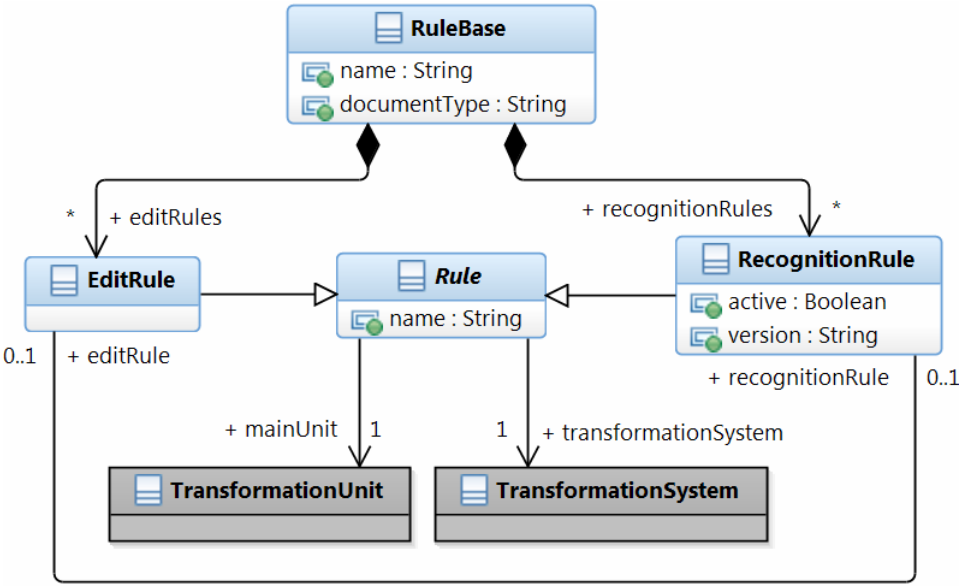


Abbildung 6.1: Regelbasis Metamodell

Die Datenverwaltung wurde in Ecore modelliert. Das hat den Vorteil, dass direkte Referenzen auf die Henshin Transformationssysteme abspeichert werden können, da diese ebenfalls auf Ecore basieren. Neben den Erkennungsregeln können auch die entsprechend dazugehörigen Editierregel abgespeichert werden. Dies hat den Vorteil, dass bei nachträglichen Änderungen an einer Editierregel die entsprechende Erkennungsregel einfach erneut generiert werden kann. Sowohl die Editierregeln als auch die Erkennungsregeln haben Namen, die später zur Anzeige und Beschreibung in der GUI dienen. Es hat sich während der Entwicklung außerdem als nützlich erwiesen, für einzelne Test Szenarien nur bestimmte Erkennungsregeln zu betrachten. Zu diesem Zweck können die Regeln

einzeln aktiviert und deaktiviert werden. Eine deaktivierte Regel wird später nicht auf die Differenz angewendet. Der Erkennungsregel Generator besitzt außerdem eine Versionsnummer die bei Änderungen am Generator vom Entwickler erhöht werden kann. Die aktuelle Versionsnummer wird dann beim Generieren in jeder Erkennungsregeln abgespeichert. Auf diese Weise können Änderungen am Algorithmus mit den Regelbasen synchronisiert werden.

Das vorgestellte Ecore-Modell dient ausschließlich der Datenverwaltung. Grundsätzlich ist diese Ebene vollständig austauschbar. Die eigentliche Schnittstelle von der Regelbasis zur Applikationslogik wird über einen s.g. Extension Point definiert.

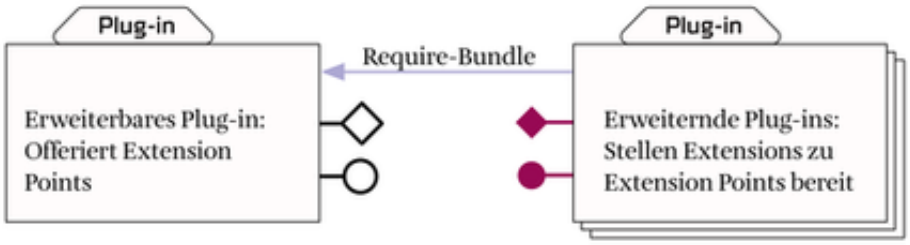


Abbildung 6.2: Extension Point [Ebe11]

Ein Extension Point ist ein Mechanismus, mit dem Plugins unter Eclipse Schnittstellen bereitstellen können, über die andere Plugins Erweiterungen (Extensions) anbieten können. Ein Plugin, welches eine Erweiterung anbieten möchte, registriert dann das Schema des Extension Points in der *Extension Registry* der Eclipse-Plattform. Von dort kann das Plugin, welches den Extension Point bereit stellt, alle Angebote nach Bedarf auslesen. [Ebe11]

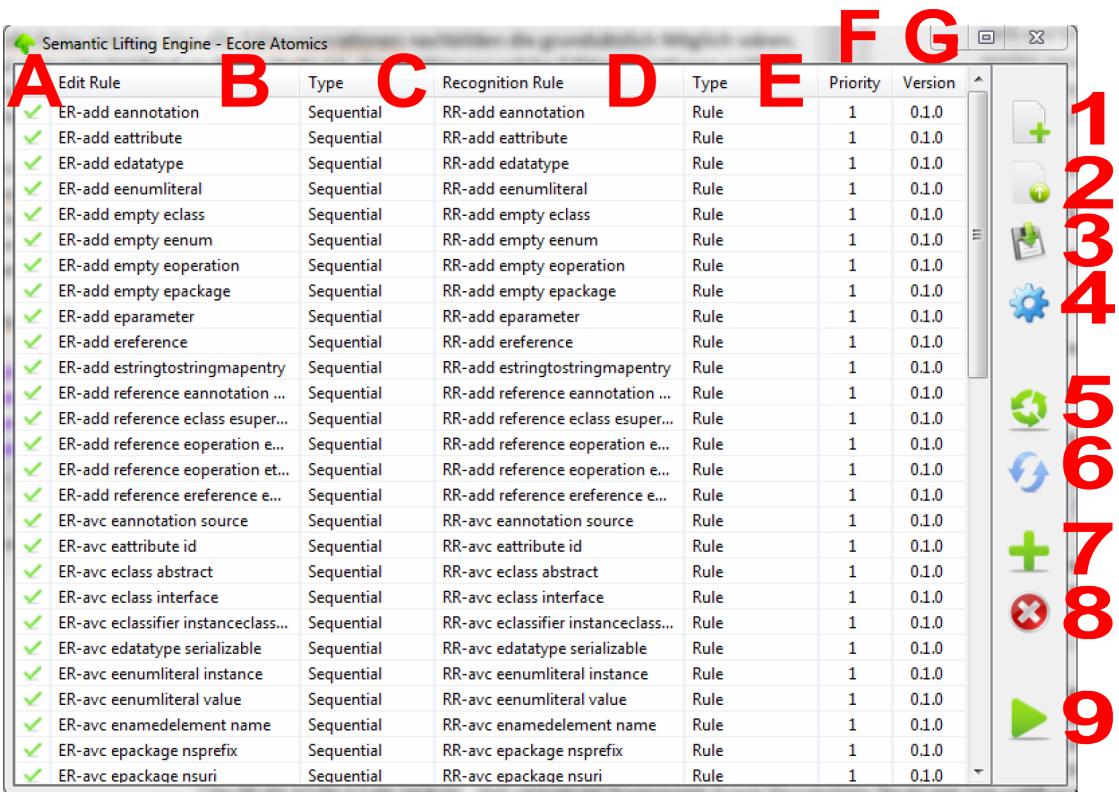
Am Extension Point werden alle benötigten Informationen an die Semantic-Lifting-Engine übergeben: Der Name und Dokumenttyp der Regelbasis sowie alle Erkennungsregeln, die auf die Differenz angewandt werden sollen. D.h. alle Regeln, die nicht aktiviert sind müssen beim Laden direkt ausgefiltert werden.

Um die Regelbasen möglichst komfortabel verwalten zu können, wird dem Benutzer eine auf diese Aufgabe ausgelegte GUI zur Verfügung gestellt. Wie in Abbildung 6.3 zu sehen, wird hier der Inhalt der Regelbasis als Liste angezeigt. Außerdem werden Funktionen geboten, um diese zu ergänzen und zu verwalten.

1: Erstellen einer neuen Regelbasis.

2: Laden einer Regelbasis im XMI Format. (*.rb.xmi)

3: Speichern einer Regelbasis im XMI Format. (*.rb.xmi)



Edit Rule	Type	Recognition Rule	Type	Priority	Version
ER-add eannotation	Sequential	RR-add eannotation	Rule	1	0.1.0
ER-add eattribute	Sequential	RR-add eattribute	Rule	1	0.1.0
ER-add edatatype	Sequential	RR-add edatatype	Rule	1	0.1.0
ER-add eenumliteral	Sequential	RR-add eenumliteral	Rule	1	0.1.0
ER-add empty eclass	Sequential	RR-add empty eclass	Rule	1	0.1.0
ER-add empty eenum	Sequential	RR-add empty eenum	Rule	1	0.1.0
ER-add empty eoperation	Sequential	RR-add empty eoperation	Rule	1	0.1.0
ER-add empty epackage	Sequential	RR-add empty epackage	Rule	1	0.1.0
ER-add eparameter	Sequential	RR-add eparameter	Rule	1	0.1.0
ER-add ereference	Sequential	RR-add ereference	Rule	1	0.1.0
ER-add estringtostringmapentry	Sequential	RR-add estringtostringmapentry	Rule	1	0.1.0
ER-add reference eannotation ...	Sequential	RR-add reference eannotation ...	Rule	1	0.1.0
ER-add reference eclass esuper...	Sequential	RR-add reference eclass esuper...	Rule	1	0.1.0
ER-add reference eoperation e...	Sequential	RR-add reference eoperation e...	Rule	1	0.1.0
ER-add reference eoperation et...	Sequential	RR-add reference eoperation e...	Rule	1	0.1.0
ER-add reference ereference e...	Sequential	RR-add reference ereference e...	Rule	1	0.1.0
ER-avc eannotation source	Sequential	RR-avc eannotation source	Rule	1	0.1.0
ER-avc eattribute id	Sequential	RR-avc eattribute id	Rule	1	0.1.0
ER-avc eclass abstract	Sequential	RR-avc eclass abstract	Rule	1	0.1.0
ER-avc eclass interface	Sequential	RR-avc eclass interface	Rule	1	0.1.0
ER-avc eclassifier instanceclass...	Sequential	RR-avc eclassifier instanceclass...	Rule	1	0.1.0
ER-avc edatatype serializable	Sequential	RR-avc edatatype serializable	Rule	1	0.1.0
ER-avc eenumliteral instance	Sequential	RR-avc eenumliteral instance	Rule	1	0.1.0
ER-avc eenumliteral value	Sequential	RR-avc eenumliteral value	Rule	1	0.1.0
ER-avc enamedelement name	Sequential	RR-avc enamedelement name	Rule	1	0.1.0
ER-avc epackage nsprefix	Sequential	RR-avc epackage nsprefix	Rule	1	0.1.0
ER-avc epackage nsuri	Sequential	RR-avc epackage nsuri	Rule	1	0.1.0

Abbildung 6.3: Regelbasis Verwaltung

4: Konfigurierung der Regelbasis.

5: Hier wird ein Dialog gestartet, mit dem aus einer bzw. mehreren Editierregeln die entsprechenden Erkennungsregeln generiert werden. Diese werden dann als neue Einträge in die Regelbasis eingefügt.

6: Generiert die in der Liste ausgewählten Erkennungsregeln erneut.

7: Startet einen Dialog, über den bereits generierte Regeln in die Regelnbasis aufgenommen werden können.

8: Entfernt die markierten Regeln aus der Regelbasis.

9: Startet den Dialog der Recognition-Engine, mit der die gesamte Differenz-Pipline gesteuert wird.

A: Aktivieren und Deaktivieren der Erkennungsregeln für die Recognition-Engine. Mit einem Klick auf den Kopf der Spalte wird die Aktivierung der Regeln invertiert.

B: Verwaltungsname der Editierregl. Kann editiert werden, wird aber nur zur Anzeige in der GUI verwendet.

C: Henshin Typ der Editierregel *mainUnit*.

D: Verwaltungsname der Erkennungsregel. Kann editiert werden, wird aber nur zur Anzeige in der GUI verwendet.

E: Henshin Typ der Erkennungsregel *mainUnit*.

F: Priorität der Erkennungsregel. (siehe 9 Post-Processing)

G: Version des verwendeten Erkennungsregel Generators.

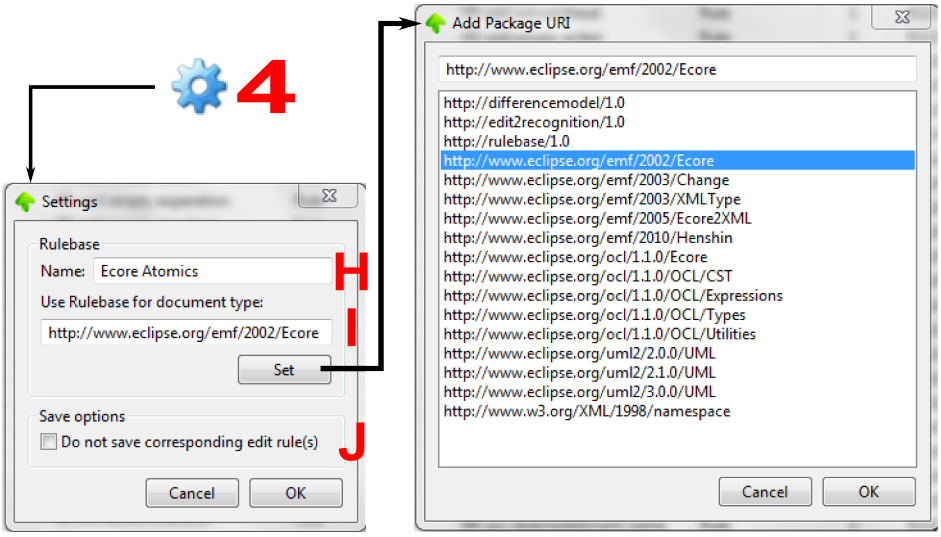


Abbildung 6.4: Regelbasis Verwaltung - Konfiguration

H: Hier kann der Name der Regelbasis angegeben werden, der später in der Recognition-Engine angezeigt wird.

I: Hier wird der Dokumenttyp der Regelbasis angegeben bzw. ausgewählt. Meistens lässt sich der Dokumenttyp aber automatisch über die Editierregeln auslesen.

J: Bei Bedarf kann hier die Verknüpfung zwischen Editier- und Erkennungsregel aufgelöst werden, sodass nur die Referenzen auf die Erkennungsregeln abgespeichert werden.

7 Laden von Resource-Sets

Grundsätzlich muss ein Modell nicht in sich geschlossen sein. Es können auch andere Modelle importiert werden und Objekte aus diesen Modellen referenziert werden. Bei Ecore basierten Modellen wird z.B. häufig die Ecore eigene Typbibliothek eingebunden. Diese Typbibliothek stellt primitive Datentypen zur Verfügung, die z.B. häufig als Attributtyp verwendet werden. Als allgemeine Typen werden aber zum Teil auch `EClass`, `EReference`, `EAttribute` usw. verwendet (siehe z.B. Differenzmodell 3.4 oder Henshin-Modell 2.3). Alle importierten Modelle müssen für ein korrektes Semantic-Lifting ebenfalls Teil der Differenz sein, da deren Modell-Elemente häufig zum Kontext der Editierregel gehören. Zum Beispiel beim Anlegen eines Attributs, wie in Abbildung 7.1 dargestellt.

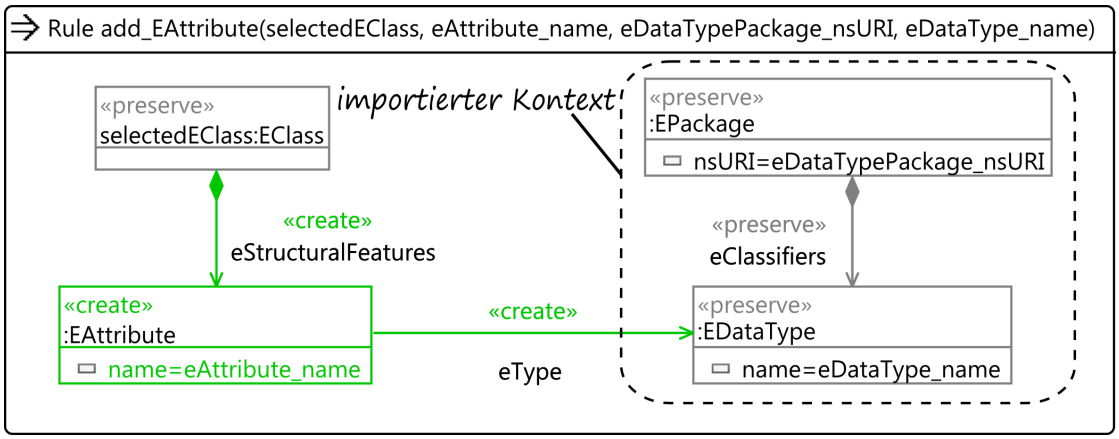


Abbildung 7.1: Add EAttribute

Aus technischer Sicht können die Modelle auf zwei Arten importiert werden:

1. Das importierte Modell, das von Modell A und B referenziert wird, ist das selbe Modell. Dies ist in der Regel dann der Fall, wenn das Modell aus der Ecore *package*

registry[3] geladen wurde. Beziehungsweise aus Sicht einer Versionsverwaltung hat sich das importierte Modell von Modell A zu Modell B nicht verändert.

2. Im Gegensatz dazu könnte es natürlich auch sein, dass sich das importierte Modell ebenfalls verändert hat. D.h. Modell A referenziert eine andere Version dieses Modells als Modell B.

Der implementierte Algorithmus befasst sich zunächst nur mit Fällen der ersten Kategorie. Hauptsächlich mit dem Ziel das Ecore-Metamodell inklusive der Typbibliothek in die Differenz einzubinden. Der folgende Algorithmus kann aber auch mit allen anderen importierten Modellen der ersten Kategorie umgehen. Hierzu wird die Differenz nach der Differenz-Ableitung um die importierten Modelle erweitert. Dabei muss bedacht werden, dass in der Erkennungsregel für jeden importierten Kontext Knoten nach einer Korrespondenz mit einem passenden Modell A und B Objekt gesucht wird. Da die Erkennungsregeln injektiv auf die Differenz abgebildet werden, heißt das auch, dass Modell A und Modell B Knoten nicht auf das selbe Objekt abgebildet werden können. Da aber bei importierten Modellen der ersten Kategorie nur eine Instanz der Modelle existiert, muss eine Kopie dieser Modelle erzeugt werden, welche dann in die Differenz eingebunden werden kann. Dazu werden folgende Schritte ausgeführt:

1. Finde alle Referenzen auf importierte Modelle in Modell A und B.

2. Erstelle eine Kopie aller referenzierten importierten Modelle. Bis hierher referenzieren die beide Modelle A und B noch die selben Instanzen der importierten Modelle. Als nächstes wird die Differenz so umgebaut, dass Modell A die originalen importierten Modelle referenziert und Modell B die Kopien.

3. Um die importierten Modelle in die Differenz einzubinden, erzeuge für jedes importierte Modell Korrespondenzen (`Correspondence`) zwischen allen originalen Modell-Element und denen der Kopie.

   Um die Differenz durch die importierten Modelle nicht unnötig groß zu machen, gibt es die Möglichkeit einen Filter zu implementieren, der festlegt, welche importierten Elemente in die Differenz übernommen werden sollen. Der im Rahmen dieses Projekts implementierte Filter übernimmt nur die referenzierten importier-

[3] siehe [SBPM09] Abschnitt 14.1.2

ten Elemente und deren Container in die Differenz. Dies sollte in der Regel für Ecore Modelle ausreichend sein.

4. Setze alle Referenzen in Modell B vom Original auf die Kopie des importierten Modells.

5. Setze alle Referenzen von Add-Reference low-level Änderungen vom Original auf die Kopie des importierten Modells.

6. Speichert alle Änderungen an der Differenz und an Modell B, damit diese nach dem eigentlichen Semantic-Lifting, zum Speichern der Differenz wieder rückgängig gemacht werden können.

Der Vorteil des beschriebenen Algorithmus besteht darin, dass kein Matching für die importierten Modelle berechnet werden muss. Da man die Korrespondenzen der importierten Modelle vom Original zur Kopie ohnehin beim Kopiervorgang gleich mit abspeichern kann.

Für importierte Modelle der zweite Kategorie müsste das Matching und die Differenz-Ableitung auf das gesamte Resource-Set beider Modelle ausgeweitet werden. Vorausgesetzt man verfügt über die entsprechenden Versionen der importierten Modelle, muss des Weiteren die Laderoutine der Modelle entsprechend angepasst werden. Es muss z.B. sichergestellt werden, dass die jeweiligen importierten Modell Versionen auch verwendet werden und nicht eine andere Version aus der Ecore *package registry* geladen wird. Theoretisch lässt sich dieser Ansatz auch auf Fälle der ersten Kategorie übertragen. Allerdings nur dann, wenn man Einfluss auf die Laderoutine hat, damit zwei Instanzen des selben importierten Modells (je für Modell A und B) geladen werden können.

8 Recognition-Engine

Die Recognition-Engine ist das eigentliche Kernstück des Semantic-Liftings. Durch das Aufrufen der Recognition-Engine werden die Erkennungsregeln auf die technische Differenz angewendet, wodurch die Semantic-Change-Sets erzeugt werden.

8.1 Aufbauen des Henshin Arbeitsgraphen

Bevor die Erkennungsregeln auf Differenz angewandt werden können muss zunächst der Henshin Arbeitsgraph aufgebaut werden. Im Arbeitsgraphen müssen alle Objekte enthalten sein, auf die die Erkennungsregeln abgebildet werden.

- Die technische Modell Differenz, welche geliftet werden soll.
- Die Modell A und B Instanzen.
- Das Metamodell von Modell A und B (in unserem Fall Ecore) für das Matching der Typknoten.
- Ecore als Meta-Metamodell für das Matching des Typs der Typknoten (`EReference`, `EAttribute`).

8.2 Filtern der Erkennungsregeln

In der Praxis hat sich gezeigt, dass die Differenzen zwischen zwei Modell Revisionen in der Regel relativ klein sind. Aufgrund dieser Annahme lassen sich an dieser Stelle bereits einige Erkennungsregeln ausfiltern, für die es keinen Match im Arbeitsgraphen geben kann. Zu diesem Zweck werden zunächst die low-level Änderungen für jeden Typ in der Differenz gezählt. Im nächsten Schritt werden dann die low-level Änderungsknoten jeder Erkennungsregel gezählt und mit der Anzahl der jeweiligen low-level Änderungen in der Differenz verglichen. Ist die Anzahl der Änderungen in der Erkennungsregel größer als in der Differenz, werden diese Regeln ausgefiltert. Enthält eine Differenz z.B. keine

Attribute-Value-Changes, dann werden alle Erkennungsregeln ausgefiltert, die ein oder mehr Attribute-Value-Change Knoten enthalten.

8.3 Optimierung der Erkennungsregeln

Die technische Abbildung einer LHS auf den Arbeitsgraphen wird in Henshin durch die Reihenfolge der Knoten in der Liste des LHS-Graphen beeinflusst. Henshin sucht in der Reihenfolge, in der die Knoten in der Liste liegen, nach passenden Objekten, auf die der Knoten abgebildet werden kann. Wurden die ersten Knoten in der Liste bereits auf Objekte abgebildet, schränkt dies die Auswahl der möglichen Objekte für weitere folgende Knoten ein. Es empfiehlt sich also, Knoten in der Liste nach vorne zu ziehen, für die es einzeln betrachtet nur wenige Abbildungsmöglichkeiten gibt.

Wenn man davon ausgeht, dass ein Modell in der Praxis in den meisten Fällen zunehmend anwächst und die Differenzen zwischen den einzelnen Revisionen relativ klein bleiben (also wenig Änderungen), dann lassen sich für die Typknoten und Änderungsknoten am wenigsten Abbildungsmöglichkeiten finden. Für die Correspondence-Knoten und Modell Knoten gibt es hingegen die meisten Möglichkeiten, diese auf den Arbeitsgraphen abzubilden. Aufgrund dieser Überlegungen ergibt sich die folgende Reihenfolge der Knoten im LHS-Graphen:

1. **Differenzknoten:** Die Differenz als Wurzelknoten ist innerhalb des Arbeitsgraphen eindeutig, sie bringt allerdings keine wirkliche Einschränkung der Abbildungsmöglichkeiten für den Rest des Transformationsgraphen.

2. **Typknoten:** Da der Typ einer Referenz oder eines Attributs über den Namen auf das Metamodell abgebildet wird, sind die Typknoten einzeln betrachtet nicht zwangsläufig eindeutig. Man kann aber davon ausgehen, dass innerhalb eines Modells die Anzahl der Referenzen und Attribute mit dem gleichen Namen in der Praxis eher gering ist.

3. **Änderungsknoten:** Die Änderungsknoten werden zusätzlich anhand der Anzahl der Änderungen in der Differenz sortiert, wobei Änderungen mit geringerer Anzahl nach vorne gezogen und häufig vorkommende Änderungen in der Liste nach hinten sortiert werden.

4. **Modell A und B Knoten:** An dieser Stelle werden die einzelnen low-level Änderungen über die Modell A bzw. B Objekte miteinander verbunden bzw. einander

korrekt zugeordnet. Dies ist besonders dann entscheidend, wenn es mehrere low-level Änderungen gibt, deren Referenzen auf Objekte des gleichen Typs zeigen. Dies ist z.B. der Fall, wenn eine Editieroperation mehrfach auf verschiedene Objekte angewendet wurde.

5. **Correspondence-Knoten:** Durch die Correspondences wird zum Schluss der Bezug zwischen Modell A und Modell B hergestellt. Beispielsweise beim Einfügen einer `EReference` in ein Modell dient die `EClass` als Container Objekt. Für dieses Container Objekt muss nun eine passende Correspondence gefunden werden. Wurden für die Abbildung der Erkennungsregel bereits die Typ-, Änderungs- und Modell B Knoten gefunden, dann lässt sich die Correspondence (falls vorhanden) in diesem Fall sofort eindeutig festlegen.

   Umgekehrt wäre es sehr aufwändig (bei einem großen Modell) nacheinander alle Correspondences mit `EClasses` zu betrachten und dann für jede Correspondence zu überprüfen ob diese durch eine Add-Reference mit einem Add-Object verbunden ist. Daher sollten die Correspondeces erst zum Schluss des Matchings auf den Arbeitsgraphen abgebildet werde.

Auf diese Weise hängt die Berechnungszeit von der Anzahl der Änderungen im Modell ab. Ansonsten würde die Berechnungszeit mit zunehmender Größe der Modelle immer länger werden, was zu einer sehr schlechten Skalierbarkeit des Algorithmus im Bezug auf wachsende Modell Revisionen führen würde.

Da man nicht mit Sicherheit davon ausgehen kann, dass die Reihenfolge der Knoten zwischen Generieren, Serialisieren und Laden gleich geblieben ist, werden die Knoten erst direkt vor dem Matching sortiert.

8.4 Ausführen der Erkennungsregeln

Das Ausführen der Erkennungsregeln lässt sich in drei Phasen unterteilen:

1. **Parallelisierungsphase:** Die erste Phase ist eine rein technische Optimierung. Sie basiert darauf, dass die Erkennungsregeln unabhängig voneinander auf die Differenz abgebildet und angewendet werden können. Um dies auszunutzen, soll die Matching- und Anwendungsphase parallelisiert werden. Dazu werden die Regeln immer in Blöcken an einzelne Threads übergeben. Die Block-Größe, also die Anzahl der Regeln pro Thread, kann in der Recognition-Engine konfiguriert werden. Bei Mehr-Kern-Systemen lies sich dadurch eine deutlich bessere Auslastung der

CPU erreichen, da sich die Last durch die Aufteilung effektiver auf die einzelnen CPU-Kerne verteilen lässt.

2. **Matchingphase:** In dieser Phase wird die linke Seite jeder Regel (LHS) auf den Arbeitsgraphen abgebildet. Wodurch die s.g. *Matches* erzeugt werden. Ein Match gibt genau den Teil des Arbeitsgraphen an, der in der Anwendungsphase durch die rechte Seite der Regel (RHS) ersetzt wird. Ein Match für eine Regel muss nicht eindeutig sein. D.h. für jede Regel können mehrere und zum Teil auch sich überschneidende Matches existieren. Im Idealfall entspricht die Anzahl der Matches für jede Erkennungsregel der Häufigkeit, mit der die Editierregel auf das Modell angewendet wurde. Welche Probleme sich durch Überschneidungen von Matches ergeben und wie diese behandelt werden, wird im nächsten Abschnitt 9 Post-Processing beschprochen.

3. **Anwendungsphase:** Wie bereits erwähnt werden in der Anwendungsphase alle Matches durch die entsprechende rechte Seite der Erkennungsregel ersetzt. Der einzige Teil, der nicht im Schnitt von LHS und RHS enthalten ist, ist das Semantic-Change-Set und die Kanten zu den jeweiligen low-level Änderungen. Das ist eben genau der Teil, welcher der Differenz hinzugefügt wird. D.h. nach diesem Schritt sind die Semantic-Change-Sets mit den dazu gehörigen low-level Änderungen in der Differenz enthalten. Da sich allerdings ggf. die Matches überschneiden können, überschneiden sich jetzt auch die entsprechenden Semantic-Change-Sets. D.h. eine low-level Änderung kommt dann in mehreren Semantic-Change-Sets vor.

9 Post-Processing

Nachdem möglichst alle low-level Änderungen zu Semantic-Change-Sets (SCS) gruppiert wurden, muss die Differenz in einem weiteren Schritt optimiert werden. Bis hierhin ist es möglich, dass sich die SCS in Teilen überschneiden oder sogar identisch sind. Im Folgenden wird jedes SCS als Menge (*engl. Set*) betrachtet. Eine low-level Änderung ist ein Element, aus der sich die Mengen zusammensetzten. Ein Element kann in mehreren Mengen vorkommen, aber nur einmal in einer Menge. Zwei SCS überschneiden sich also genau dann, wenn mindestens einmal die selbe low-level Änderung in beiden SCS vorkommt. Zwei SCS sind identisch, wenn beide die selben low-level Änderungen enthalten. Zu Beginn müssen die Kriterien festgelegt werden, nach denen die geliftete Differenz optimiert werden soll:

1. Eliminieren aller Überschneidungen von SCS. D.h. nach dem Post-Processing soll jede low-level Änderung nur in genau einem SCS vorkommen.
2. Erreichen einer möglichst guten Abdeckung der low-level Änderungen. D.h. beim Entfernen von sich überschneidenden SCS sollen möglichst wenige bis keine low-level Änderungen entstehen, die in keinem SCS enthalten sind.
3. **Kompression:** Es sollten möglichst die SCS mit großem Betrag ($|SCS|$) erhalten bleiben. D.h. ein SCS mit vielen low-level Änderungen wird gegenüber einem SCS mit wenigen Änderungen bevorzugt.

Der im Folgenden vorgestellte Algorithmus ist in 5 Schritte aufgeteilt. Das Vorgehen basiert auf dem in [KKT11] (S.8) beschriebenen Konzept. In jedem Schritt wird die Differenz nach den oben beschriebenen Kriterien weiter optimiert. Um den Algorithmus zu notieren werden zwei neue Mengen für SCS eingeführt.

- PCS_{min}: In dieser Menge werden alle überschneidungsfreien SCS abgelegt, die später in der Differenz erhalten bleiben sollen.
- PCS_D: Diese Menge enthält die noch zu bearbeitenden SCS. PCS_D enthält zu Beginn alle potenziellen SCS aus der gelifteten Differenz. Während jedem Schritt des

Algorithmus wird PCS_D nach den oben beschriebenen Kriterien optimiert. D.h. alle SCS, die Überschneidungen aufweisen, werden nach und nach entfernt. Alle SCS, die überschneidungsfrei sind, werden dann aus PCS_D in PCS_{min} verschoben. Am Ende des Algorithmus muss PCS_D leer sein.

9.1 Identische Semantic Change Sets

Zu Beginn wird PCD_D mit allen potenziellen SCS aus der Differenz gefüllt. Identische SCS können relativ leicht identifiziert werden und werden deshalb bereits hier ausgefiltert. Um zu entscheiden welche SCS behalten werden sollen und welche verworfen werden, wurden zwei Qualitätskriterien für SCS angelegt.

- **Priority:** Die Priorität kann für jede Regel konfiguriert werden. Die Priorität wird als Integerwert angegeben. Eine hohe Priorität wird durch einen großen Wert angegeben, eine niedrige Priorität durch einen kleinen Wert. Es sind auch negative Werte möglich. Als Standard wird bei der Generierung der Wert 1 für normale Erkennungsregeln und 0 für Erkennungsregeln mit einer Amalgamation-Unit gesetzt. Am häufigsten tritt das Problem von identischen SCS dann auf, wenn nur die Kernregel einer Amalgamation-Unit Erkennungsregel ausgeführt wird und die Kernregel genau einer normalen Erkennungsregel entspricht. Zum Beispiel hat die Advanced-Regel Pull-Up-EAttribute als Kernregel die Atomic-Regel Move-EAttribute. In der Regel möchte man aber in einem Fall, wo nur ein einziges Attribut verschoben wurde, eher die Move Regel als die Pull-Up Regel als Lifting angezeigt bekommen.

- **Refinment-Level:** Wenn anhand der Priorität keine Entscheidung über die identischen SCS getroffen werden kann, weil beide Prioritäten ebenfalls gleich sind, wird das Refinment-Level betrachtet. Der Wert des Refinment-Level leitet sich aus der Editierregel ab und gibt an wie speziell bzw. abstrakt eine Editierregel ist. Dazu wird für jeden (`«delete»`, `«create» und «preserve»`) Knoten der Edierregel die Anzahl der Supertypen ermittelt und aufsummiert. Dieser Wert wird direkt bei der Generierung der Erkennungsregel ermittelt und beim Ausführen der Erkennungsregel in das erzeugte SCS geschrieben. Im Sinne des Semantic-Liftings soll die möglichst spezielle Editieroperation angezeigt werden. Daher werden SCS mit einem hohen Refinment-Level bevorzugt in PCD_D übernommen. Zum Beispiel die Editieroperation die das Attribut `name` der abstrakten Ecore Klasse `ENamedElement` setzt, hat ein Refinment-Level von 1, da der einzige Supertyp `EModelElement` ist.

Im Gegensatz dazu hat die Editieroperation, die den in diesem Fall vererbten Namen einer `EClass` setzt, ein Refinment-Level von 3. Die Supertypen sind hier: `EClassifier`, `ENamedElement` und `EModelElement`.

Ist auch das Refinment-Level der SCS gleich, so wird das erste in Differenz vorkommende SCS in PCD_D übernommen. Das Resultat dürfte allerdings relativ zufällig sein, da die SCS parallel berechnet wurden. Außerdem wurden in verschiedenen Berechnungsschritten zuvor Sets verwendet, die auf Hashfunktionen basieren. Daher ist die Reihenfolge der SCS in der Differenz nicht abzusehen.

9.2 Überschneidungsfreie Semantic Change Sets

Alle SCS die keine Überschneidungen mit anderen SCS haben, können sofort aus PCS_D in PCS_{min} verschoben werden. In der Praxis dürften in der Regel die meisten SCS überschneidungsfrei sein. Nach Definition im Abschnitt 4 sind auf jeden Fall alle Atomic-Regeln untereinander überschneidungsfrei. Es ist aber möglich, dass sich die Atomic-Regeln mit Advanced-Regeln zum Teil oder vollständig überschneiden. Die dadurch ggf. entstehenden SCS verbleiben zunächst in PCS_D.

9.3 Verschachtelte Semantic Change Sets

In diesem Schritt werden SCS gesucht, bei denen alle überschneidenden SCS Teilmengen dieses SCS sind. Die jeweiligen Teilmengen können dann aus PCS_D entfernt und in PCS_{min} übernommen werden.

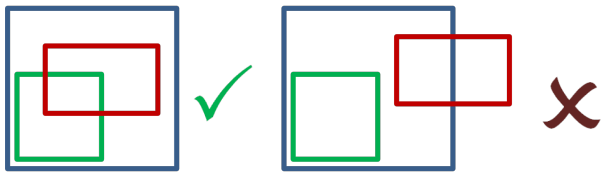


Abbildung 9.1: Verschachtelte Semantic Change Sets

Diese Situation lässt sich auch ganz gut aus Sicht der Regeln betrachten. Ist das Metamodell vollständig durch Atomic-Regeln abgedeckt, dann sollten sich alle Advanced-Regeln in Atomic-Regeln zerlegen lassen. Für die verbliebenen SCS heißt das, dass bei genau solchen Fällen alle SCS, die durch Atomic-Regeln entstanden sind, verworfen werden und nur die SCS erhalten bleiben, die durch die überdeckenden Advanced-Regeln

entstanden sind. Problematisch sind dann nur noch sich überschneidende SCS, die durch Advanced-Regeln erzeugt wurden. Solche Zusammensetzungen können an dieser Stelle noch nicht gelöst werden. Damit verbleiben alle SCS in der Differenz, die sich in Teilbereichen überschneiden, aber auch alle SCS, die in solche SCS verschachtelt sind.

9.4 Semantic Change Sets mit exklusiven low-level Änderungen

Nach der Überschneidungsfreiheit der SCS ist die Forderung nach einer möglichst guten Abdeckung aller low-level Änderungen das nächst wichtigere Optimierungskriterium. Ist in PCS_D zu diesem Zeitpunkt ein SCS X vorhanden, das low-level Änderungen enthält, die in keinem anderen SCS enthalten sind, dann müsste SCS X grundsätzlich in PCS_{min} übernommen werden, um eine vollständige Abdeckung aller low-level Änderungen zu erreichen. Dies gilt allerdings nur dann, wenn alle mit SCS X überlappenden SCS aus PCS_D entfernt werden können, ohne ungruppierte low-level Änderungen zu erzeugen. Ansonsten kann diese Operation für SCS X nicht ausgeführt werden, da hier noch nicht entschieden werden kann, ob es nicht evtl. eine bessere Kombination von SCS gibt, durch die weniger ungruppierte low-level Änderungen entstehen.

Vorausgesetzt alle Korrespondenzen wurden korrekt ermittelt (Abschintt 3.1 Matching), ist ein solcher Fall wie hier beschrieben ein Indiz dafür, dass die Atomic-Regelbasis nicht vollständig ist, da sonst beim Entfernen eines überlappenden SCS die restlichen low-level Änderungen immer maximal in SCS zerfallen, die von Atomic-Regeln stammen.

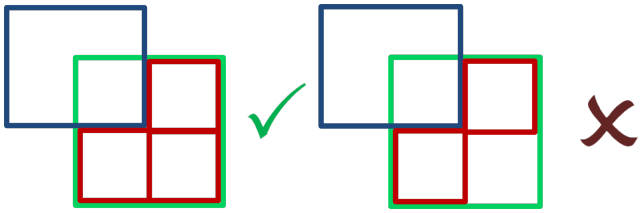


Abbildung 9.2: Semantic Change Sets mit exklusiven low-level Änderungen

9.5 Minimierung der Semantic Change Set Menge

Alle SCS, die jetzt noch in PCS_D enthalten sind, überschneiden sich so, dass sich nicht sofort eindeutig feststellen lässt, welche Kombination zu einem optimalen Ergebnis führt. Daher werden alle möglichen Kombinationen betrachtet und die beste Kombination ausgewählt. Da die Laufzeit und Skalierbarkeit eines solchen kombinatorischen Optimie-

rungsproblems stark von der Anzahl der sich überschneidenden SCS abhängt, werden zunächst alle SCS in PCS_D in disjunkte Mengen zerlegt. D.h es werden immer nur SCS betrachtet welche durch Überschneidungen eine zusammenhängende Teilmenge von PCS_D bilden. Eine solche Teilmenge (*engl. subset*) aus PCS_D wird im Folgenden allgemein als PCS_{DS} bezeichnet. Wobei $PCS_{DS} \subseteq PCS_D$ gilt. In der Praxis kann man davon ausgehen, dass die PCS_{DS} Teilmengen relativ klein sind.

> „Finally, the remaining change sets are partially overlapping. Our practical evaluation has shown that these cases are very rare.“ [KKT11] (S.8)

In der Regel dürfte es daher möglich sein, alle Kombinationen zu berechnen und jede Kombination nach den genannten Optimierungskriterien zu bewerten. Da keine Abhängigkeit zwischen den einzelnen PCS_{DS} Teilmengen besteht, können alle Teilmengen parallel berechnet werden.

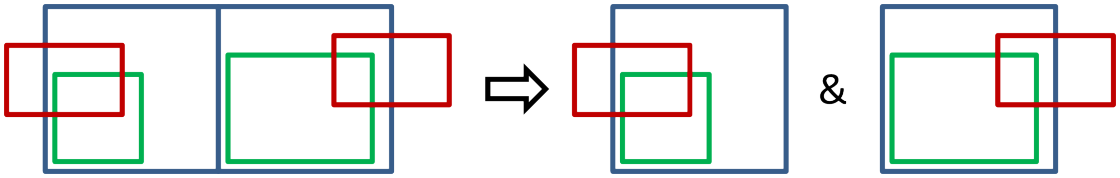


Abbildung 9.3: Zerlegung der Semantic-Change-Sets in disjunkte Mengen

Sei PCS_{DS1} eine solche Teilmenge aus PCS_D. Beim Auswählen der Kombinationen aus PCS_{DS1} spielt die Reihenfolge der SCS innerhalb einer Kombination keine Rolle. Es sollen alle möglichen Kombinationen ausgewählt werden. Dies entspricht genau der Potenzmenge von PCS_{DS1}. Die Potenzmenge enthält alle möglichen Teilmengen von PCS_{DS1}. Bei n SCS in PCS_{DS1} sind dies genau 2^n Teilmengen bzw. Kombinationen, wobei die leere Menge und die Kombination die gleich PCS_{DS1} ist, vernachlässigt werden können. Damit lässt sich die Komplexität dieses Algorithmus Schritts bei n SCS pro PCS_{DS} mit $\mathcal{O}(2^n)$ angeben. Das Limit der maximal zu berechnenden Kombinationen von SCS pro PCS_{DS} kann für das Post-Processing Modul konfiguriert werden. Ist $(2^n) > Limit$, dann wird die Berechnung der Kombinationen bei Erreichen des Limits abgebrochen.

Die Potenzmenge von PCS_{DS} lässt sich relativ leicht über Binärzahlen berechnen. Dazu berechnet man alle Binärzahlen zwischen 0 und 2^n. Durch jede dieser Binärzahlen wird jetzt eine Teilmenge für die Potenzmenge gebildet, wobei jede Stelle einer Binärzahl mit dem SCS an der entsprechenden Stelle in PCS_{DS} assoziiert wird. Bei einer 1 wird das SCS in die Teilmenge übernommen und bei einer 0 nicht.

Zum Beispiel: $PCS_{DS1} = \{SCS1, SCS2, SCS3\}, 3_d \rightarrow 011_b \Rightarrow \{SCS2, SCS3\}$

Jede Kombination, die für eine PCS_{DS} Teilmenge berechnet wird muss anschließend bewertet werden. Alle Kombinationen, die überlappende SCS enthalten, können dabei übersprungen werden. Für das wichtigste Kriterium wird zunächst die Anzahl der low-level Änderungen bestimmt, welche in den SCS von PCS_{DS}, aber nicht in den SCS der Kombination enthalten sind. Dies sind genau die low-level Änderungen, die später ungruppiert wären, wenn diese Kombination als beste Lösung für das Problem ausgewählt würde. Es werden immer die Kombinationen bevorzugt, welche am wenigsten ungruppierte low-level Änderungen produzieren. Das nächst wichtigere Kriterium ist eine möglichst große Kompression der Differenz zu erhalten. Die Kompression ergibt sich in diesem Fall aus der Anzahl der SCS in der Kombination zur Anzahl der SCS in PCS_{DS}. Für den Vergleich der Kombinationen heißt dies, dass immer die Kombination mit den wenigsten SCS bevorzugt wird. Wobei zu beachten ist, dass die Kompression erst dann verglichen wird, wenn die Anzahl der ungruppierten low-level Änderungen der Kombinationen gleich ist. Die beste Kombination, die jeweils für eine PCS_{DS} Teilmenge gefunden wurde, wird dann in PCS_{min} übernommen und alle PCS_{DS} werden aus PCS_D entfernt.

9.6 Ergebnis

Damit ist PCS_D leer. Alle SCS, die nun in PCS_{min} sind, bleiben in der Differenz erhalten und alle anderen werden entfernt. ($Differenz = Differenz \bigcap PCS_{min}$) Im Idealfall wurden durch das Post-Processing keine ungruppierten SCS erzeugt. Entstehen ungruppierte SCS, kann das ein Indiz dafür sein, dass die Atomic-Regelbasis nicht vollständig ist oder dass nicht korrekte Korrespondenzen gebildet wurden. Auf jeden Fall muss die Differenz nach dem Post-Processing überschneidungsfrei sein.

10 Sequential-Recognition-Engine

Die bis hierher beschriebene Recognition-Engine erkennt Editieroperationen nur dann, wenn sie nicht in sequentieller Abhängigkeit zueinander stehen. Sind zwei Editierregeln X und Y sequentiell abhängig, dann bedeutet das, Editierregel Y kann erst angewandt werden, wenn Editierregel X bereits ausgeführt wurde. In diesem Fall würde also nur die Editieroperationen der Editierregel X durch die entsprechende Erkennungsregel in der Differenz geliftet. Es können auch mehrere Editierregeln in sequentieller Abhängigkeit stehen. Grundsätzlich lassen sich zwei Arten von Abhängigkeiten unterscheiden.

- **Create-Use-Dependence:** Eine solche Abhängigkeit tritt immer dann auf, wenn ein neu hinzugefügtes Objekt wieder als Kontext für eine weitere Editieroperation benötigt wird. Der Kontext ist im Fall einer Henshin Editierregel immer der «`preserve`» Teil der Regel. Am häufigsten wird der Container eines hinzuzufügenden oder zu entfernenden Objekts als Kontext benötigt.

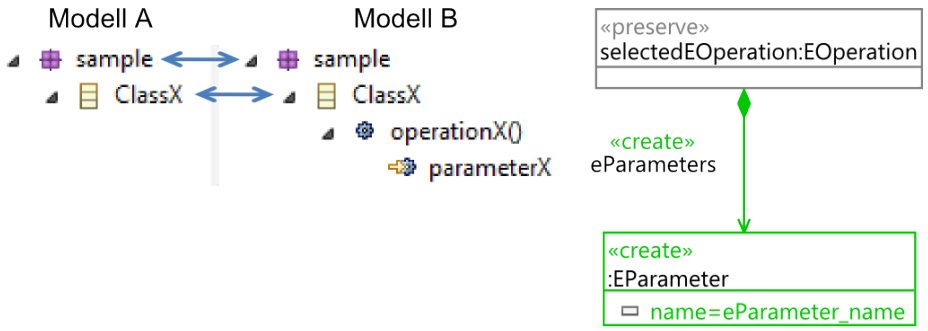


Abbildung 10.1: Create-Use-Dependence Beispiel

Abbildung 10.1 beschreibt einen solchen Fall. Hier existiert eine Korrespondenz zwischen dem Paket `sample` und der Klasse `ClassX`. Die Operation `operationX()` und der Parameter `parameterX` wurden dem Modell hinzugefügt. Das Einfügen der Operation kann zunächst problemlos durch die entsprechende Erkennungsregel geliftet werden. Wie in der Abbildung 10.1 der Editierregel zum einfügen eines Parameters zu erkennen ist, wird hier eine bereits bestehende `EOperation` als Kon-

text erwartet. Innerhalb der Erkennungsregel ergibt sich nun das Problem, dass für die `EOperation` nach einem Correspondence-Pattern gesucht wird. Es existiert aber nur ein Add-Object-Pattern für `operationX()`. Daher wird das einfügen des Parameters `parameterX` nicht geliftet.

- **Removed-Used-Dependence:** Das Problem dieser Abhängigkeit verhält sich im Prinzip analog zum Create-Use-Dependence Problem. In diesem Fall wurde der Kontext, auf den sich die erste Editieroperation bezog, im nächsten Schritt durch eine weitere Editieroperation entfernt. Damit ergibt sich beim Lifting grundsätzlich das gleiche Problem.

  Wie Abbildung 10.2 zeigt, fehlt wieder die Korrespondenz für `operationX()`, um das Entfernen des Parameters `parameterX` zu liften, da für `operationX()` nur ein Remove-Object-Pattern in der Differenz existiert.

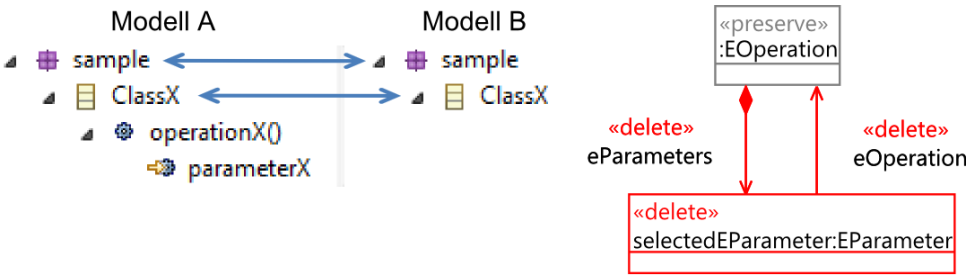


Abbildung 10.2: Removed-Used-Dependence Beispiel

Eine Möglichkeit dieses Problem zu betrachten ist es davon auszugehen, dass für ein vollständiges Lifting immer alle Zwischenzustände des Modells benötigt werden, bevor eine Create-Use- oder Removed-Used-Dependence entsteht. Die vollständig geliftete Differenz wäre dann die Summe aller gelifteten Zwischenzustände. Genau dieses Prinzip wird von der Sequential-Recognition-Engine verfolgt. Die Hauptaufgabe besteht darin die erwähnten Zwischenzustände zu berechnen. Das Lifting zwischen zwei Modell Zuständen wird wie bisher von der Recognition-Engine mit nachfolgendem Post-Processing übernommen.

Die Zwischenzustände errechnen sich immer aus dem gelifteten Teil der Differenz. Startet die Sequential-Recognition-Engine also mit einer ungelifteten Differenz, wird zunächst die normale Recognition-Engine aufgerufen, um danach alle Semantic-Change-Sets durch das Post-Processing überschneidungsfrei zu machen. Die Überschneidungsfreiheit ist insofern wichtig, damit die folgenden Schritte nicht mehrfach ausgeführt wer-

den. Als nächstes werden alle low-level Änderungen, welche durch Semantic-Change-Sets gruppiert sind, auf die Modelle angewendet um die Zwischenzustände zu erzeugen:

1. Als erstes werden low-level Änderungen vom Typ Add-Object behandelt. Ein Add-Object referenziert immer ein Objekt (`objB`) aus Modell B. Von diesem Objekt wird nun eine Kopie erstellt (`objA`). Als nächstes wird die entsprechende Correspondence zwischen `objA` und `objB` angelegt und der Differenz hinzugefügt. Es ist zu beachten, dass `objA` zu diesem Zeitpunkt nur durch die Correspondence referenziert wird, es wird hier noch kein Container Objekt festgelegt. Die Container Referenzen werden wie normale Referenzen erst nach Anlegen aller Objekte eingefügt. Das hat den Vorteil, dass keine Abhängigkeiten zwischen Objekten beachtet werden müssen, die innerhalb eines Semantic-Change-Sets auftreten können.

2. Die Add-References werden jetzt ebenfalls in Modell A eingefügt. Dazu muss zunächst Quelle und Ziel der einzufügenden Referenz aus Modell B auf Modell A abgebildet werden. Dazu werden die Correspondences zwischen den Quell- und Zielobjekten der beiden Modelle verwendet. Als nächstes wird dann die Referenz des Modell A Quellobjekts auf das entsprechende Modell A Zielobjekt gesetzt.

3. Die referenzierten Objekte der Remove-Object low-level Änderungen werden analog zu den der Add-Objekts kopiert und als Modell B Objekt in einer neuen Correspondence abgelegt.

4. Alle Remove-References werden ebenfalls analog zu den Add-References von Modell A auf Modell B übertragen.

5. Für alle Attribute-Value-Changes werden die Werte der Attribute in Modell A auf die neuen Werte aus Modell B gesetzt. Dies reicht in der Regel aus, um Attribute zwischen Modell A und Modell B in der Erkennungsregel auf Gleichheit zu prüfen. (siehe 5.11 Preserved-Attribute-Value-Pattern)

Nachdem alle low-level Änderungen aus allen Semantic-Change-Sets mit den Modellen verschmolzen wurden, können sowohl die low-level Änderungen als auch die Semantic-Change-Sets temporär aus der Differenz entfernt werden. Der so errechnete Zwischenzustand dient nun als Ausgangspunkt für das nächste Lifting und Post-Processing. Dieser Vorgang wird solange wiederholt, wie in einem Durchlauf noch neue Semantic-Change-Sets erzeugt werden oder bis keine low-level Änderungen mehr in der Differenz vorhanden sind.

Durch die in Abschnitt 8.2 beschriebene Filterung der Regeln reduziert sich die Anzahl der Erkennungsregeln von Schritt zu Schritt, da immer weniger low-level Änderungen in der Differenz zurück bleiben. Würde man die Editierregeln auf ihre Abhängigkeit analysieren, dann könnte man die anzuwendenden Erkennungsregeln noch gezielter eingrenzen. Zunächst würde ein ganz normaler Erkennungsdurchlauf erfolgen. In den nächsten Schritten würden dann immer nur die Erkennungsregeln angewendet, welche mindestens eine sequentielle Abhängigkeit zu den Editieroperationen haben, die im Schritt zuvor erkannt wurden.

Am Ende werden alle Semantic-Change-Sets, inklusive deren low-level Änderungen, die in den einzelnen Lifting-Schritten entfernt wurden, wieder in der Differenz zusammengeführt. Außerdem müssen alle eingefügten Correspondences wieder entfernt werden, um die Differenz auf ihren Ausgangspunkt zurückzusetzen.

11 Benutzerschnittstelle der Semanitc-Lifiting-Engine

Die Semantic-Lifting Funktionalitäten können in Eclipse über ein dafür angelegtes *Action Set* ausgewählt werden (Abbildung 11.1 (links)). Durch ein *Action Set* lässt sich die Eclipse Plattform um Menüs und Werkzeugleisten-Einträge erweitern.

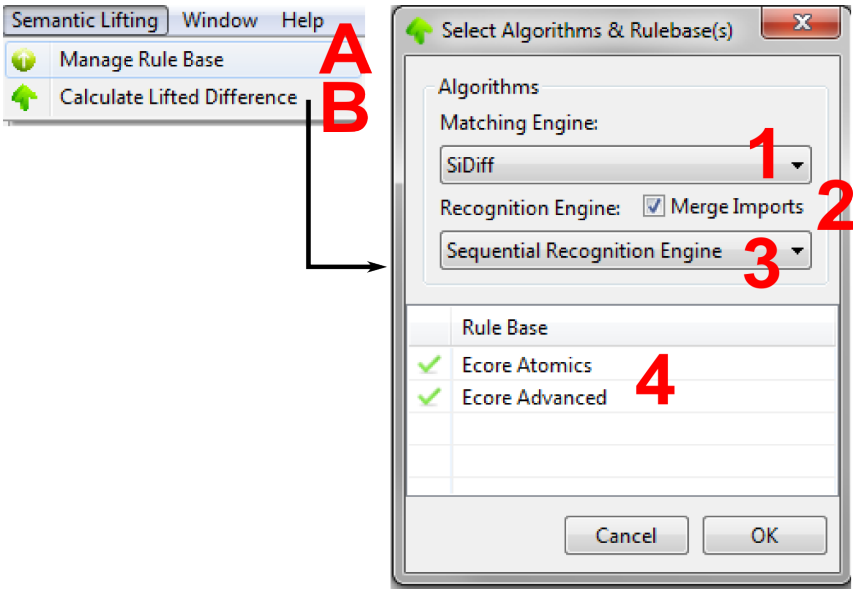


Abbildung 11.1: Semantic-Lifting Dialog

A: Startet die in Abschnitt 6 beschriebene Regelbasis-Verwaltung.

B: Startet einen Dialog, mit dem die Semantic-Lifting-Engine gesteuert werden kann. Zunächst muss der Benutzer entscheiden, ob er eine neue technische Differenz berechnen will oder ob eine bereits bestehende technische Differenz geladen und geliftet werden soll. Als nächstes wird dann ein Dateidialog gestartet, in dem entsprechend der zuvor getroffenen Auswahl entweder eine ungeliftete technische Differenz

oder zwei zu vergleichende Modelle (Modell A und B) auszuwählen sind. Als nächstes wird der in Abbildung 11.1 (rechts) gezeigte Dialog aufgerufen.

1: Hier können die in Abschnitt 3.1 beschriebenen Vergleichsalgorithmen ausgewählt werden, durch die die Korrespondenzen zwischen Modell A und B berechnet werden. Zur Auswahl stehen *SiDiff* (Abschnitt 3.1.1) und der *Named-Element-Matcher* (Abschnitt 3.1.2). Diese Option kann nur gewählt werden, wenn zu Beginn zwei neu zu vergleichende Modelle ausgewählt wurden.

2: Durch die Option *Merge Imports* kann der Algorithmus zur Behandlung von importierten Modellen in Resource-Sets an- und abgeschaltet werden (Abschnitt 7).

3: An dieser Stelle werden die Algorithmen für das Semantic-Lifting ausgewählt. Es stehen folgende Optionen zur Verfügung.

- **No Semantic Lifting:** Es wird kein Semantic-Lifting durchgeführt. Dadurch wird eine ungeliftete technische Differenz erstellt, wenn zu Beginn zwei Modelle zum Vergleich ausgewählt wurden.
- **Simple Recognition Engine:** Die technische Differenz wird nur semantisch geliftet. Es wird allerdings kein Post-Processing durchgeführt, daher können überlappende Semantic-Change-Sets in der Differenz auftreten.
- **Post Processed Recognition Engine:** Es wird eine semantisch geliftete und durch das Post-Processing überschneidungsfreie Differenz berechnet.
- **Sequential Recognition Engine:** Es wird eine semantisch geliftete Differenz mit dem unter Abschnitt 10 beschriebenen Algorithmus berechnet. Hierdurch werden auch die Editieroperationen erkannt, die in sequentieller Abhängigkeit zueinander stehen.

4: Hier werden alle Regelbasen angezeigt, deren Dokumenttyp dem Metamodell von Modell A und Modell B entsprechen. Die einzelnen Regelbasen können für das Semantic-Lifting an- und abgewählt werden. Die Regelbasen werden, wie in Abschnitt 6 beschrieben, als *Extension Point* über die *Extension Registry* der Eclipse Plattform ausgelesen.

Nach diesem Dialog wird die Differenzberechnung mit den ausgewählten Optionen gestartet. Nachdem die Differenz gespeichert wurde, kann diese in dem durch Ecore generierten Editor angezeigt werden. Der Editor wurde so angepasst, dass alle low-level Änderungen, die geliftet wurden, auch als Gruppierung unterhalb der Semantic-Change-Sets angezeigt

werden. Alle nicht gruppierte low-level Änderungen werden mit der Differenz als Container angezeigt, wie in Abbildung 11.2 zu sehen ist. Hier wurden zwei Modell Versionen aus dem EMF Projekt (org.eclipse.emf.codegen.ecore) verglichen, wobei hier keine sequentiell abhängigen Editieroperationen geliftet wurden, daher sind ein paar low-level Änderungen ungruppiert, die sonst gruppiert wären.

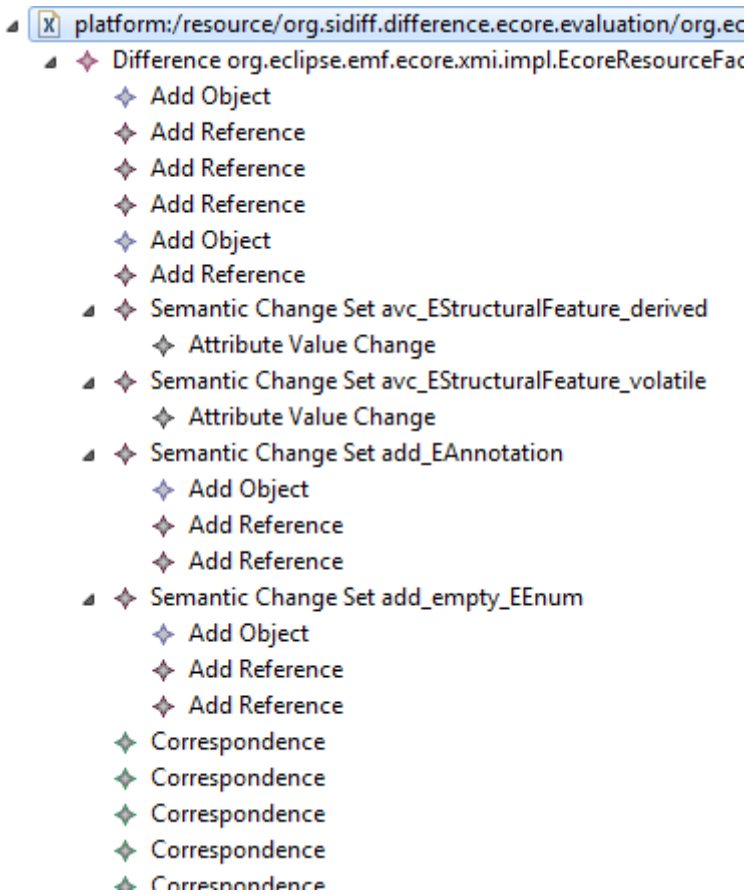


Abbildung 11.2: Anzeigen der semantisch gelifteten Differenz

12 Evaluierung

Für die Evaluierung wurde ein Ecore Modell aus dem Eclipse Modeling Framework (EMF) ausgewählt (org.eclipse.emf.codegen.ecore). Verglichen wurden einmal die Versionen 1.0 bis 10.0 und in 5er Schritten die Versionen 1.0 bis 45.0. Die Ergebnisse der Semantic-Lifting Berechnungen sind in folgender Tabelle 12.1 aufgelistet.

Tabelle 12.1: GenModel Auswertung

Version	SCS	GLL/ULL	KF	K	LS	ML	LL
01x02	7	10/0	1,4	105	2	345	797
02x03	6	24/0	4	106	1	147	411
03x04	1	4/0	4	112	1	122	223
04x05	1	4/0	4	113	1	129	104
05x06	1	1/0	2,6	114	1	113	81
06x07	9	23/0	4	114	2	120	464
07x08	1	4/0	4	121	1	126	111
08x09	1	4/0	4	122	1	146	240
09x10	1	4/0	4	123	1	144	110
01x05	15	42/0	2,8	105	2	393	1246
05x10	13	36/0	2,8	114	2	387	960
10x15	20	64/0	3,2	122	3	279	2779
15x20	13	46/0	3,5	136	3	248	1275
20x25	14	49/0	3,5	148	2	419	1823
25x30	6	21/0	3,5	161	2	213	675
30x35	17	64/0	3,8	167	2	214	3007
35x40	7	27/0	3,9	184	1	195	582
40x45	14	37/0	2,6	191	2	228	797

- **SCS:** Anzahl der Semantic-Chang-Sets.
- **GLL/ULL:** Gruppierte / Ungruppierte low-level Änderungen.

- **KF:** Der mittlere Kompressionsfaktor $\frac{SCS}{GLL}$.
- **K:** Anzahl der Korrespondenzen
- **LS:** Lifting Schritte zum liften sequenziell abhängiger Editieroperationen.
- **ML:** Laufzeit von Matching und Differenz-Ableitung in *ms*.
- **LL:** Lifting Laufzeit in *ms*.

Was zunächst positiv auffällt ist, dass alle Editieroperationen erkannt wurden, so dass keine ungruppierten low-level Änderungen in der Differenz zurück bleiben. Die dadurch erreichte Kompression liegt im Mittel bei 3,2, was bei Ecore auch in etwa zu erwarten ist, da meistens 3 low-level Änderungen anfallen, wenn ein Objekt mit Referenzen vom Container zum Objekt und vom Objekt zum Container eingefügt wird.

Was bei den Korrespondenzen auffällt ist, dass die Modelle in der Regel wachsen und nur selten Objekte entfernt werden. Auch sind die Anzahl der Änderungen von einer Modell Version zur nächsten in den meisten Fällen relativ klein. Um auch sequentiell abhängige Editieroperationen zu erkennen, werden für dieses Modell nie mehr als 3 Lifting Schritte benötigt, was zeigt, dass die Abhängigkeiten zwischen den Editieroperationen meistens nicht allzu komplex werden.

Die Laufzeit des Semantic-Lifting hängt hauptsächlich von der Anzahl der low-level Änderungen ab und nicht von der Anzahl der Korrespondenzen. Also auch nicht von der Größe des Modells. Dieses ist genau das Verhalten, welches die Optimierungen in Abschnitt 8.3 bewirken soll. Insgesamt sind die Ergebnisse also schon sehr nah an dem, was erwartet wurde.

13 Schlussfolgerung

In dieser Arbeit wurde ein Proof-Of-Concept des in [KKT11] beschriebenen Konzepts durchgeführt. Dabei wurde für ein ausgewähltes Metamodell ein vollständiger Durchlauf durch alle Phasen des Konzepts beschrieben. Von der Erstellung der Editierregeln und der automatischen Generierung der entsprechenden Erkennungsregeln, über die Berechnung einer technischen Differenz, bis hin zur vollständig semantisch gelifteten Differenz.

Es wurde gezeigt, dass es möglich ist mit Hilfe von Henshin als Mustererkennungswerkzeug den low-level Änderungen einer technischen Differenz wieder eine bestimmte Editieroperation zuzuordnen, was in den Praxistests auch zu durchweg guten und vor allem aus Benutzersicht lesbaren Ergebnissen geführt hat. Durch die Strukturierung der Differenz lässt sich auf den ersten Blick nachvollziehen welche und wie viele Änderungsschritte an einem Modell durchgeführt wurden, ohne jede low-level Änderung einzeln zu betrachten. Außerdem werden dem Benutzer einige grafische Werkzeuge an die Hand gegeben, um die Semantic-Lifiting-Engine zu konfigurieren und um die Berechnung zu steuern.

In der Praxis hat sich auch gezeigt, dass der Algorithmus auf relativ gute Berechnungszeiten kommt. Auf technischer Ebene wurde dies durch ein paar auf den Algorithmus angepasster Optimierungen erreicht (Abschnitt 8.2 und 8.3). Hier könnte aber bei einer genaueren Analyse wahrscheinlich noch Potenzial für weitere Optimierungen gefunden werden. Zum Beispiel durch eine genauere (ggf. statische) Analyse der Erkennungsregeln um nicht anwendbare Regeln noch gezielter auszufiltern. Ggf. könnte auch eine Minimierung des Arbeitsgraphen auf dem die Erkennungsregeln arbeiten einen Performance Gewinn bringen. Ein Großteil des Arbeitsgraphen wird häufig für die Erkennung der Editieroperationen nicht benötigt.

Zwei Teile, die konzeptuell Probleme bereitet haben, waren zum einen das Liften von sequentiell abhängigen Editieroperationen (Abschnitt 10) und zum anderen das Referenzieren von importierten Modellen (Abschnitt 7). Letzteres könnte man auch als technisches Problem ansehen. Für beide Probleme wurden aber auch Lösungsansätze geliefert. In der Praxis tun diese in den meisten Fällen ihren Dienst. Aus theoretischer Sicht ist hier aber noch etwas Analysebedarf. Ansonsten ließen sich die anderen Teile des Algorithmus

wie im Konzept [KKT11] beschrieben ohne letztendliche Probleme umsetzen.

Aufbauende Projekte auf dieser Arbeit, könnten vor allem sich an die Differenz Pipline (Abbildung 3.1) anschließende Funktionalitäten sein. Neben der Gruppierung von low-level Änderungen wäre es für den Benutzer sicherlich hilfreich, wenn die entsprechenden Änderungen auch anschaulich visualisiert werden könnten. Um die gewonnen Differenzen dann weiter zu verwerten, könnten sich aber auch typische Repository Funktionen, wie z.B. das Mergen von Modellen auf Basis der einzelnen Editierschritte, anschließen. Der Vorteil bei Mergen auf semantischer Ebene ist, dass ein dazu berechneter Patch auch auf Modelle angewendet werden kann die sich vom Ausgangsmodell unterscheiden.

Ein weiterer Punkt der bei der Erstellung der Editierregeln (Abschnitt 4) auffällt ist, dass die Erzeugung der Atomic-Editierregeln relativ schematisch von der Hand geht. In der Praxis verbraucht diese Aufgabe aber selbst für kleinere Metamodelle einiges an Zeit und erfordert außerdem ein genaues Verständnis des Metamodells. Um den Konfigurationsaufwand des Algorithmus zu minimieren wäre es denkbar, dem Benutzer beim Erstellen der Editierregel einen Großteil der Arbeit abzunehmen, indem man die Regeln soweit wie möglich automatisch aus dem Metamodell ableitet.

Insgesamt lässt sich sagen, dass die technischen Ergebnisse der Implementierung mit den erwarteten aus dem Konzept [KKT11] übereinstimmen. Die Editieroperationen werden wieder erkannt, wodurch sich die Komplexität der Differenz aus Benutzersicht stark verringert. Außerdem gibt es in diesem Bereich bisher keine vergleichbaren Ansätze.

> „We are not aware of a generic algorithm which semantically lifts the low-level differences and which can be adapted to a large variety of model types.“ [KKT11] (S.10)

Alles in allem lassen sich die aus diesem Projekt resultierenden Ergebnisse also durchaus als vielversprechend bezeichnen.

Abbildungsverzeichnis

Literaturverzeichnis

[ABJ+10] Arendt, Thorsten ; Biermann, Enrico ; Jurack, Stefan ; Krause, Christian ; Taentzer, Gabriele: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: *MoDELS 2010: Proceedings of the 13th Int. Conference on Model Driven Engineering Languages and Systems*, 2010

[BE09] Bendix, Lars ; Emanuelsson, Pär: Collaborative Work with Software Models – Industrial Experience and Requirements. In: *Proc. Intl. Conf. Model-Based Systems Engineering*, 2009

[BESW10] Biermann, Enrico ; Ermel, Claudia ; Schmidt, Johann ; Warning, Angeline: Visual Modeling of Controlled EMF Model Transformation using Henshin. In: *GraBaTs 2010: Proceedings of the Fourth International Workshop on Graph-Based Tools*, 2010

[BG05] Bacvanski, Vladimir ; Graff, Petter: Mastering Eclipse Modeling Framework. In: *EclipseCon 2005*, InferData Ltd., 2005

[Ebe11] Ebert, Ralf: *Eclipse RCP*. 2011

[EMF12] *http://www.eclipse.org/modeling/emf/*. 2012

[ERe12] *http://www.eclipse.org/modeling/emft/refactor/*. 2012

[GPR06] Gruhn, Volker ; Pieper, Daniel ; Röttgers, Carsten: *MDA: Effektives Softwareengineering mit UML2 und Eclipse*. Springer, 2006

[HEN12] *http://www.eclipse.org/modeling/emft/henshin/*. 2012

[KKT11] Kehrer, Timo ; Kelter, Udo ; Taentzer, Gabriele: A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In: *ASE 2011: Proc. 26th IEEE/ACM Intl. Conf. Automated Software Engineering*, 2011

[KPRP09] Kolovos, Dimitrios S. ; Pierantonio, Alfonso ; Ruscio, Davide D. ; Paige, Richard F.: Different Models for Model Matching. In: *IEEE 2009: Proc. Intl. ICSE Workshop on Comparison and Versioning of Software Models*, 2009

[MM03] Miller, Joaquin ; Mukerji, Jishnu: MDA Guide Version 1.0.1. In: *Object Management Group*, 2003

[MOF11] *OMG Meta Object Facility (MOF) Core Specification Version 2.4.1.* Object Management Group, 2011

[PTN+07] Pietrek, Georg ; Trompeter, Jens ; Niehues, Benedikt ; Kamann, Thorsten ; Holzer, Boris ; Kloss, Michael ; Thoms, Karsten ; Beltran, Juan Carlos F. ; Mork, Steffen: *Modellgetriebene Softwareentwicklung. MDA und MDSD in der Praxis.* Georg Pietrek and Jens Trompeter, 2007

[RIH12] *http://www.mozilla.org/rhino/*. 2012

[SBPM09] Steinberg, Dave ; Budinsky, Frank ; Paternostro, Marcelo ; Merks, Ed: *Eclipse Modeling Framework, Second Edition.* 2009

[SiD11] *http://www.sidiff.org*. 2011

[TJF+09] Tisi, Massimo ; Jouault, Frédéric ; Fraternali, Piero ; Ceri, Stefano ; Bézivin, Jean: On the Use of Higher-Order Model Transformations. In: *Proceedings of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA)*, Springer, 2009, S. 18–33

[Wie10] Wierse, Gerd: A Catalog of Modeling Patterns for the Unified Modeling Language. In: *DFG: Systematische Entwicklung komplexer Software in verteilten Teams*, 2010

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig unter ausschließlicher Nutzung der angegebenen Quellen und Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche gekennzeichnet.
Diese Arbeit lag in gleicher oder ähnlicher Weise noch keiner Prüfungsbehörde vor und wurde bisher noch nicht veröffentlicht.

Siegen, 23. Februar 2012

Manuel Ohrndorf