

SiLift - Benutzerhandbuch für Entwickler

Universität Siegen - Praktische Informatik

13. April 2014

Inhaltsverzeichnis

1	Einleitung	3
2	Voraussetzung und Installation	3
3	SiLift-Architektur	4
4	Integration eigener Matching-Engines	7
5	Erstellen eines Technical Difference Builders	13
6	Erstellen von Editierregeln	16
6.1	Metamodell	16
6.2	Exkurs in EMF-Henshin	16
6.3	Atomare Editierregeln	17
7	Generieren von Erkennungsregeln	30
7.1	Rulebase Plug-in Projekt	30
7.2	Rulebase File	30
7.3	Der Rulebase-Manager	35
7.4	Erkennungsregeln deployen und nutzen	36
8	Links und weitere Informationen	38

1 Einleitung

SiLift ist ein *Eclipse*-basiertes Framework mit dessen Hilfe sich Differenzen von *EMF-Modellen semantisch liften* lassen.

Generell werden dabei alle *EMF*-basierten Modellierungssprachen unterstützt, sofern die entsprechenden *Editierregeln* implementiert bzw. aus diesen *Erkennungsregeln* abgeleitet wurden. Dieses Benutzerhandbuch umfasst neben einer Installationsanleitung einen Einblick in die *SiLift*-Architektur sowie einführendes Tutorial, in dem Sie anhand eines kleinen *Metamodells* lernen mit Hilfe von *EMF-Henshin* Editierregeln zu erstellen um danach aus diesen die Erkennungsregeln abzuleiten.

2 Voraussetzung und Installation

SiLift ist als *Eclipse-Feature* unter folgender *Update-Site* erhältlich:

<http://pi.informatik.uni-siegen.de/Projekte/SiLift/updatesite>.

Hinweis: Vergewissern Sie sich, ob ihr *Eclipse* die notwendigen Voraussetzungen erfüllt. Eine Liste der benötigten Plugins ist unter <http://pi.informatik.uni-siegen.de/Projekte/SiLift/download.php> zu finden. Bitte beachten Sie dabei die entsprechenden Hinweise zu den jeweiligen Versionen.

Sofern alle Voraussetzungen erfüllt sind, kann *SiLift* wie gewohnt über den Menüpunkt **Help** ▸ **Install New Software...** installiert werden (vgl. Abb. 1).

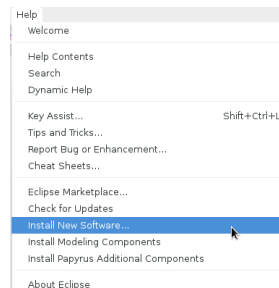


Abbildung 1: Eclipse: Install New Software...

Es sollten Ihnen vier Kategorien angezeigt werden (vgl. Abb. 2).

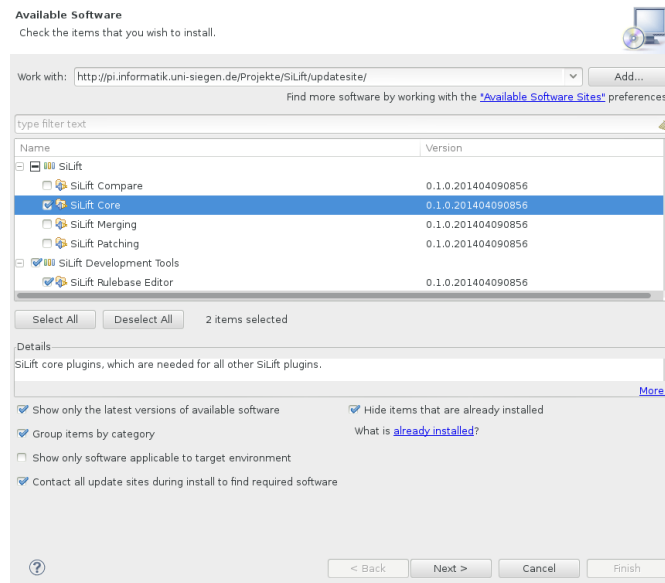


Abbildung 2: SiLift Update Site

Für die folgenden Tutorials benötigen das Feature **SiLift Core** aus der Kategorie **SiLift** und das Feature **SiLift Rule Base Editor** aus der Kategorie **SiLift Development Tools**. Danach klicken Sie auf **Next** und folgen dem Installationsassistenten.

3 SiLift-Architektur

Mit SiLift können Differenzen von *EMF-basierten* Modellen, d.h. Modelle die auf dem *Ecore-Metamodell* basieren, semantisch geliftet werden. Basierend auf einer gelifteten Differenz lassen sich *Patches* bilden, sowie Modelle mischen.

Für *domainspezifische* Modellierungssprachen bedeutet das, dass deren Metamodell (vgl. 6.1) zuerst in ein entsprechendes *Ecore-Modell* übertragen, sowie ein entsprechender *Matcher* (vgl. 4) und *Technical Difference Builder* (vgl. 5) bereit gestellt werden müssen, bevor Editierregeln implementiert und Erkennungsregeln abgeleitet werden können.

Dieser Abschnitt führt die *SiLift-Pipeline* ein und dient als Grundlage der folgenden Abschnitte.

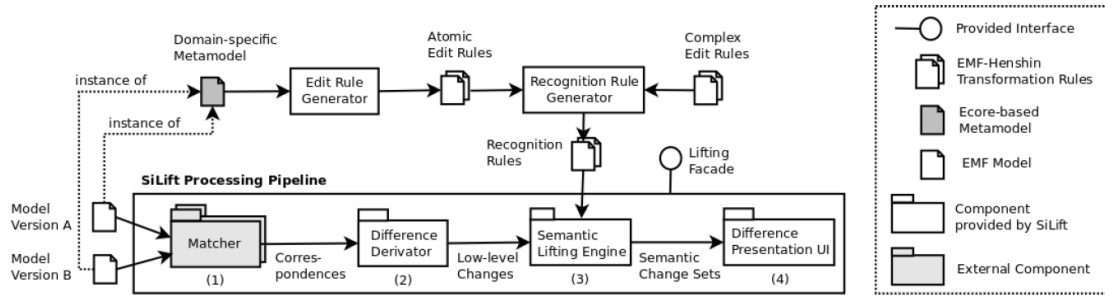


Abbildung 3: SiLift Processing Pipeline

Die Vorgehensweise von SiLift lässt sich am besten mit einer vierstufigen *Pipeline*, wie in Abbildung 3 dargestellt, vergleichen. Als Eingabe dienen immer zwei Versionen eines Modells:

1. **Matching:** Aufgabe eines *Matcher* ist es, die korrespondierenden Elemente aus Modell A und Modell B, also die Elemente, die in beiden Modellen übereinstimmen, zu identifizieren. Dabei ist das Ergebnis vor allem davon abhängig anhand welcher Kriterien der *Matcher* eine Übereinstimmung festlegt. Hier wird unter anderem unterschieden zwischen *ID*-, *signatur*- und *ähnlichkeitsbasierten* Verfahren.

In SiLift stehen standardmäßig folgende *Matcher-Engines* zur Verfügung:

- **EcoreID Matcher:** Ein *ID-basierter* *Matcher* (nutzt Werte von Attributen, die im Metamodell als ID-Attribute deklariert sind).
- **EMF Compare:** Unterstützt alle drei Verfahren. **EMF Compare** kann unter **Window** > **Preferences**: **EMF Compare** konfiguriert werden.¹
- **NamedElement Matcher:** Ein *signaturbasierter* *Matcher*, welcher die entsprechenden Korrespondenzen anhand der Werte der jeweiligen Namensattribute bestimmt.
- **URIFragment Matcher:** Ein *signaturbasierter* *Matcher*, welcher die entsprechenden Korrespondenzen anhand der Werte der *Uri* der Elemente bestimmt (z.B. `eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore #//EString"`).
- **UUID Matcher:** Ein *ID-basierter* *Matcher* (basiert auf XMI-IDs der XMI-Repräsentationen der Modelle, falls vorhanden).

Diese Liste ist keineswegs abgeschlossen und kann durch zusätzliche Matching-Engines, wie z.B. *SiDiff* oder auch eigener *Matcher* ergänzt werden (siehe Ab-

¹Informationen zum **EMF Compare Project** finden Sie unter <http://www.eclipse.org/emf/compare>.

schnitt 4).

2. **Difference derivation:** Ausgehend von den gefunden Korrespondenzen berechnet der *Difference Derivator* eine technische Differenz (*low-level difference*) der Modelle. Alle Objekte und Referenzen, für die keine Korrespondenz existiert müssen demnach entweder in Modell B hinzugefügt, oder aus Modell A entfernt worden sein.
3. **Semantic Lifting:** Die zuvor berechnete technische Differenz enthält alle Änderungen auf Basis des Metamodells. Diese sollen nun semantisch geliftet werden. Bei einer *semantisch gelifteten Differenz* handelt es um eine halbgeordnete Menge von auf einem vorhandenen Modell (dem Basismodell) ausgeführten *Editieroperationen*. Durch das liften der technischen Differenz werden die einzelnen Änderungen mit Hilfe von *Erkennungsregeln* (engl. *recognition rules*) in sogenannte *Semantic Change Sets* gruppiert. Diese repräsentieren wiederum jeweils eine vom Benutzer ausgeführte Editieroperation. Das Verhalten einer Editieroperation wird durch die zugehörige *Editierregel* definiert, aus denen sich mit Hilfe des *Recognition Rules Generators* die Erkennungsregeln ableiten lassen. Was wiederum eine gültige bzw. sinnvolle Editierregel ist hängt zum einem vom Metamodell, zum anderen von den Benutzerpräferenzen ab. Daher lassen sich die Editierregeln und somit auch die Erkennungsregeln grob zweier sogenannter Regelbasen (engl. *Rule Bases*) zuordnen:

- **Atomic Rule Base:** Atomare Regeln umfassen das Erzeugen (engl. *create*), Löschen (engl. *delete*), Verschieben von Elementen (engl. *move*) sowie das Ändern von Attributwertenv(engl. *change*). Sie lassen sich nicht in kleinere Teile zerlegen, ohne dass deren Anwendung zu einem inkonsistenten Modell führen würde.

Atomaren Regeln können mit Hilfe eines *Editrulegenerators*² direkt aus dem Metamodell abgeleitet werden. Problematisch wird es, wenn weitere Restriktionen (engl. *Constraints*), wie sie bspw. die UML in Form von *OCL-Ausdrücken* benutzt, die Konsistenzkriterien eines Modells bzw. dessen Elemente weiter eingrenzen. I.d.R. werden diese nicht bei der Implementierung eines Metamodells berücksichtigt. Hier bleibt nur die Möglichkeit die Regeln manuell zu editieren bzw. anzupassen.

²Weitere Information zum Editrulegenerator finden Sie unter <http://pi.informatik.uni-siegen.de/mrindt/SERGe.php>.

- **Complex Rule Base:** Die komplexen Editierregeln setzen sich i.d.R. aus den atomaren und anderen komplexen Regeln zusammen und beschreiben umfangreichere Editieroperationen, welche vor allem beim *Refactoring* auftreten. Da solche Refactorings sehr benutzerspezifisch sind müssen komplexe Regeln generell von Hand erstellt werden.
4. **Difference Presentation UI:** SiLift stellt zwei Benutzerschnittstellen (engl. *User Interfaces*) zur Verfügung, um die semantisch gelifteten Differenzen anzuzeigen: einen Baum-basierten und einen grafischen Editor, in dem die Differenzen *gehighlightet* werden.³

4 Integration eigener Matching-Engines

Wie bereits erwähnt lassen sich auch eigene Matching-Engines in die SiLift-Processing-Pipeline integrieren.

Erstellen Sie dazu über **File** ▷ **New** ▷ **Other...**: **Plug-in Development** ▷ **Plug-in Project** eines neues Plugin und öffnen Sie die **MANIFEST.MF**. Wechseln Sie auf den Reiter **Dependencies** und fügen Sie die in Abbildung 4 markierten Abhängigkeiten hinzu.

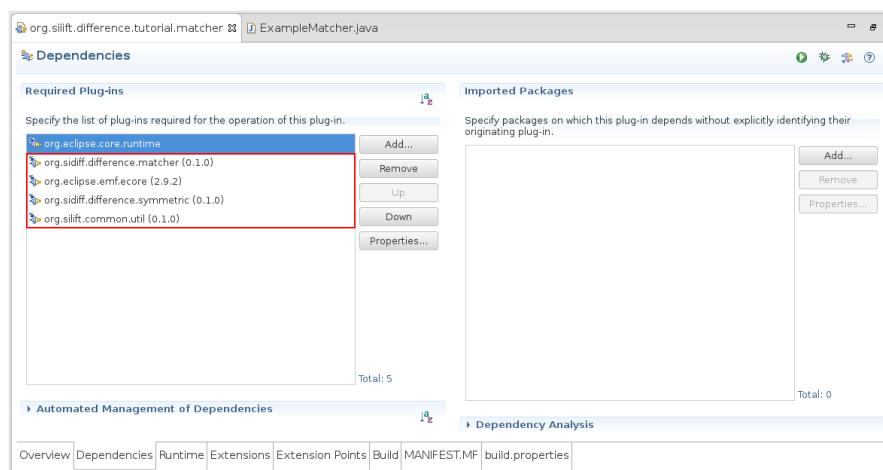
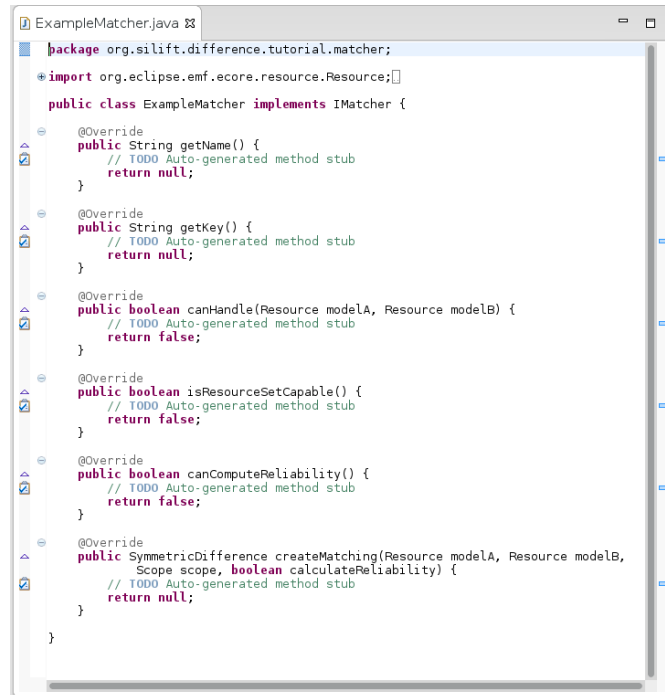


Abbildung 4: **MANIFEST.MF** ▷ **Dependencies**

Als nächstes wird eine Klasse benötigt die die Schnittstelle **IMatcher** implementiert (vgl. Abb. 5). Neben den zu implementierenden Methoden ist es Ihnen frei gestellt, ob sie Ihren Matcher in dieser Klasse implementieren oder diese nur als Adapter für einen ausgelagerten Matcher nutzen.

³Beispielansichten finden Sie im **SiLift - Benutzerhandbuch für Endanwender**



```

ExampleMatcher.java
package org.silift.difference.tutorial.matcher;

import org.eclipse.emf.ecore.resource.Resource;

public class ExampleMatcher implements IMatcher {

    @Override
    public String getName() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public String getKey() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public boolean canHandle(Resource modelA, Resource modelB) {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean isResourceSetCapable() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean canComputeReliability() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public SymmetricDifference createMatching(Resource modelA, Resource modelB,
        Scope scope, boolean calculateReliability) {
        // TODO Auto-generated method stub
        return null;
    }
}

```

Abbildung 5: Klasse `ExampleMatcher` implementiert `IMatcher`

Anstatt die Schnittstelle `IMatcher` von Grund auf zu implementieren, kann auch die “Convenience”-Klasse `BaseMatcher` erweitert werden (vgl. Abb. 6). Diese stellt bereits Methoden zum Iterieren über die Modellelemente bereit. Zusätzlich muss noch die Methode `isCorresponding` implementiert werden.

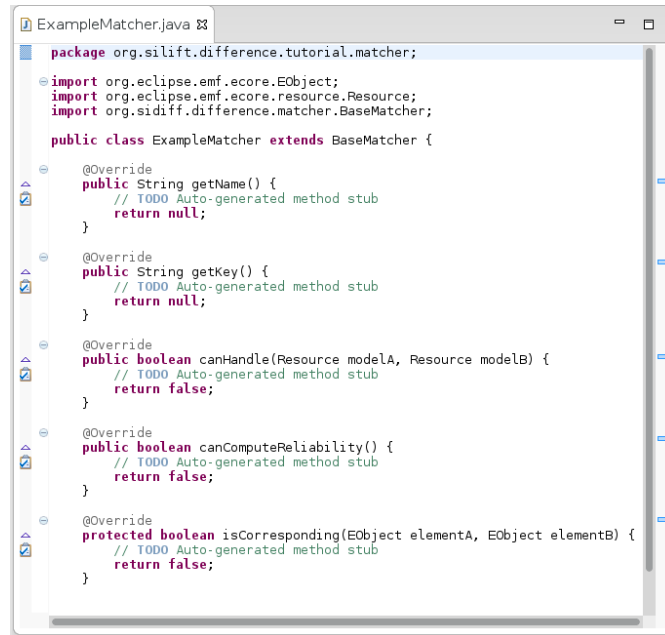


Abbildung 6: Klasse `ExampleMatcher` erweitert `BaseMatcher`

Danach muss das Plugin noch als Extension für SiLift definiert werden. Gehen Sie dazu wieder in die `MANIFEST.MF` und wechseln Sie auf den Reiter `Extensions` (vgl. Abb. 7).

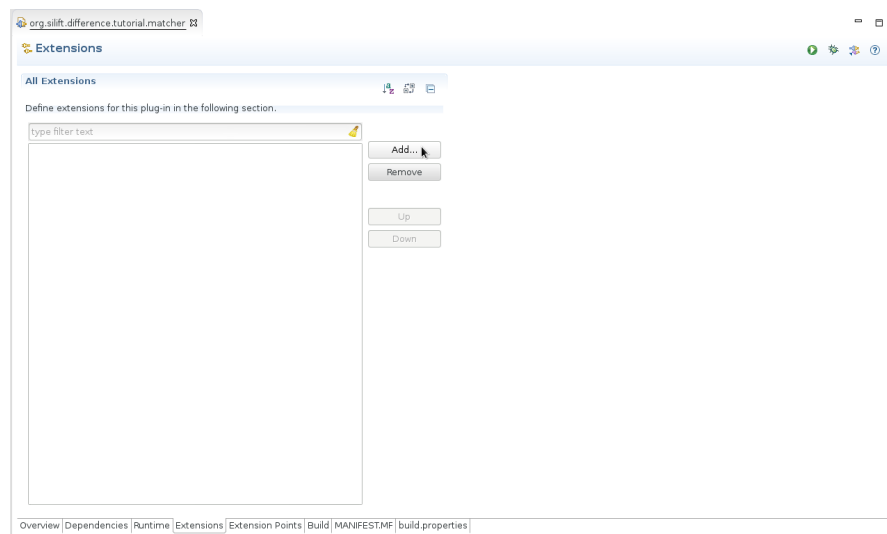


Abbildung 7: `MANIFEST.MF` ▸ `Extensions`

Klicken Sie auf auf `Add...` und selektieren Sie den *Extension Point* `org.sidiff.difference.matcher.matcher_extension` (vgl. Abb. 8).

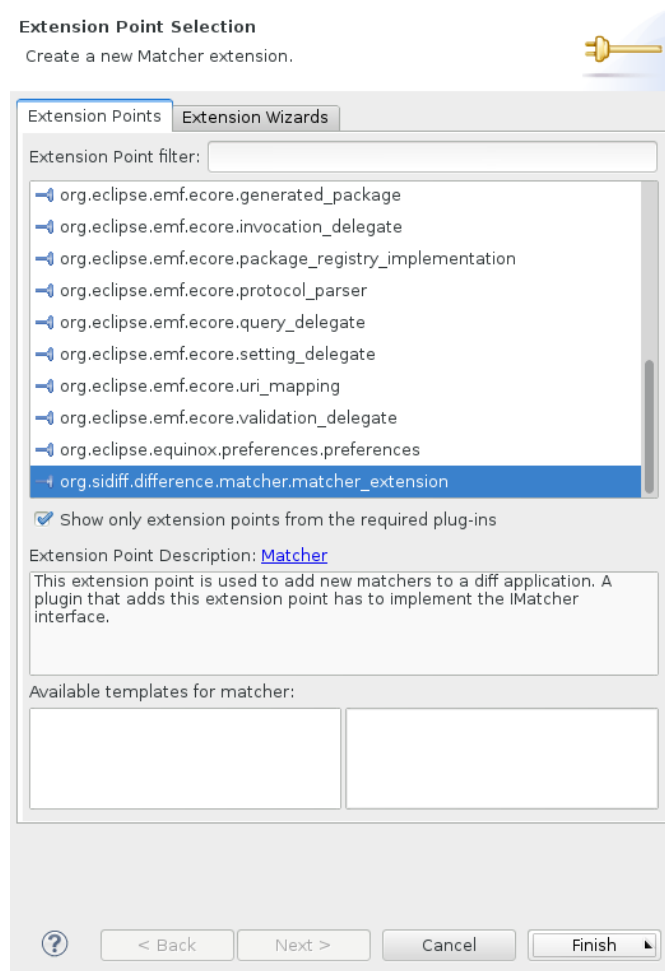


Abbildung 8: MANIFEST.MF ▷ Extensions: Extension Point Selection

Wechseln Sie in den Reiter `plugin.xml` und fügen dem eben erstellten Extension Point die entsprechende URI der Erweiterung bei (vgl. Abb. 9).

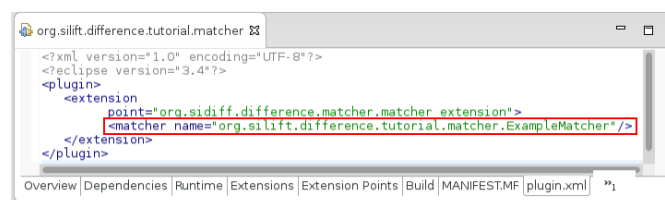


Abbildung 9: MANIFEST.MF ▷ `plugin.xml`

Als letztes muss das Plugin noch *deployed* werden. Öffnen Sie mit der rechten Maustaste auf Ihrem Plugin-Projekt im Package Explorer das Kontextmenü und klicken Sie auf

Export... (vgl. Abb. 10).

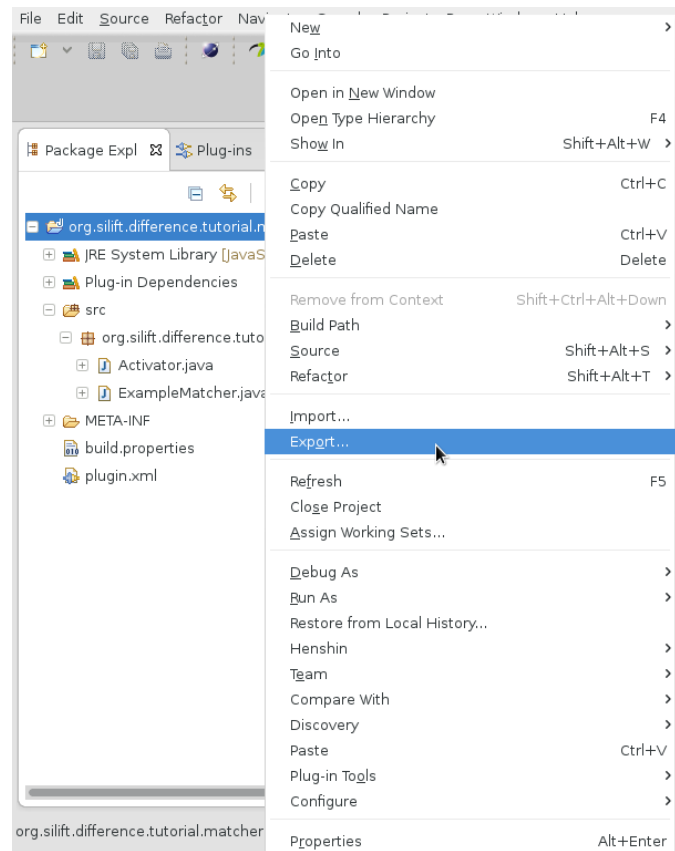


Abbildung 10: Export Plugin

Selektieren Sie **Plug-in Development** > **Deployable plug-ins and fragments** und klicken Sie **Next** (vgl. Abb. 11).

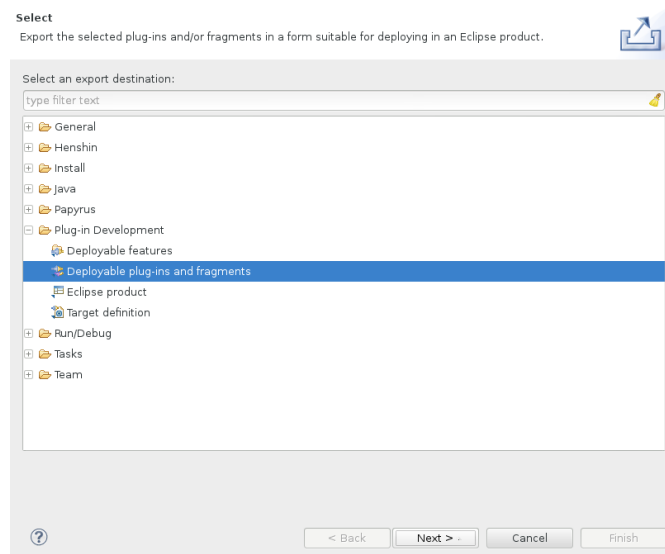


Abbildung 11: Export Wizard: Page 1

Wählen Sie als nächstes `install into host.Repository` aus und geben Sie den Pfad zum Plugin-Ordner Ihrer Eclipse-Installation an (vgl. Abb. 12). Klicken Sie auf **Finish** und starten Sie nach erfolgreicher Installation Ihr Eclipse neu.

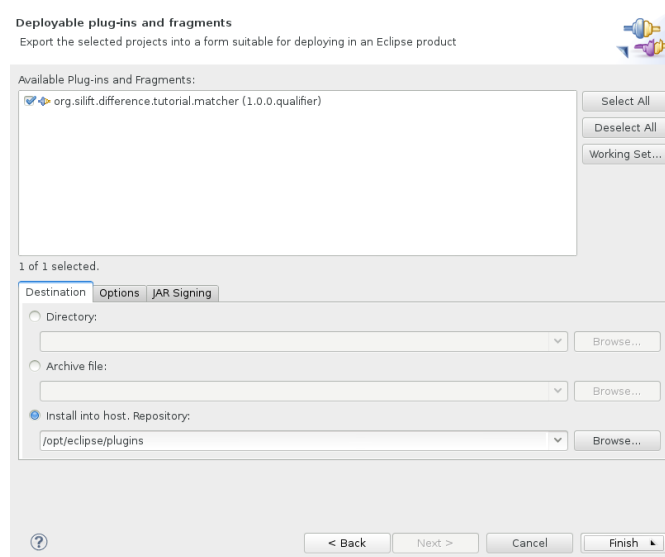


Abbildung 12: Export Wizard: Page 2

Ihr Matcher kann nun in Verbindung mit SiLift genutzt werden.

5 Erstellen eines Technical Difference Builders

Ein Matcher kann i.d.R. für mehrere Modelle unterschiedlichen Typs benutzt werden. Dahingegen muss für jeden Modelltyp (Metamodell) ein separater *Technical Difference Builder* erzeugt werden.

Erstellen Sie dazu über **File** ▷ **New** ▷ **Other...**: **Plug-in Development** ▷ **Plug-in Project** eines neues Plugin und öffnen Sie die **MANIFEST.MF**. Wechseln Sie auf den Reiter **Dependencies** und fügen Sie die in Abbildung 13 gelisteten Abhängigkeiten hinzu.

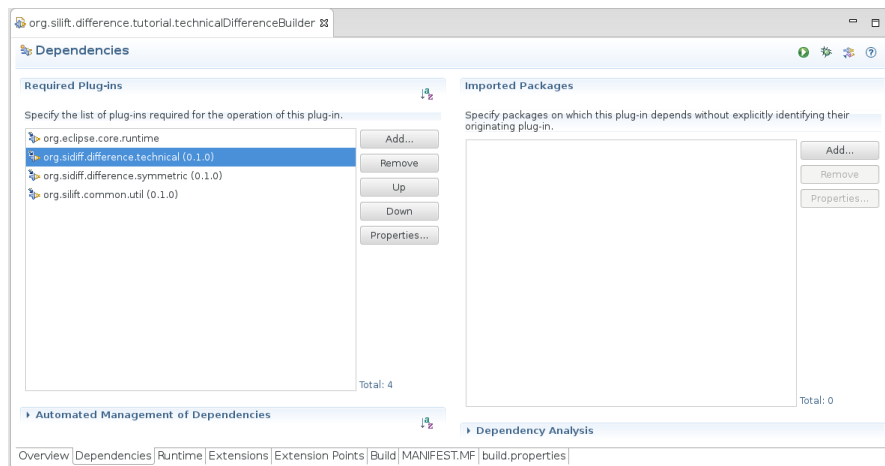


Abbildung 13: MANIFEST.MF ▷ Dependencies

Als nächstes wird eine Klasse benötigt die die Schnittstelle `ITechnicalDifferenceBuilder` implementiert (vgl. Abb. 14).

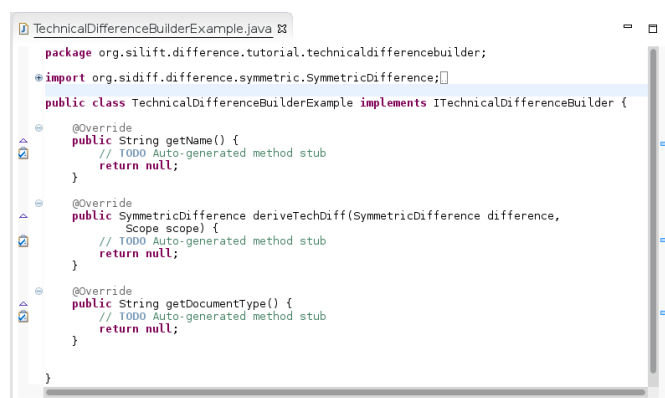
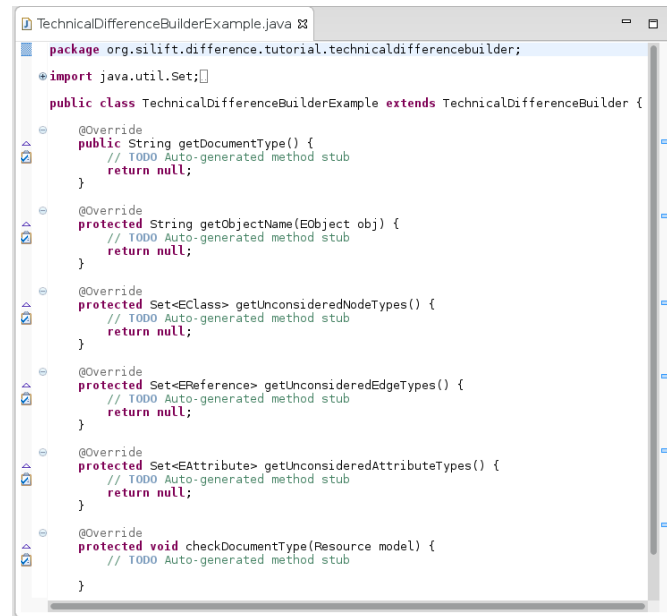


Abbildung 14: Klasse `TechnicalDifferenceBuilderExample` implementiert `ITechnicalDifferenceBuilder`

Anstatt die Schnittstelle `ITechnicalDifferenceBuilder` von Grund auf zu implementieren, kann auch die “Convenience”-Klasse `TechnicalDifferenceBuilder` erweitert werden.

Diese implementiert bereits die Methode `deriveTechDiff`. Durch die manuelle Implementierung der Methoden `getUnconsideredNodeTypes`, `getUnconsideredEdgeTypes` und `getUnconsideredAttributeTypes` können zudem Modellelemente gefiltert und somit von der Differenzbildung ausgeschlossen werden (vgl. Abb. 15).



```
TechnicalDifferenceBuilderExample.java
package org.sidiff.difference.tutorial.technicaldifferencebuilder;

import java.util.Set;

public class TechnicalDifferenceBuilderExample extends TechnicalDifferenceBuilder {

    @Override
    public String getDocumentType() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected String getObjectNames(EObject obj) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected Set<EClass> getUnconsideredNodeTypes() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected Set<EReference> getUnconsideredEdgeTypes() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected Set<EAttribute> getUnconsideredAttributeTypes() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    protected void checkDocumentType(Resource model) {
        // TODO Auto-generated method stub
    }
}
```

Abbildung 15: Klasse `TechnicalDifferenceBuilderExample` erweitert `TechnicalDifferenceBuilder`

Danach muss das Plugin analog zu Abschnitt 4 noch als Extension für `SiLift` definiert werden. Gehen Sie dazu wieder in die `MANIFEST.MF` und wechseln Sie auf den Reiter `Extensions`. Klicken Sie auf `Add...` und selektieren Sie den *Extension Point* `org.sidiff.difference.technical.technical_difference_builder_extension`.

Klicken sie danach mit der rechten Maustaste auf die Extension und wählen sie `New > technical` (vgl. Abb. 16).

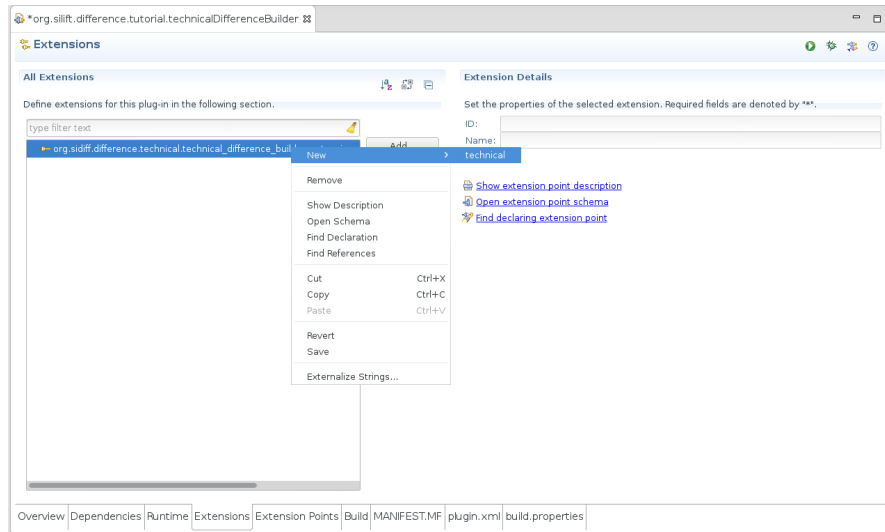


Abbildung 16: Klasse MANIFEST.MF ▷ Extensions

Als letztes müssen Sie noch unter **Extension Element Details** ▷ **difference_builder** Ihre zuvor erstellte Klasse eingeben (vgl. Abb. 13).

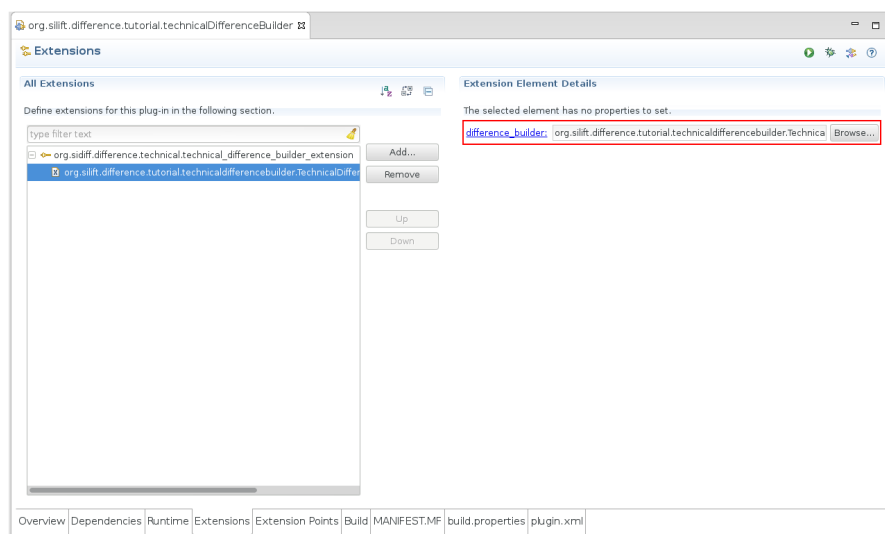


Abbildung 17: Klasse MANIFEST.MF ▷ Extensions: difference builder

Damit SiLift Ihren Difference Builder findet muss dieser noch analog zu Abschnitt 4 deployed werden.

6 Erstellen von Editierregeln

6.1 Metamodell

Abbildung 18 stellt ein Metamodell für *Zustandsautomaten* (engl. *statemachines*) ähnlich dem der UML dar.

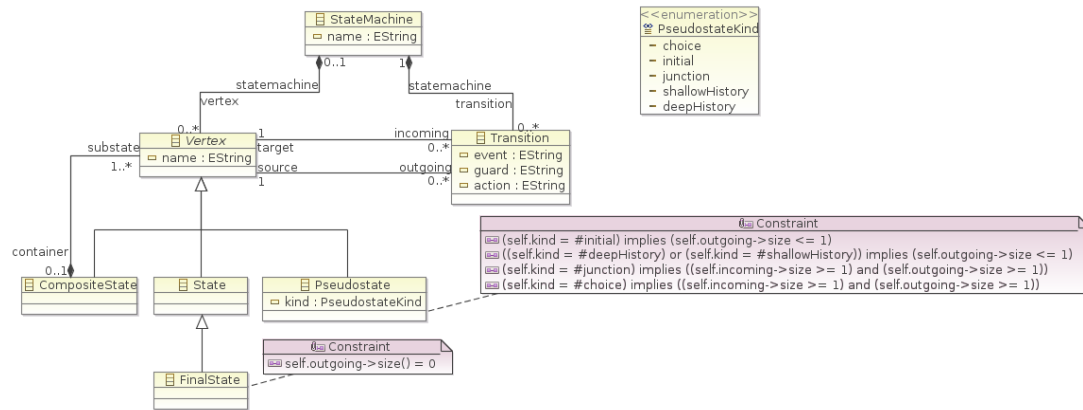


Abbildung 18: Zustandsautomat Metamodell

Ein *Zustandsautomat* (*StateMachine*) besteht aus einer endlichen Anzahl von *Zuständen* (*Vertex*) und *Transitionen* (*Transition*). Bei den Zuständen wird zwischen *normalen Zuständen* (*State*), *Pseudozuständen* (*Pseudostate*), *Endzuständen* (*FinalState*) und *zusammengesetzten Zuständen* (*CompositeState*) unterschieden. Eine *Transition* verbindet immer zwei Zustände und kann ein *Ereignis* (*Event*), eine *Bedingung* (*Guard*) und einer *Aktion* (*Action*) besitzen.

Neben den Kardinalitäten der Referenzen sind weitere Konsistenzkriterien durch *OCL-Ausdrücke* definiert. So darf z.B. eine Startzustand (*PseudostateKind: initial*) keine eingehende und maximal eine ausgehende Transition besitzen.

Im Folgendem lernen Sie ausgehend von dem vorgestellten Metamodell Editierregeln mit Hilfe von *Henshin-Regeln* zu erstellen.

Hinweis: SiLift unterstützt momentan nur *statisches EMF*, d.h. dass das Metamodell als *Eclipse-Plugin* “*deployed*“ sein muss.

6.2 Exkurs in EMF-Henshin

Bei *Henshin* handelt es sich um eine Transformationsprache bzw. ein Graphersetzungssystem für *EMF Modelle*. Modelle werden als *Graphen* interpretiert (man spricht hier

auch von dem Arbeitsgraphen, der das Modell repräsentiert) und nach *Teilgraphen* durchsucht, die vorher in einer sogenannten *Transformationsregeln* definiert wurden. Man spricht bei diesem Teilgraphen von der *Left-Hand Side* (kurz *LHS*) der Transformationsregel. Wird ein solcher Teilgraph gefunden wird er durch die sogenannte *Right-Hand Side* (kurz *RHS*) der Transformationsregel ersetzt. Dabei werden die Knoten und Kanten, die in der linken und nicht in der rechten Seite vorkommen gelöscht und die, die in der rechten Seite und nicht in der linken vorkommen werden erzeugt. Knoten, die in beiden Seiten vorkommen bleiben unverändert. Man spricht bei diesen Knoten auch vom Kontext der Transformationsregel. Dieser Kontext kann durch sogenannte *Negative Application Conditions* (kurz *NACs*) weiter eingeschränkt werden.⁴

Mehrere Regeln lassen sich in einem *Modul* (engl. *module*) zusammenfassen, wobei die Reihenfolge der Ausführung der Regeln durch sogenannte *Units* bestimmt wird: *Independent Unit*, *Sequential Unit*, *Conditional Unit*, *Priority Unit*, *Iterated Unit* und *Loop Unit*.

Hinweis: Momentan verlangt SiLift für jede Editieroperation immer ein Transformationssystem mit genau einer *mainUnit* (**Independent**, **Priority** oder **Sequential**) und einer darin enthaltenen Regel. Eine Ausnahme bilden geschachtelte Regeln, in der eine *Kernel-Regel* und eine beliebige Menge von *Multi-Regeln* definiert werden.

Henshin stellt zum Entwickeln von Transformationssystemen zwei Editoren zur Verfügung:

1. **Henshin Transformation Model Editor:** Ein baumbasierter Editor mit entsprechender linken und rechten Seite der Transformationsregeln (vgl. Abb.23: links).
2. **Henshin Diagram Editor:** Ein graphenbasierter Editor mit den “*Stereotypen*“ **create**, **delete**, **preserve**, **require** und **forbid** zum markieren entsprechender Operationen (vgl. Abb. 23: rechts). Knoten und Kanten die mit **delete** markiert sind, kommen nur in der *LHS* vor, wohingegen Knoten und Kanten, die mit **create** markiert sind nur in der *RHS* vorkommen. Knoten, die auf beiden Seiten vorkommen werden als **preserve** markiert. Mit **require** und **forbid** lassen sich Teilgraphen erzeugen, die in einem Kontext auftreten müssen bzw. nicht auftreten dürfen.

6.3 Atomare Editierregeln

Wie bereits erwähnt umfassen die atomaren Regeln das Erzeugen, Löschen, Verschieben von Elementen und das Ändern von Attributwerten. Um eine Editierregel zu erstellen

⁴Einführende Beispiele finden Sie unter <http://www.eclipse.org/henshin/examples.php>.

legen Sie ein neues Projekt an und wählen Sie **File** ▸ **New** ▸ **Other...** und hier **Henshin Model** aus. Klicken Sie auf **Next** (vgl. Abb. 19).

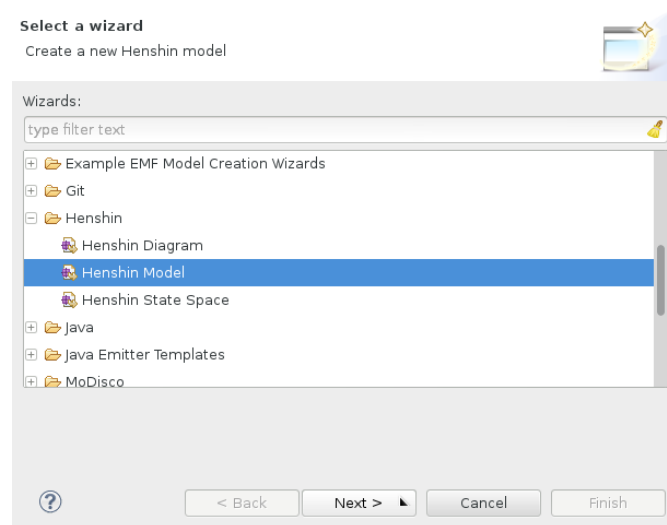


Abbildung 19: Erstellen eines neuen Transformationsmodells: Page 1

Die Datei kann einen beliebigen Namen bekommen, jedoch muss sie auf `_execute.henshin` enden. Da der Name der Regel eindeutig sein muss ist es zu empfehlen die Datei nach ihrer späteren Regel zu benennen. Somit behalten Sie den Überblick. Die folgende Regel soll einen Zustand in einen vorhandenen Zustandsautomaten einfügen. Hier würde sich bspw. der Name `Create_StateInStateMachine_execute.henshin` anbieten (vgl. Abb.20). Wählen sie den Zielordner und geben Sie den Dateinamen ein. Klicken Sie auf **Next**.

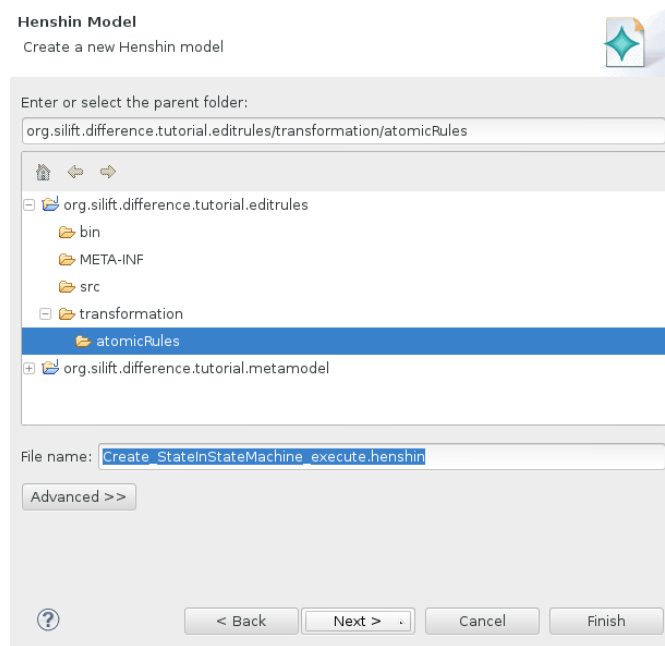


Abbildung 20: Erstellen eines neuen Transformationsmodells: Page 2

Als letztes müssen Sie noch Ihr Metamodel aus der Registry (**Add From Registry**) laden (vgl. Abb. 21).

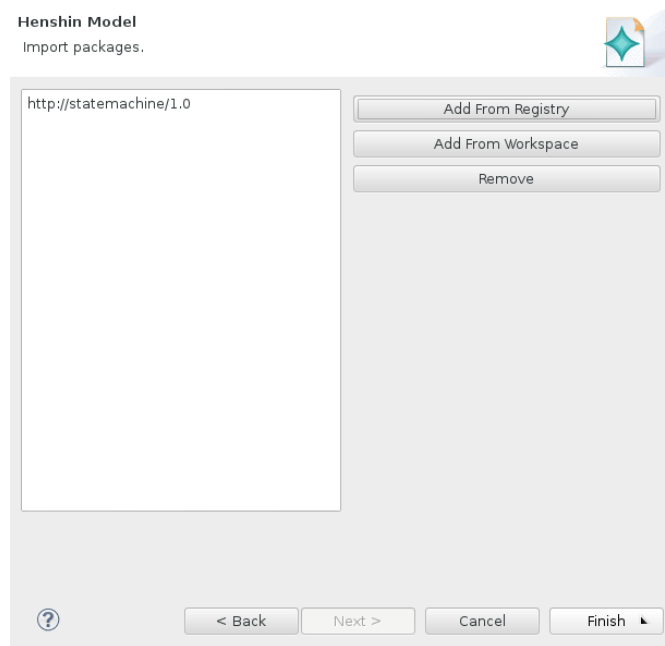


Abbildung 21: Erstellen eines neuen Transformationsmodells: Page 3

Um neben dem baumbasierten auch den graphenbasierten Editor nutzen zu können muss zusätzlich noch ein *Henshin-Diagramm* erzeugt werden. Klicken Sie hierzu mit der rechten Maustaste auf die `CREATE_StateInStateMachine_execute.henshin`, wählen Sie **Henshin** ▸ **Initialize Diagram File** aus und folgen Sie dem Assistenten (vgl. Abb. 22).

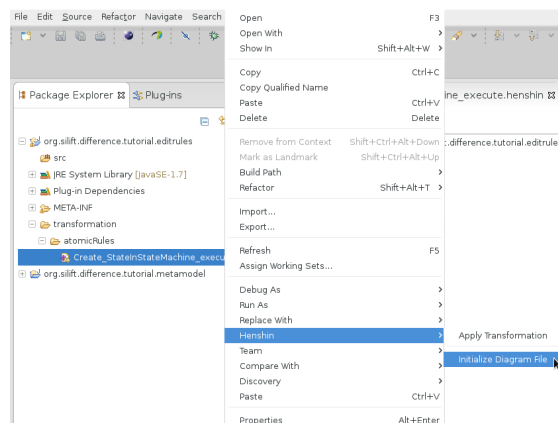


Abbildung 22: Erstellen eines Henshin Diagramms

Create-Regeln

Erstellen Sie eine Regel, indem sie aus der Palette das *Rule-Werkzeug* auswählen und danach auf die Arbeitsfläche klicken. Geben Sie der Regel den Namen `createStateInState Machine`. Einer Regel können ähnlich einer Funktion oder Methode Parameter übergeben werden (vgl. Abb. 23). Dabei lässt sich zwischen *Object-* und *Value-Parametern* unterscheiden:

- **Object-Parameter:** Durch Objekt-Parameter können Objekte an die Regel übergeben werden, um diese gezielt aus dem Arbeitsgraphen zu löschen, in dem Arbeitsgraphen zu erzeugen oder zu verändern. Des Weiteren können dadurch Kontextobjekte eindeutig bestimmt werden. Sofern der Objekt-Parameter ein zu löschenden Knoten identifiziert handelt es sich um einen *in*-Parameter. Repräsentiert der Parameter hingegen ein zu erzeugendes Objekt handelt es sich um einen *out*-Parameter (vgl. Seite 24).
- **Value-Parameter:** Mit Hilfe von Value-Parametern können (Attribut-) Werte (z.B. Zeichenketten, Zahlen etc.) übergeben, verglichen und neu berechnet werden.

Für das Erzeugen eines neuen Zustands wird ein Parameter (**Selected**) benötigt, der den Zustandsautomaten, in den ein neuer Zustand eingefügt werden soll, identifiziert, sowie ein Parameter der den erzeugten Zustand zurück gibt (**New**).

Danach müssen mit Hilfe des *Node-Werkzeugs* die beiden Knoten (**Nodes**) vom Typ **StateMachine** und **State** erstellt und der entsprechende Parameter voran gestellt werden (vgl. Abb. 23). Wählen Sie aus der Palette das *Edge-Werkzeug* und verbinden Sie die beiden Knoten miteinander, um die Kanten zwischen den gerade erstellten Knoten zu ziehen. Sofern nicht bereits der gewünschte Typ der Kanten erzeugt wurde, kann dieser unter den *Properties* angepasst werden. Selektieren Sie dazu die entsprechende Kante mit der rechten Maustaste und wählen Sie **Show Properties View**.

Hinweis: Abgeleitete (engl. *derived*) Referenzen dürfen nicht durch Editierregeln erstellt werden.

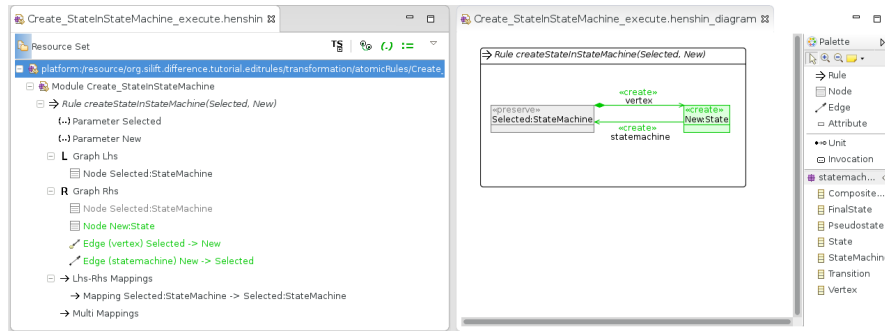


Abbildung 23: Editierregel: createStateInStateMachine

Wie bereits erwähnt, verlangt SiLift für jede Editieroperation immer ein Modul mit genau einer Unit und (mit Ausnahme geschachtelter Regeln) einer darin enthaltenen Regel. Diese Unit muss per Konvention den Namen *mainUnit* tragen.

Zwar lassen sich die Units ebenfalls im graphenbasierten Editor erstellen, dennoch ist in unserem Fall der baumbasierte Editor zu bevorzugen.

Wechseln Sie in den baumbasierten Editor, indem sie die `*_execute.henshin` öffnen. Im Gegensatz zur graphischen Repräsentation der Regel, lassen sich im baumbasierten Editor die linke und rechte Seite der Transformationsregel genau unterscheiden. Beim Anwenden einer Regel wird der Arbeitsgraph auf das Vorkommen des Teilgraphen der LHS untersucht. In unserem Beispiel also auf den Teilgraphen, der nur einen Knoten vom Typ `StateMachine` enthält. Durch einen Objekt-Parameter sagen wir zusätzlich um welchen Knoten vom Typ `StateMachine` es sich genau handeln muss. Existiert ein solcher Teilgraph, so wird dieser durch den Graphen der RHS ersetzt. In diesem Fall also einer `StateMachine`, einem `State` und den beiden Referenzen `statemachine` und `state`. Zu beachten ist, dass es sich bei dem `StateMachine`-Knoten um einen “*preserve*“-Anteil der Regel handelt, also einem Knoten der eigentlich nicht verändert wird. Dazu muss dieser Knoten auf beiden Seiten der Regel vorkommen und ein sogenanntes *Mapping* zwischen diesen existieren (vgl. Abb. 23: links).

Zum Erstellen der *mainUnit* klicken Sie mit der rechten Maustaste auf `Module` und wählen `New Child > PriorityUnit` (vgl. Abb 24).

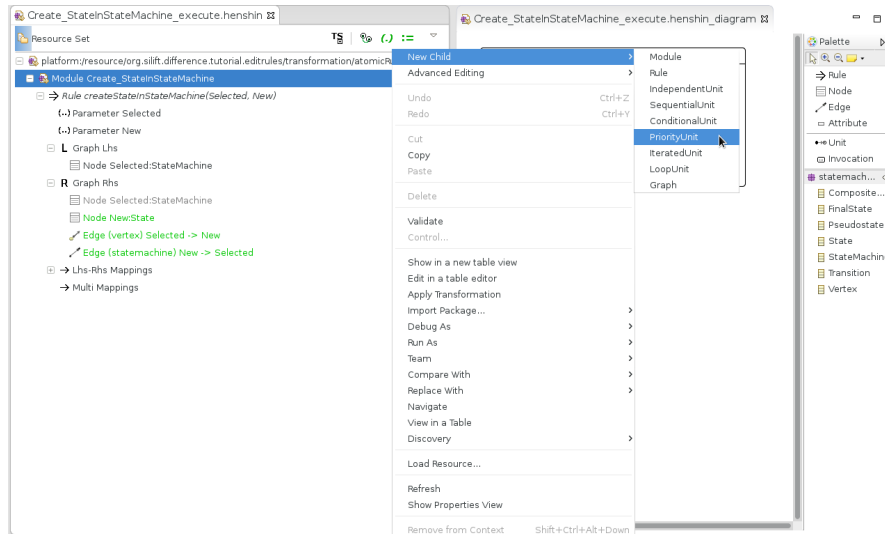


Abbildung 24: Editierregel: Erstellen einer *Priority Unit*

Nennen Sie diese in “*mainUnit*“ um und kopieren Sie die eigentliche Regel per *Drag and Drop* in die Unit hinein.

Zusätzlich muss die Unit die gleiche Anzahl an Parametern besitzen, wie die Regel. Selektieren Sie dazu die *Priority Unit* mit der rechten Maustaste und wählen Sie **New Child** ▸ **Parameter** (vgl. Abb. 25). Dabei dürfen die Namen der Parameter durchaus von denen in der Regel abweichen.

Hinweis: In Ausnahmefällen kann eine Unit auch weniger Parameter definieren als die beinhaltete Regel. So z.B., wenn die Regel intern Variablen benötigt (z.B. bei der Berechnung von Attributwerten). Derartige interne Variablen werden in Henshin-Regeln auch als Parameter deklariert und sind von Übergabeparametern nicht zu unterscheiden. Die Parameter der Unit repräsentieren letztlich aber die extern sichtbaren Parameter der Editieroperation, also die Signatur der Editieroperation.

WICHTIG: Es muss immer ein *in*-Parameter mit dem Namen `selectedEObject` existieren, der einen vorhandenen Knoten hinein gibt, der zum Kontext der Editierregel gehört.

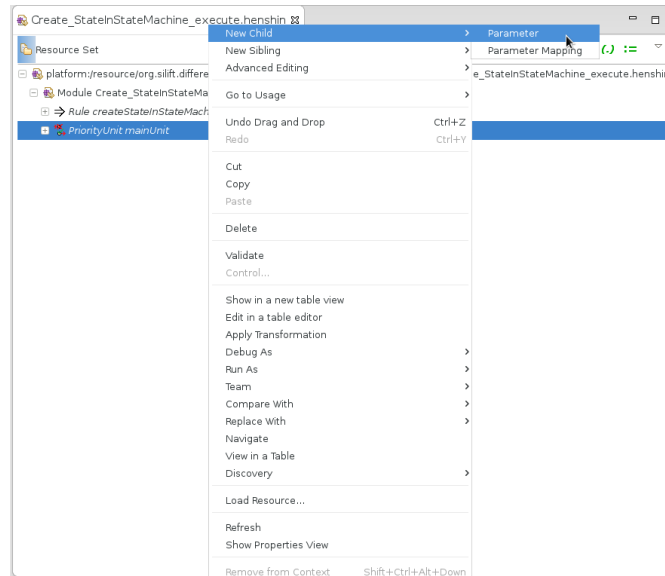


Abbildung 25: Editierregel: Erstellen von Parametern

Nachdem Sie die Parameter erstellt haben müssen diese nun auf die entsprechenden Parameter der Regel “*gemappt*“ werden. Durch das *Mapping* wird die Richtung der Parameter (*in/out*) festgelegt. Klicken Sie wieder mit der rechten Maustaste auf die *Priority Unit* und wählen Sie diesmal **New Child** ▸ **Parameter Mapping**. Jedes *Parameter-Mapping* besteht aus einem **Source**- und **Target**-Parameter, welche unter der *Properties-View* gesetzt werden können.

Wird ein Parameter der *Unit* auf einen Parameter der *Rule* gemappt, so handelt es sich um einen *in-Parameter* der aus dem aktuellen Arbeitsgraphen ein Objekt bzw. Knoten an die Regel übergibt (vgl. Abb. 26). Für `selectedEObject` ist das immer der Fall. Das gleiche gilt für Objekte die gelöscht werden sollen.

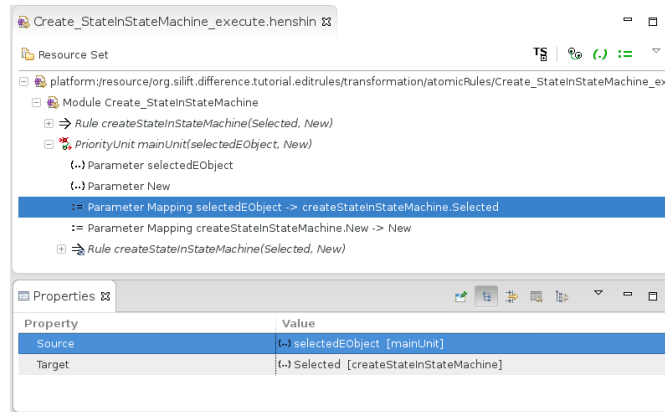


Abbildung 26: Editierregel: *in-Parameter*

Objekte, welche durch die *Rule* erzeugt werden, müssen durch *out-Parameter*, über die *Unit* zurück an den Arbeitsgraphen gegeben werden. D.h. der entsprechende Parameter der *Rule* muss auf einen Parameter der *Unit* gemappt werden (vgl. 27).

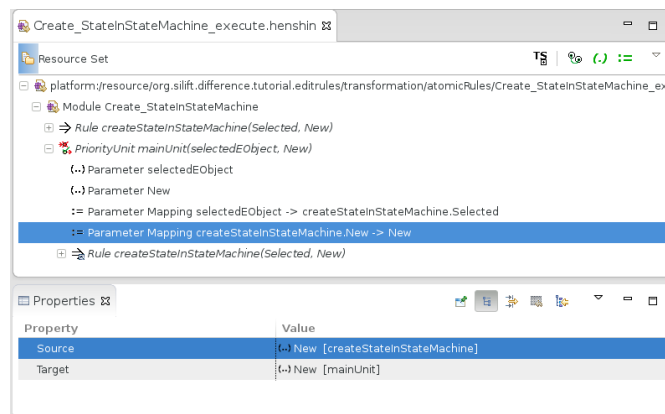


Abbildung 27: Editierregel: *out-Parameter*

Damit ist die Regel `createStateInStateMachine` fertig.

Delete-Regeln

Im vorherigem Abschnitt haben Sie gelernt, worauf Sie beim Erstellen einer *Create-Regel* achten müssen. Neben dem Hinzufügen einzelner Modellelemente, benötigt man jedoch auch Regeln, um Elemente zu verschieben oder aus dem Modell zu entfernen.

Erstellen Sie analog zum vorherigem Abschnitt ein neues Henshin-Modell mit dem Namen `DELETE_StateInStateMachine_execute.henshin` und dem entsprechenden Dia-

gramm. Ersetzen Sie den “Stereotypen“ `create` durch `delete` (vgl. Abb 28).

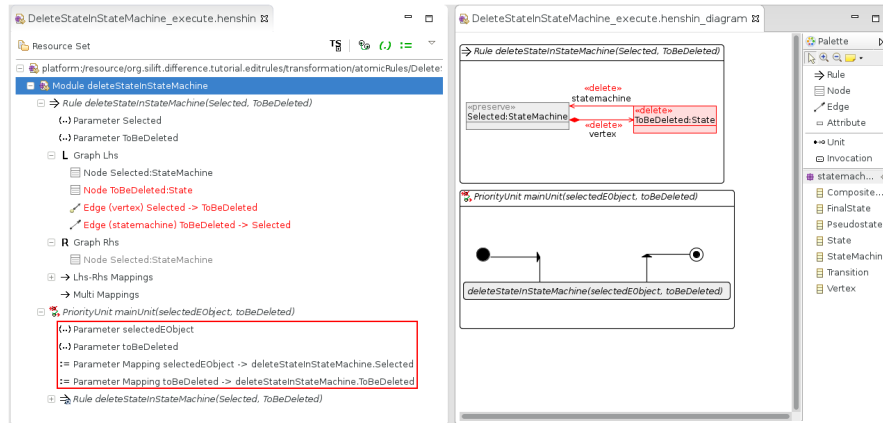


Abbildung 28: Editierregel: `deleteStateInStateMachine`

Zu beachten ist, dass dieser Regel anstatt einem, zwei Objekte übergeben wird. Des Weiteren gibt die Regel auch kein Objekt zurück. Dem entsprechend muss das Parameter Mapping angepasst werden (vgl. Abb. 28).

Ein weiterer wichtiger Punkt beim Löschen von Elementen sind sogenannte *hängende Referenzen* (engl. *dangling edges*).

Abbildung 29 beschreibt die Funktionsweise eines Garagentors in Form eines Zustandsautomaten. Initial ist der Zustandsautomat im Zustand `geschlossen`. Bei Tastendruck wird der Zustand `öffnend` eingenommen usw.

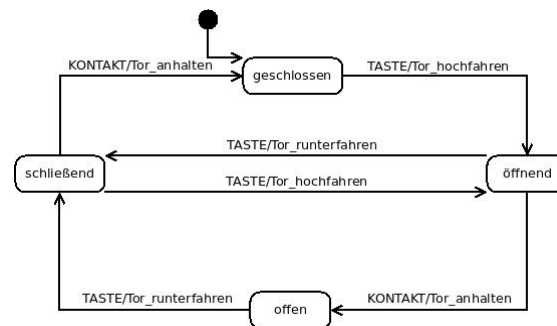


Abbildung 29: Zustandsautomat: Garagentor

Die interne Repräsentation des Modells, mit der Henshin arbeitet, ist in Abbildung 30 zu sehen. Um die Übersicht zu behalten wurden, mit Ausnahme der umrandeten Elemente ,einige Referenzen ausgelassen.

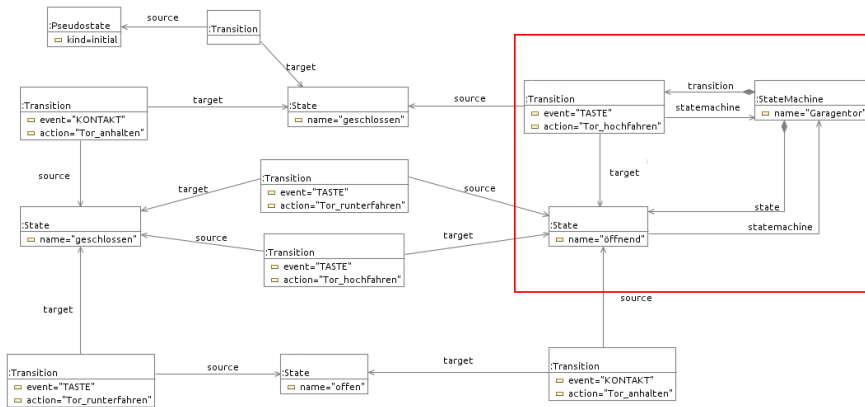


Abbildung 30: Arbeitsgraph

Beim Anwenden der Regel `deleteStateInStateMachine`, um z.B. den Zustand `öffnend` zu löschen, würden hängende Referenzen und somit ein inkonsistentes Modell entstehen. Die Regel löscht nur die Referenzen zwischen *StateMachine* und *State*, alle anderen bleiben bestehen (vgl. Abb. 31).

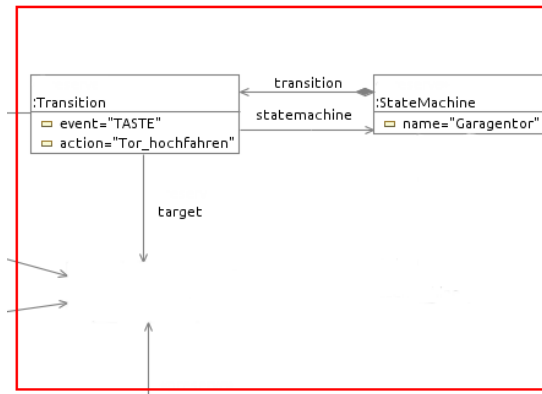


Abbildung 31: hängende Referenzen

Um in so einem Fall das Anwenden einer Regel zu verhindern, muss für die entsprechende Regel die Option `Check Dangling` auf `true` gesetzt sein (vgl. 32).

Hinweis: Die Option `Check Dangling` ist per *default* auf `true` gesetzt und sollte bei der Verwendung von *SiLift* auch nicht geändert werden.

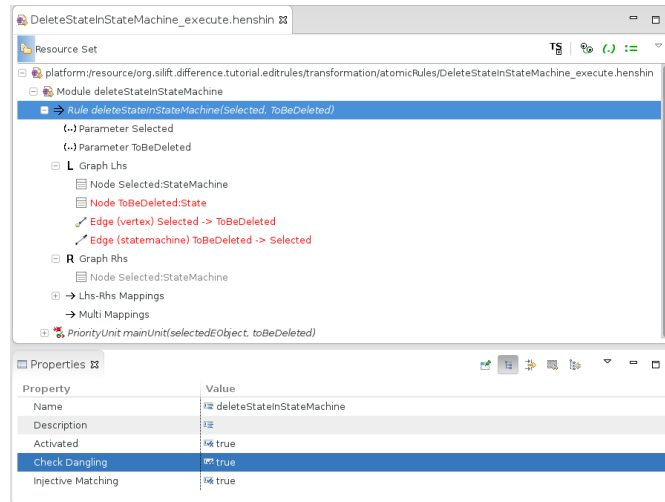


Abbildung 32: hängende Referenzen

Negative Application Conditions

Abbildung 33 zeigt eine Editierregel, die eine neue Transition von einem Startzustand zu einem normalen Zustand erzeugt.

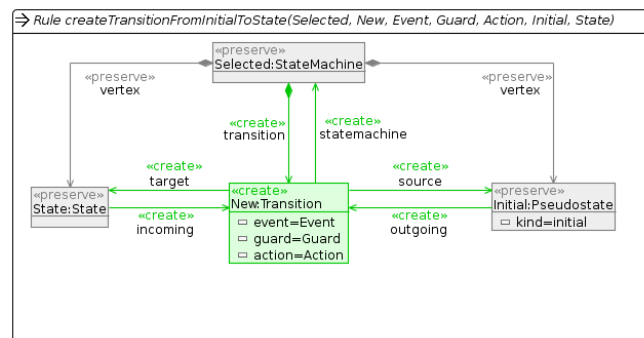


Abbildung 33: Editierregel: createTransitionFromInitialToState

Betrachtet man jetzt nochmal das Metamodell auf Seite 16, so darf ein *Startzustand* maximal eine ausgehende Transition besitzen. Solche Bedingungen können mit *NACs* umgesetzt werden. Dazu modelliert man den unerwünschten Fall als Teilgraph und markiert diesen mit dem “*Stereotyp*“ *forbid* (vgl. Abb. 34).

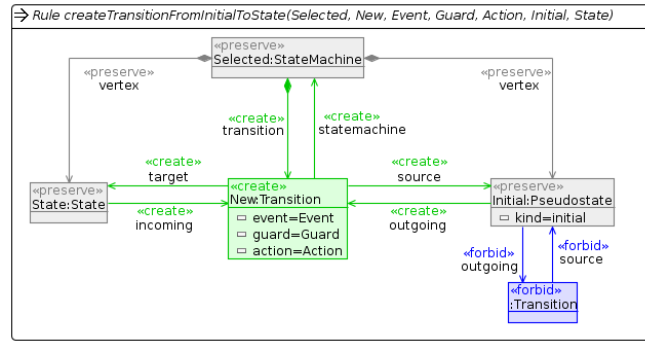


Abbildung 34: Editierregel: `createTransitionFromInitialToState` mit *NAC*

Dieser erzeugt in der *LHS* eine **Application Condition**, welche negiert wird. Die Condition besitzt ein Graph-Objekt, welches den unerwünschten Teilgraphen modelliert. In unserem Fall besteht der Teilgraph aus vier Knoten und vier Kante. Zusätzlich wird noch ein Mapping der Knoten `Initial:Pseudostate`, `Selected:StateMachine` und `State:State` aus der LHS auf den NAC-Graphen benötigt (vgl. Abb. 34).

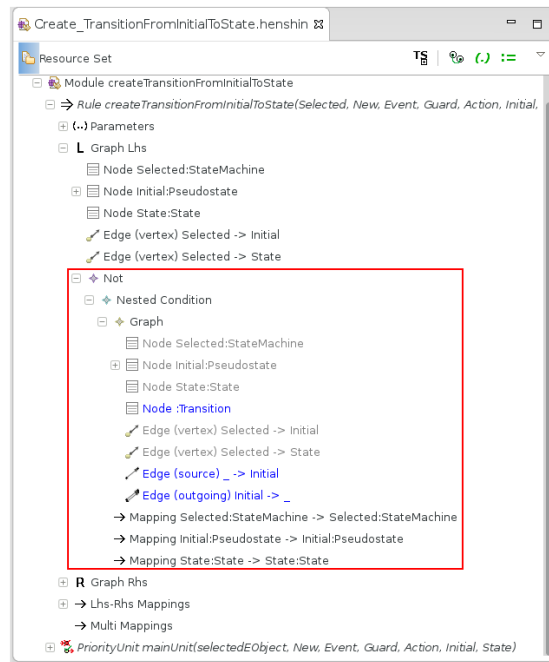


Abbildung 35: Editierregel: `createTransitionFromInitialToState` mit *NAC*

Hinweis: In Henshin lassen sich *NACs* beliebig schachteln und durch boolsche Operatoren wie **AND** und **OR** verknüpfen. *SiLift* unterstützt in der aktuellen Version nur die

Konjunktion von Anwendungsbedingungen.

7 Generieren von Erkennungsregeln

Um einer *low-level* Änderung der technischen Differenz eine bestimmte Editieroperation zuzuordnen werden sogenannte *Erkennungsregeln* benötigt. Dabei handelt es sich ebenfalls um Henshin-Regeln, die sich mit Hilfe des *Recognitionrule Generators* direkt aus den zuvor erstellten Editierregeln ableiten lassen.

7.1 Rulebase Plug-in Projekt

Um ein Rulebase Plug-in Projekt zu erstellen importieren sie am besten ein bestehendes Rulebase Plug-in Projekt (z.B. `org.sidiff.ecore.recognitionrules.atomic`): **File** ▷ **Import** ▷ **Plug-Ins and Fragments** ▷ **Projects with source folders**.

Anschließend passen Sie den Projektnamen und weitere projektspezifische Bezeichner an Ihre Bedürfnisse an. Existierende Rulebases können Sie aus dem Projekt löschen. Verfahren Sie anschließend gemäß Abschnitt 7.2 mit dem eigentlichen Erzeugen der neuen Erkennungsregeln und des sog. Rulebase-Files.

Hinweis: In Kürze wird für das Erstellen eines Rulebase Plug-in Projekts auch ein komfortablerer Wizard zur Verfügung stehen.

7.2 Rulebase File

Des Weiteren kann es sein, dass Sie für eine Domain (hier unser Zustandsautomat) mehrere Regelbasen zur Verfügung stellen möchten. Das ist z.B. dann der Fall, wenn man sich die Editierregeln mittels Generator hat generieren lassen und einige jetzt noch manuell nachgebessert oder ergänzt werden müssen. Um eine neue Regelbasis zu erstellen, gehen Sie wieder auf **File** ▷ **New** ▷ **Other...** und wählen Sie **SiLift** ▷ **Rulebase File**

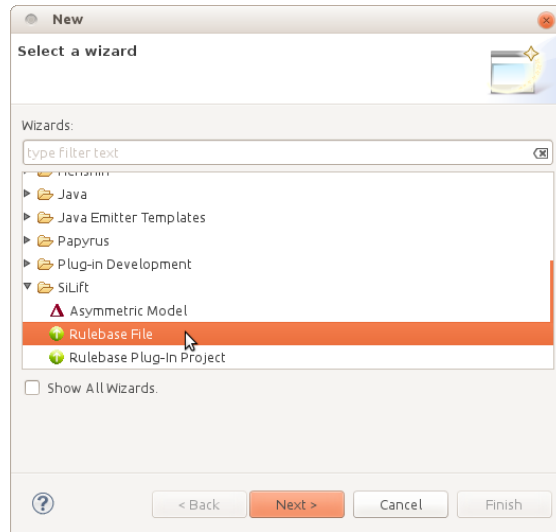


Abbildung 36: Erstellen einer neuen Regelbasis

Im nächsten Schritt wählen Sie das Verzeichnis **rulebase** des bestehenden Projekts für die Erkennungsregeln und geben der Regelbasis einen Namen (vgl. Abb. 37).

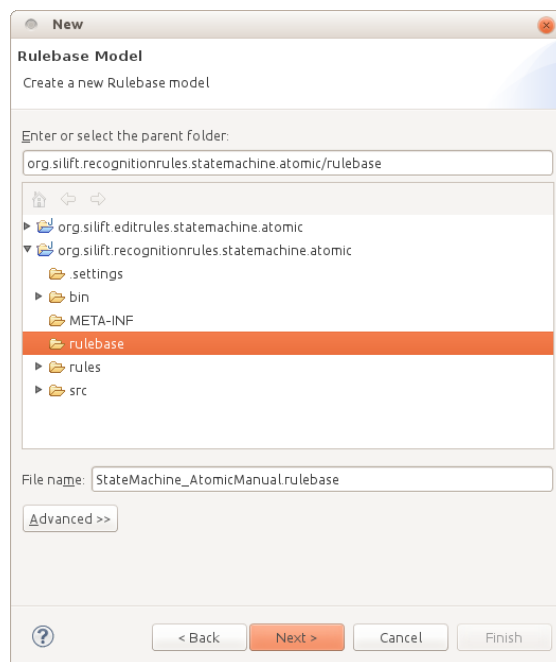


Abbildung 37: Erstellen einer neuen Regelbasis

Jetzt wählen Sie wie bereits zuvor die gewünschten Editierregeln aus und klicken auf

Finish (vgl. Abb. 38).

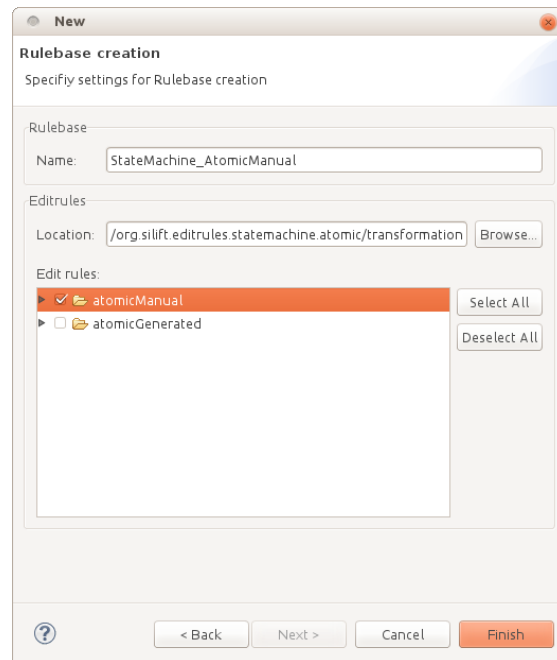


Abbildung 38: Erstellen einer neuen Regelbasis

Damit haben Sie Ihrem Projekt eine neue Regelbasis hinzugefügt (vgl. Abb. 39). Als nächstes muss sich diese Regelbasis noch als Erweiterung (engl. *Extension*) bei dem Plugin registrieren.

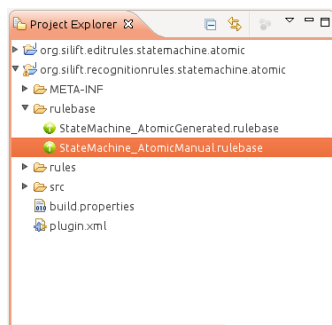


Abbildung 39: Project Explorer: Regelbasen

Öffnen Sie das Verzeichnis `src` über den *Package Explorer*, kopieren Sie die bereits existierende Klasse und nennen diese entsprechend um (vgl. Abb. 40). Danach öffnen Sie die Klasse und passen den Wert der Variablen `RULE_BASE_NAME` entsprechend an.

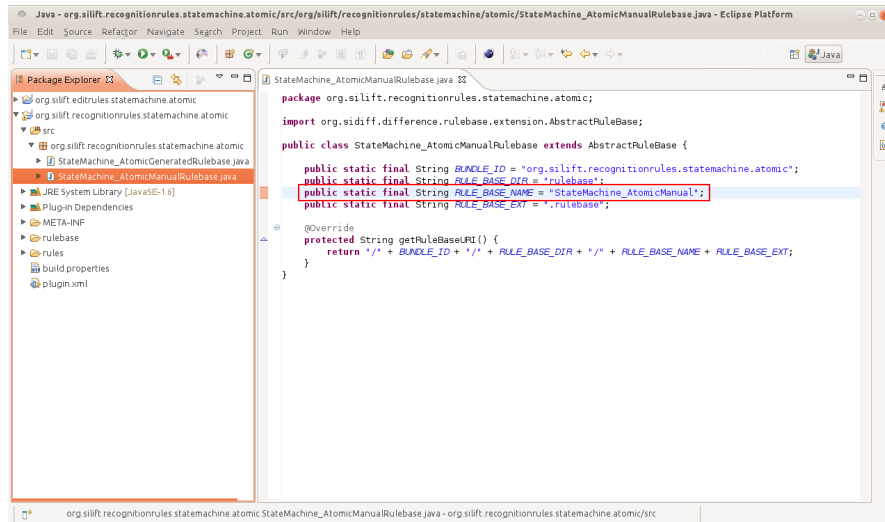


Abbildung 40: Klasse: StateMachine_AtomicManualRulebase

Öffnen Sie die MANIFEST.MF und wählen Sie den Reiter Extensions aus (vgl. Abb. 41).

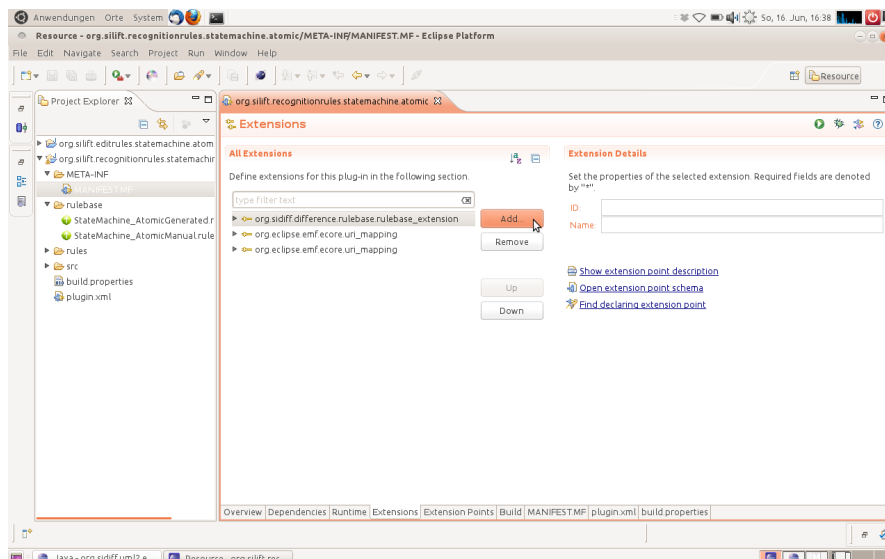


Abbildung 41: Manifest.MF: Extensions

Klicken Sie auf Add... und selektieren Sie den *Extension Point* org.sidiff.difference.rulebase.rulebase_extension (vgl. Abb. 42). Klicken Sie auf Finish.



Abbildung 42: Extension Point: Rule Base

Wechseln Sie danach in den Reiter `plugin.xml` und fügen dem eben erstellen Extension Point die entsprechende URI der Erweiterung bei (vgl. Abb. 43).

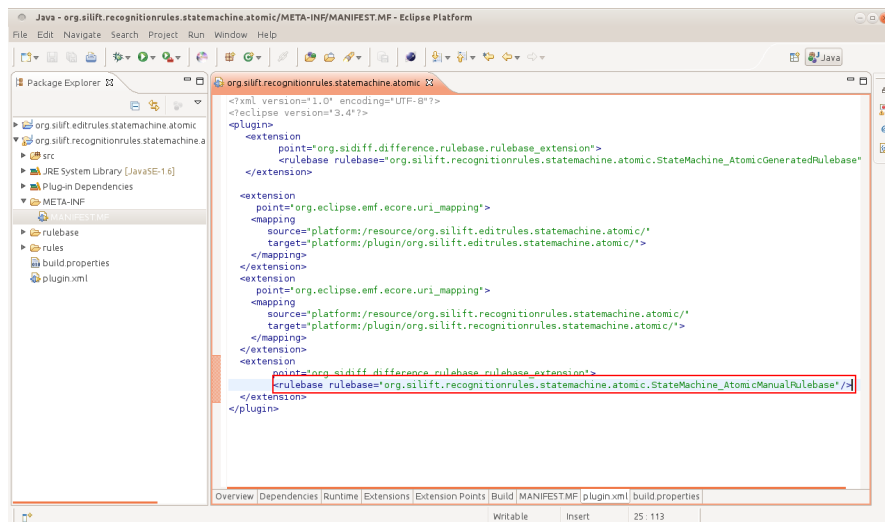


Abbildung 43: Manifest.MF: `plugin.xml`

7.3 Der Rulebase-Manager

Generierte Erkennungsregeln können mit Hilfe des *Rulebase Manager* verwaltet werden (vgl. Abb. 44).

A	B	ER-Type C	RR-Type D	Priority E	Refinement F	P.D. G	Version H
<input checked="" type="checkbox"/>	setVertexName	Priority	Rule	1	0	21	0.0.1
<input checked="" type="checkbox"/>	setTransitionGuard	Priority	Rule	1	0	4	0.0.1
<input checked="" type="checkbox"/>	setTransitionEvent	Priority	Rule	1	0	4	0.0.1
<input checked="" type="checkbox"/>	setTransitionAction	Priority	Rule	1	0	4	0.0.1
<input checked="" type="checkbox"/>	setStateName	Priority	Rule	1	1	11	0.0.1
<input checked="" type="checkbox"/>	setStateMachineName	Priority	Rule	1	0	1	0.0.1
<input checked="" type="checkbox"/>	setPseudostateName	Priority	Rule	1	1	8	0.0.1
<input checked="" type="checkbox"/>	setPseudostateKind	Priority	Rule	1	1	8	0.0.1
<input checked="" type="checkbox"/>	setFinalStateName	Priority	Rule	1	2	9	0.0.1
<input checked="" type="checkbox"/>	setCompositeStateName	Priority	Rule	1	1	7	0.0.1
<input checked="" type="checkbox"/>	moveVertexRefSubstateToCompositeState	Priority	Rule	1	2	26	0.0.1
<input checked="" type="checkbox"/>	moveVertexRefStateToStateMachine	Priority	Rule	1	0	20	0.0.1
<input checked="" type="checkbox"/>	moveTransitionRefTransitionToStateMachine	Priority	Rule	1	0	3	0.0.1
<input checked="" type="checkbox"/>	moveStateRefSubstateToCompositeState	Priority	Rule	1	3	16	0.0.1
<input checked="" type="checkbox"/>	moveStateRefStateToStateMachine	Priority	Rule	1	1	10	0.0.1
<input checked="" type="checkbox"/>	movePseudostateRefSubstateToCompositeState	Priority	Rule	1	3	12	0.0.1
<input checked="" type="checkbox"/>	movePseudostateRefStateToStateMachine	Priority	Rule	1	1	6	0.0.1
<input checked="" type="checkbox"/>	moveFinalStateRefSubstateToCompositeState	Priority	Rule	1	4	14	0.0.1
<input checked="" type="checkbox"/>	moveFinalStateRefStateToStateMachine	Priority	Rule	1	2	8	0.0.1
<input checked="" type="checkbox"/>	moveCompositeStateRefSubstateToCompositeState	Priority	Rule	1	3	12	0.0.1
<input checked="" type="checkbox"/>	moveCompositeStateRefStateToStateMachine	Priority	Rule	1	1	6	0.0.1
<input checked="" type="checkbox"/>	deleteTransitionInStateMachine	Priority	Rule	1	0	28	0.0.1
<input checked="" type="checkbox"/>	deleteStateInStateMachine	Priority	Rule	1	1	13	0.0.1
<input checked="" type="checkbox"/>	deleteStateInCompositeState	Priority	Rule	1	2	17	0.0.1
<input checked="" type="checkbox"/>	deletePseudostateInStateMachine	Priority	Rule	1	1	11	0.0.1
<input checked="" type="checkbox"/>	deletePseudostateInCompositeState	Priority	Rule	1	1	11	0.0.1

Abbildung 44: Erstellen eines *Rulebase Manager*

- (A) Durch Klicken auf das Häkchen können einzelne Erkennungsregeln für die *Recognition-Engine* aktiviert (grün) bzw. deaktiviert (grau) werden.
- (B) Repräsentiert den Verwaltungsname der Editier-, bzw. Erkennungsregel. Dieser kann durch den Rulebase Manager editiert werden, wird aber nur zur Anzeige in der GUI verwendet.
- (C) Henshin Typ der *mainUnit* der Editierregel (**Independent**, **Priority**, **Sequential** oder **Amalgamation Unit**).
- (D) Henshin Typ der Erkennungsregel mainUnit.
- (E) Priorität der Erkennungsregel: Gerade unter zusätzlicher Verwendung komplexer Editierregeln kann es vorkommen, dass zwei *Semantic Change Sets* (vgl. 6) die gleichen *low-level*-Änderungen beinhalten. Für den Fall kann man einer Regel eine höhere Priorität zuordnen.
- (F) *Refinement-Level*: Für den Fall, dass auch die Prioritäten zweier identischer *Semantic Change Sets* gleich sind, versucht *SiLift* anhand der Anzahl der Supertypen die “speziellere” Regel zu bestimmen. D.h. je mehr Supertypen die Knoten der Regel besitzen, desto spezieller ist diese.

- (G) *Potential Dependencies*: Anzahl der potentiellen Abhängigkeiten zu anderen Editieroperationen. Das sequentielle Ausführen mehrere Editieroperationen ist nicht kommutativ, d.h. es können zwischen den jeweiligen Editieroperationen Abhängigkeiten existieren, die beim generieren eines Patches berücksichtigt werden müssen.
- (H) Version des verwendeten *Recognitionrule-Generators*.

I.d.R. wächst eine Regelbasis mit der Zeit. Es ist so gut wie unmöglich alle möglichen Editieroperationen im Vorfeld aufzudecken und zu implementieren. Um Ihrer bestehenden Regelbasis neue Regeln hinzuzufügen klicken Sie auf **Generate new recognition rules** (vgl. Abb. 45) und wählen die entsprechenden Editierregeln aus. Die abgeleiteten Erkennungsregeln werden nun der bestehenden Regelbasis hinzugefügt.

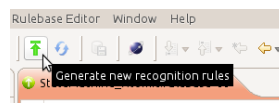


Abbildung 45: Erkennungsregeln einer bestehenden *Rulebase* hinzufügen

7.4 Erkennungsregeln deployen und nutzen

Um die Erkennungsregeln zu testen, gibt es zwei Möglichkeiten:

1. Eclipse Application: Öffnen Sie die **MANIFEST.MF** des Projekts der Erkennungsregeln und starten sie über das *Launch Icon* (vgl. Abb. 46) eine zweite Eclipse-Instanz. Innerhalb dieser Instanz sind alle Projekte aus Ihrem Workspace registriert und können verwendet werden.

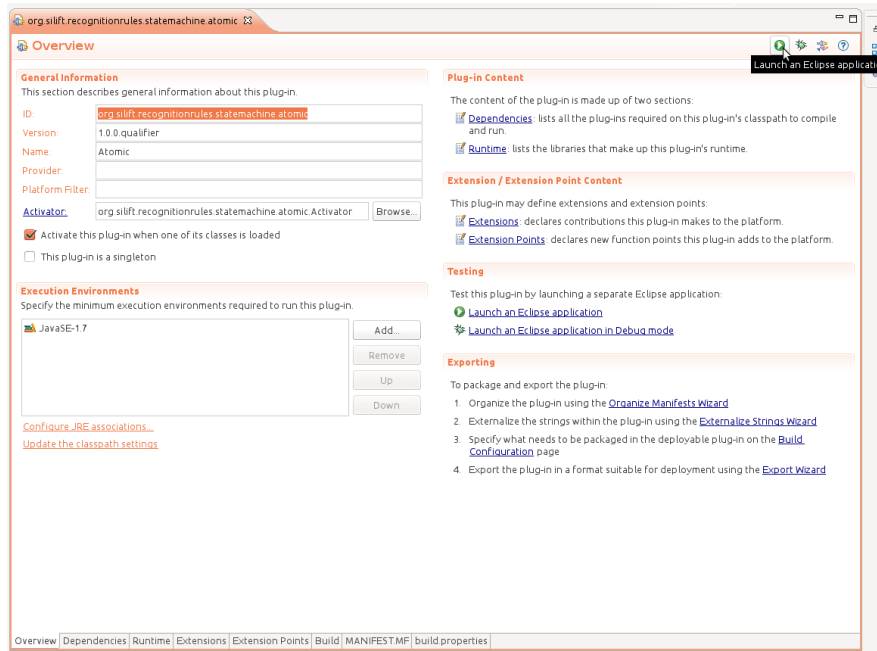


Abbildung 46: Run Eclipse Application

2. Deployable Plugins and Fragments: analog zu Abschnitt 4.

Wenn Sie Ihre Regeln erstmal nur testen möchten, ist die erste Variante zu bevorzugen. Sofern Sie die zweite Variante nutzen und ggf. mit Hilfe des *Rulebase-Managers* an den Erkennungsregeln etwas verändern möchten, müssen Sie die installierten Plugins zuerst deinstallieren.

Eine umfassende Einführung in die Nutzung von SiLift als Differenzwerkzeug finden Sie im **SiLift - Benutzerhandbuch für Endanwender**.

ENDE

8 Links und weitere Informationen

- EMF-Compare: <http://www.eclipse.org/emf/compare>
- EMF-Henshin: <http://www.eclipse.org/henshin/>
- SERGe: <http://pi.informatik.uni-siegen.de/mrindt/SERGe.php>
- SiDiff: <http://pi.informatik.uni-siegen.de/Projekte/sidiff/>
- SiLift: <http://pi.informatik.uni-siegen.de/Projekte/SiLift/>