Graphical Modeling Framework

Bernd Kolb, bernd@applied-abstractions.com www.applied-abstractions.com

Sven Efftinge, sven@applied-abstractions.com www.applied-abstractions.com

Markus Voelter, markus@applied-abstractions.com www.applied-abstractions.com

Arno Haase, arno@applied-abstractions.com www.applied-abstractions.com

Inhalt

Einführung

Seit nunmehr knapp einem Jahr gibt es unter Eclipse.org das Graphical Modeling Framework, kurz GMF. Seit einigen Wochen hat GMF innerhalb des Callisto-Release-Trains seine Version 1.0 veröffentlicht. Dies wollen wir zum Anlass nehmen uns dieses Framework genauer anzusehen.

Bei GMF handelt es sich um ein Framework welches sich zum Ziel gesetzt hat voll funktionsfähige graphische, Eclipse-basierte Editoren für selbst zu definierende EMF-basierte Metamodelle zu generieren. Mögliche Beispiele für solche Editoren wären unter anderem UML-Diagramme, Fluss-Diagramme, oder Editoren zur Abbildung von Geschäftsprozessen. Typischerweise wird GMF aber dazu genutzt, eine auf den zu beschreibenden Bereich exakt zugeschnittene Modellierungssprache zu entwerfen. Eine solche auf die Bedürfnisse der entsprechenden Domäne angepasste Sprache nennt man auch Domänen-spezifische Sprache (Domain-Specific Language – DSL).

Das GMF Projekt kann in zwei Teile unterteilt werden: Einen Tooling-Teil und einen Runtime-Teil. Beim Tooling handelt es sich um Editoren, die zur Beschreibung eines Editors genutzt werden, sowie einen Generator, welcher aus der Editor-Definition einen lauffähigen Editor erstellt. Die Runtime stellt Funktionalitäten wie zum Beispiel

Oberklassen für Figuren, Layout-Klassen und -Algorithmen oder eine Anbindung an eine OCL-Engine zur Validierung von Modellen zur Verfügung.

GMF benutzt im Wesentlichen zwei weitere Eclipse-basierte Frameworks. Zum einen das Eclipse-Modeling-Framework (EMF), zum anderen das Graphical-Editor-Framework (GEF). Das EMF stellt ein sog. Meta-Metamodell namens Ecore zur Verfügung. Mit einem Meta-Metamodel kann man Metamodelle erstellen, aus deren Elementen wiederum die Modelle bestehen. Bei der OMG ist die MOF (Meta Object Facility) das Meta-Metamodel und die UML2 stellt z.B. ein entsprechendes Metamodell dar. GEF hingegen ist ein Framework mit dessen Hilfe graphische Editoren erstellt werden können. GEF folgt dabei dem Model-View-Controller Pattern. Diese Editoren können dazu genutzt werden einen Objekt-Graphen graphisch darzustellen und zu editieren, bzw. zu erweitern. Der Aufbau eines solchen Editors ist immer identisch: Zum einen gibt es Modellobjekte. Einem Modellobjekt ist eine graphische Repräsentation zugeordnet. In der GEF Terminologie werden diese Repräsentationen als „Figures“ bezeichnet. Diese Figuren können auf einem „Canvas“, also einem Editor mittels einer „Palette“ angelegt werden. Dieser Editor selbst repräsentiert selbst ebenfalls ein Modellobjekt. Die Eigenschaften von Modellobjekten, also die Attribute ihrer Metaklassen, können mit Hilfe des Properties-Views verändert werden (siehe Abbildung 1).

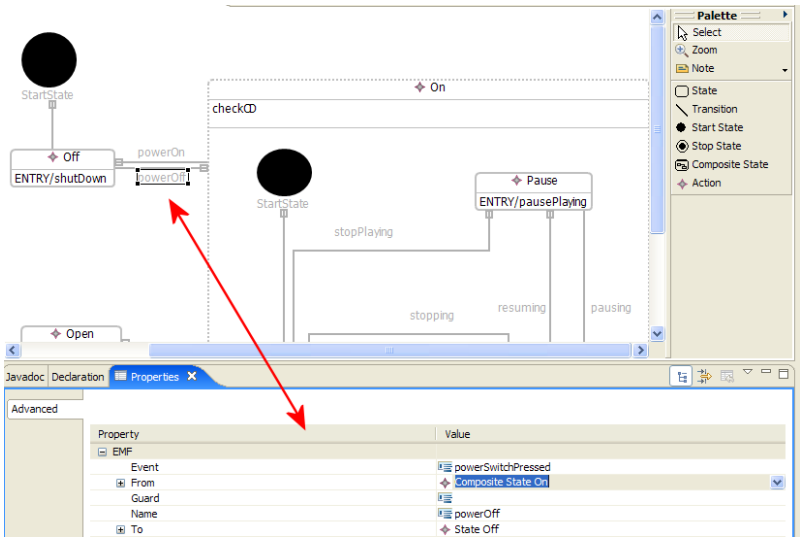


Abb. 1: Ein von GMF generierter GEF-Editor

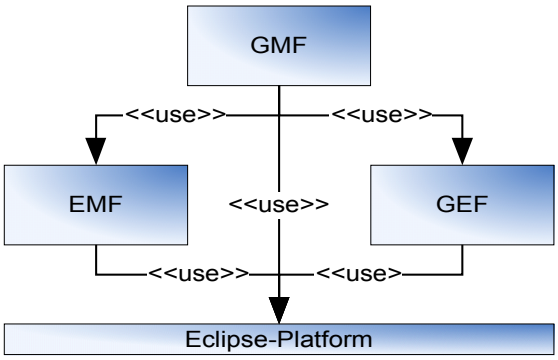
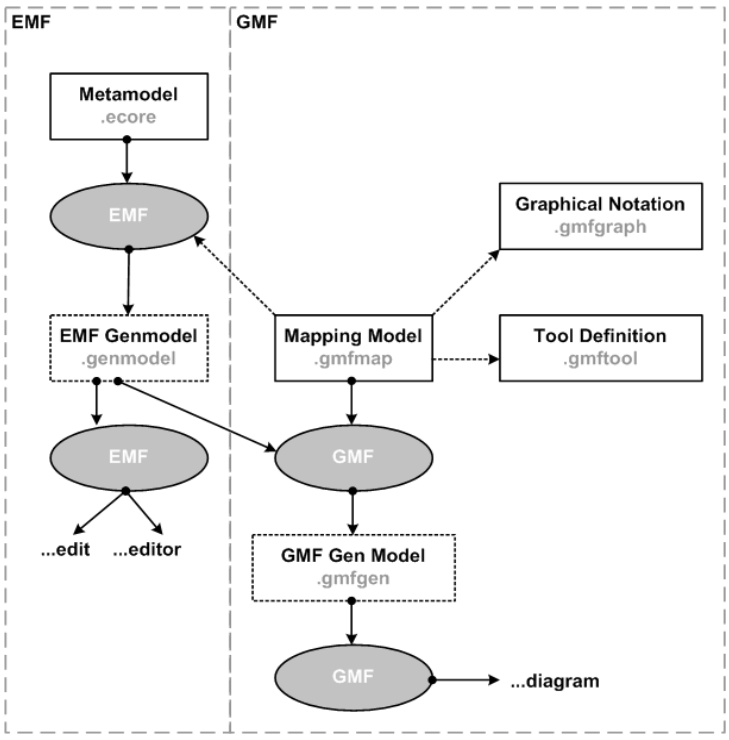


Abb. 2: Beziehung zwischen den Frameworks

Die Aufgabe von GMF ist es nun ein mit EMF-Mitteln beschriebenes, Domänen-spezifisches Metamodell auf einen graphischen Editor abzubilden.

Hierfür nutzt GMF drei Modelle: Eines zur Beschreibung der Figuren, ein weiteres zur Beschreibung der Palette und ein drittes um das Metamodell, die graphische Definition sowie die Palettenbeschreibung zu verbinden. Dieses dritte Modell reichert GMF in einem weiteren Schritt an um daraus dann schließlich den graphischen Editor zu generieren. Das Diagramm in Abbildung 3 veranschaulicht die beschriebenen Abhängigkeiten.



Ein Beispiel – Editor für einen Zustandsautomaten

Um mit Hilfe von GMF einen Editor zu erstellen sind verschiedene Schritte nötig. Im weiteren Verlauf dieses Artikels wollen wir diese Schritte anhand eines graphischen Editors zur Modellierung von hierarchischen Zustandsautomaten beschreiben.

Das Metamodell für den Editor ist in Abbildung 4 dargestellt:

- Eine *Statemachine* besteht aus mehreren *States*.
- Ein *State* kann entweder ein *StartState*, ein *StopState*, ein *CompositeState* oder ein „normaler" *State* sein.
- Eine *Transition* verbindet zwei *States*. *States* kennen ihre eingehenden und ausgehenden Transitionen.
- Ein *Compositestate* kann seinerseits wieder *States* enthalten.
- Ein *State* hat *Actions*. Eine Aktion kann entweder eine Eingangs- oder Ausgangsaktion sein.

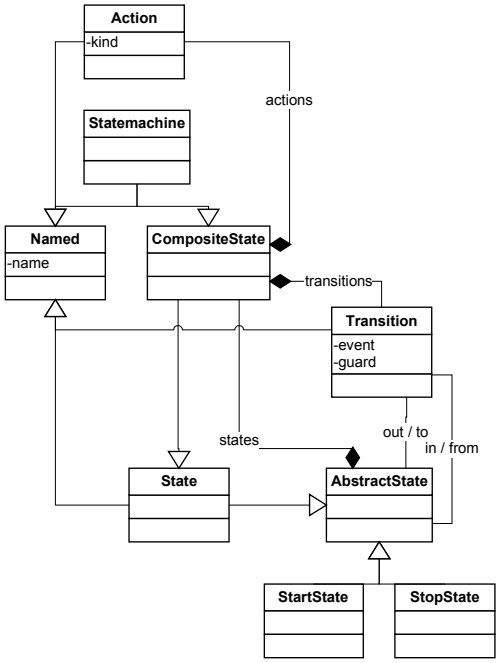
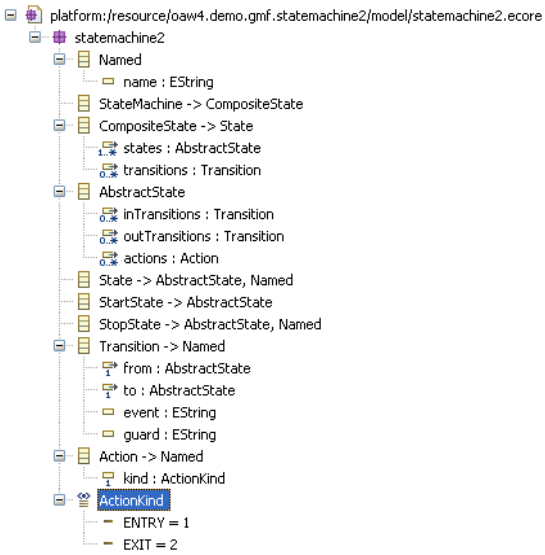


Abb. 4: Statemachine Metamodell

Bevor wir beginnen können, müssen wir das Graphical Modeling Framework und dessen Abhängigkeiten installieren. Hierzu starten wir Eclipse (in der Version 3.2) und wählen im Update Manager die Callisto-Update-Site. In der Kategorie „Models and Model Development" ist GMF zu finden. Wir wählen es aus und wählen mittels der Schaltfläche

auf der rechten Seite alle Abhängigkeiten aus. Nach erfolgreicher Installation und einem Neustart von Eclipse kann es los gehen.

Zur Definition des EMF-Modells nutzen wir den mit EMF mitgelieferten Baum-basierten Editor. (Wie genau ein solches EMF-Modell definiert werden kann ist in dem online verfügbaren Artikel[1] auf Eclipse.org beschrieben). Abbildung 5 zeigt das fertige Metamodel, dargestellt im EMF Editor. Nachdem das Modell definiert wurde lassen wir uns durch die EMF-Wizards das genmodel und daraus die Implementierungsklassen sowie das Edit-PlugIn erstellen. Diese Implementierungsklassen sowie das Edit-PlugIn werden später vom Editor benutzt um das Modell zu bearbeiten. Die Erfahrung zeigt, dass es sinnvoll ist die ecore-Datei (Die Datei die das Metamodell enthält), das genmodel (eine Anreicherung der ecore-Datei, welche EMF zur Generierung verwendet), sowie alle GMF-Modelle, welche wir im Folgenden erstellen, mit demselben Namen zu versehen, da es sonst zu Problemen bei der Generierung des Editors kommen kann.



Als nächstes widmen wir uns den graphischen Repräsentationen. Hierzu legen wir eine neue Grafik-Definitionsdatei an. GMF hat auch hier einige Wizards, die uns das Anlegen einer solchen Definitionsdatei erleichtern sollen, diese befinden sich aber derzeit noch in der „Erprobung“ und werden somit von uns hier nicht genutzt. Wir wählen also im New-File-Wizard von Eclipse die Kategorie „Graphical Modelling Framework“ und darunter nicht das „GMFGraph Simple Model“ sondern benutzen das unter „default“ befindliche „GMFGraph Model“. Da EMF-Modelle immer eine baumartige Struktur haben, müssen wir den Modellelementtyp des obersten Knotens angeben. Für die Definition der Figuren ist dies der „Canvas“. Im neu geöffneten Editor öffnen wir den obersten Knoten welcher

[1]http://www.eclipse.org/articles/Article-Using%20EMF/using-emf.html

die Datei repräsentiert. Darunter befindet sich ein Element vom Typ „Canvas“ wie wir es eben erstellt haben. Mit Hilfe des Kontextmenüs öffnen wir den Properties-View. In diesem Fenster werden die Eigenschaften des gerade selektierten Elements angezeigt und können hier auch verändert werden. Ebenfalls mit Hilfe des Kontextmenüs legen wir eine neue „Figure Gallery“ unterhalb des „Canvas“ an. Die „Figure Gallery“ wird später die Beschreibung aller unserer Figuren, also graphischen Repräsentationen für States, CompositeStates, Actions, Transitions usw. beinhalten (vergl. Abbildung 6).

Ein State entsteht

Beginnen wir zunächst mit einem einfachen State: Hierzu legen wir unter der Figure Gallery ein neues Element vom Typ „Rounded Rectangle“ an. Mit Hilfe des Properties-Views lassen sich auch hier wieder verschiedene Einstellungen vornehmen. So können wir hierüber der neu erstellten Figur den Namen „StateRectangle“ geben.

GMF kennt eine Reihe von Basisfiguren, so zum Beispiel ein Rechteck, ein Rechteck mit runden Ecken, eine Ellipse oder ein Polygon. Diese Figuren können mit Hilfe von verschieden Layout-Klassen kombiniert werden. Auf diese Art und Weise können recht ansehnliche Figuren auch ohne tiefere Kenntnisse des von GEF zur Darstellung benutzten Grafikframeworks draw2D erstellt werden. Sollte diese Funktionalität jedoch nicht ausreichen, kann der absolute Name einer Klasse angegeben werden, die dann als Figur verwendet wird.

Der GMF-Generator wird uns mit Hilfe dieser Beschreibung später eine Figur mit dem Namen `StateRectangle` generieren, welche von `org.eclipse.draw2d.RoundedRectangle` abgeleitet wird. Damit wir diese Figur später aus unserem Mapping referenzieren können legen wir parallel zu unserer Figure-Gallery ein neues Element vom Typ Node an. Wie der Name schon vermuten lässt wird mit Hilfe eines Nodes ein Diagramm-Knoten gekennzeichnet. Ein Diagramm-Knoten hat selbst ebenfalls wieder einen Namen und zeigt auf eine Figur. Dadurch ist es möglich eine Figur für mehr als einen Diagramm-Knoten zu verwenden. Wir setzen den Namen auf „StateNode“

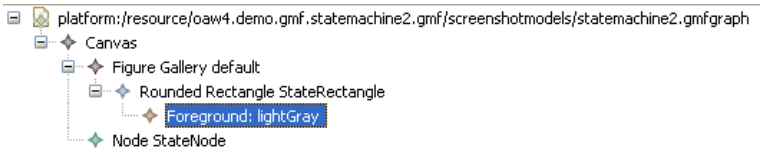


Abb. 6: Die GMF-Graph Beschreibung

Als nächstes wollen wir die Toolingdefinition für einen State erstellen. Hierzu wählen wir im New-File-Wizard erneut die Kategorie des Graphical Modelling Frameworks und darin

das unter default befindliche GMF-Tool. Als Basiselement wählen wir diesmal die „Tool Registry“. Mit Hilfe der Toolregistry wird es in Zukunft möglich sein nicht nur eine Palette sondern auch das Kontextmenü oder die Editor-Toolbar zu beschreiben. Doch kümmern wir uns weiter um das Anlegen eines States. Unter der Toolregistry legen wir ein neues Element vom Typ „Palette“ an und erzeugen hierunter ein neues „Creation Tool“. Über den Properties-View versehen wir es mit dem Namen „State“. Auf Wunsch kann dem Creation Tool noch eine Beschreibung und ein Bild hinzugefügt werden. Damit wäre die Definition der Palette fürs erste abgeschlossen.

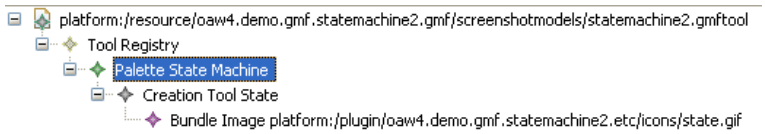


Abb. 7: Die GMF-Tool-Beschreibung

Stell die Verbindung her

Als nächste und damit letzte GMF-Datei ist das Mapping an der Reihe. Wir erinnern uns: Das Mapping wird dazu genutzt unser Statemachine-Metamodell mit den graphischen Definitionen und der eben erstellten Toolingdefinition zusammen zu bringen (siehe Abbildung 3). Nachdem wir die Mapping-Datei mit einem „Mapping“ als obersten Knoten erstellt haben, müssen wir zunächst dafür sorgen, dass dem Mapping-Modell unsere anderen Modelle bekannt sind. Hierzu wählen wir die Aktion „Load Resource“ im Kontextmenü unseres Editors. Aus dem Workspace wählen wir die Datei „statemachine2.ecore“, „statemachine2.gmfGraph“ sowie „statemachine2.gmfTool“. Hierbei ist es möglich auch mehrere Figurdefinitionen zu laden, um dadurch z.B. Figurdefinitionen anderer wiederzuverwenden.

Nachdem unserem Mapping nun die Referenzen auf unsere anderen Modelle bekannt sind können wir mit der eigentlichen Abbildung beginnen. Zunächst soll die Diagrammfläche, oder auch „Canvas“ genannt, ein Modellobjekt vom Typ „Statemachine“ repräsentieren. Um dies zu erreichen, legen wir unterhalb des Mappings ein neues Element vom Typ Canvas an. Diesem neuen Element müssen wir nun einige Eigenschaften setzten. Zunächst wählen wir als Domain Model das Root-Element unseres Metamodells aus. In diesem Fall ist dies ein EPackage mit dem Namen statemachine2. Damit teilen wir GMF mit, dass alle Elemente, die auf diesem Canvas erstellt werden sollen, innerhalb des Domänen Modells liegen. Als nächstes spezifizieren wir das Element das die Diagrammfläche repräsentiert. Wie erwähnt handelt es sich hierbei um eine EClass mit dem Namen „StateMachine“. In der Kategorie „misc“ wählen wir die Palette aus unserer Toolingdefinition. Des weiteren setzen wir als visuelle Repräsentation das Attribut „Diagram Canvas“ auf den Canvas aus unser graphischen Definition.

Als nächstes wollen wir das Diagramm mit dem State bekannt machen. Hierzu legen wir unterhalb des Mappings eine neue TopNodeReference an. Wann immer auf dem Diagramm ein neuer State erstellt wird, soll dieser in die Liste aller States einer Statemachine aufgenommen werden. Hierfür existiert im Metamodell die Composite-Beziehung „states“ zwischen einem CompositeState und einem AbstractState. Da eine StateMachine von CompositeState und ein State von AbstractState erbt, besteht die Beziehung auch für diese beiden Elemente. Wir setzen also das Attribut „Containment Feature“ der eben erstellen TopNodeReference auf die EReference „states“, bevor wir unterhalb der Referenz ein neues NodeMapping anlegen. Dieses NodeMapping soll ein Element vom Typ State mit dem CreationTool „State“ aus der Toolingdefinition, sowie den Knoten „StateNode“ aus der Figurendefiniton zusammenbringen.

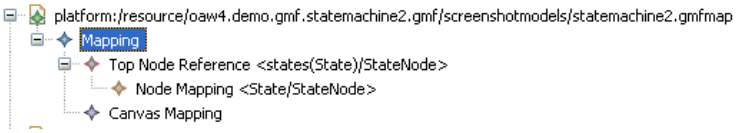


Abb. 8: Das GMF-Mapping

Nun ist es soweit: Wir sind bereit für die Generierung des Editors. Über das Kontextmenü des Mappingeditors lassen wir uns das Generator-Modell mit dem Namen „statemachine2.gmfGen“ erstellen. In dieser Datei werden unsere drei GMF-Dateien zusammengebracht und so aufbereitet, dass sie vom Generator einfach zu verarbeiten sind. Im Kontextmenü der gmfGen-Datei wählen wir die Aktion „Generate Diagram Code“ was uns ein neues Eclipse-PlugIn-Projekt beschert. Dieses PlugIn starten wir in der Runtime-Workbench von Eclipse. Nachdem wir ein neues Projekt angelegt haben können wir aus der Beispielsektion des New-File-Wizards ein neues Statemachine-Diagramm erstellen lassen. Im sich dadurch öffnenden Editor können wir über die Palette neue States anlegen. Parallel zu unserer statemachine2_diagram-Datei wurde eine weitere Datei mit der Endung „statemachine2“ erstellt. Öffnen wir diese sehen wir das über den graphischen Editor erstellte Modell im XMI-Format.

Ein Label für den State!

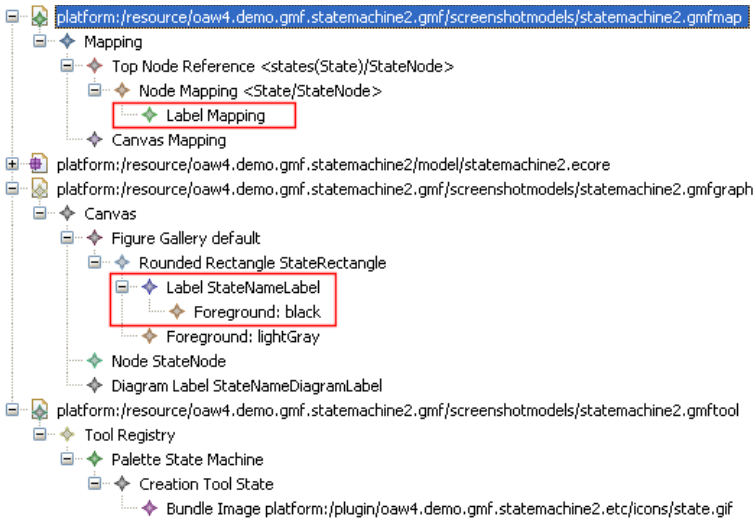
Bisher wird unser State nur durch ein abgerundetes Rechteck repräsentiert. Bei mehreren States ist es jedoch mühsam immer im Properties-View nach dem Namen des gerade selektierten States Ausschau zu halten. Deshalb wollen wir als nächstes dem Rechteck des States ein Label hinzufügen, das den Namen des States im Diagram zeigt. Dazu öffnen wir unsere gmfGraph-Datei und fügen dem unterhalb der Figure-Gallery befindlichen StateRectangle ein neues Kind vom Typ Label hinzu. Dieses neue Element wird mit dem Namen „StateNameLabel“ versehen. Genauso wie im letzten Artikel ein Knoten für einen

State angelegt wurde, muß nun ein Diagram Label für das Label angelegt werden. Als Figur weisen wir ihm das eben erstellte Label zu und nennen es „StateNameDiagramLabel“. Damit wäre die Erweiterung des graphischen Mappings abgeschlossen.

Als nächstes sollte das Label in der der gmfMap-Datei eingebunden werden. Das Label soll den Namen eines States darstellen. Dazu wird unterhalb des Node Mappings des States, ein neues Label Mapping angelegt. Als Diagram Label wählen wir das „StateNameDiagramLabel“. Zusätzlich muss aus der Liste der Features das Feature mit dem Namen „Name“ ausgewählt werden. GMF bietet die Möglichkeit innerhalb eines Labels mehrere Attribute zu verändern. Wollen wir die Attribute A und B im Format „A / B“ darstellen, so müssen A und B in die Liste der Features aufgenommen, sowie als View- und Edit Pattern {0} / {1} angegeben werden.

Eine Änderung des gmfTools ist nicht notwendig, da das Label automatisch mit einem State erzeugt wird.

Abbildung 2: Das geänderte GMF-Graph und Mappingmodel. Änderungen wurden rot umrahmt



Nachdem das gmfGen-Model neu erzeugt wurde, kann der Editor generiert und in der Eclipse Runtime-Workbench gestartet werden. States können nun direkt im Diagramm mit einem Namen versehen werden.

Transitions

Als nächstes soll die Möglichkeit geschaffen werden zwei States miteinander zuverbinden. Im Metamodell wird eine solche Linie durch die Metaklasse „Transition“ abgebildet.

Beginnen wir auch hier wieder mit dem graphischen Mapping: In der Figure Gallery wird ein neues Element „Polyline Connection „ mit dem Namen „TransitionFigure“ angelegt. Auch hier kann wieder ein Label als Kind-Element angelegt werden. Dieses Label soll später den Namen, das Event sowie den Guard einer Transition anzeigen und editieren können. Als weiteres Kind-Element wählen wir eine Polyline Decoration. Dekorationen sind, soweit nicht anders spezifiziert, nicht ausgefüllte Pfeile. Soll die Dekoration eine andere Form haben kann diese mit Hilfe von Points als Kinder der Dekoration beschrieben werden. Um die Dekoration beispielsweise auf das Ende der Transition-Linie anzuwenden, wählen wir die angelegte Dekoration als Target Decoration in der Polyline Connection. Zum Abschluss muss auch hier das graphische Element auf ein vom Mapping zugreifbares Element gemapped werden. Im Falle der PolylineConnection ist dies die Connection, die wir unterhalb des Canvas anlegen. Wir versehen sie mit dem Namen „TransitionConnection“ und wählen die entsprechende PolylineConnection als Figur. Das selbe Vorgehen wählen wir für das Label und nennen es „TransitionDiagramLabel“

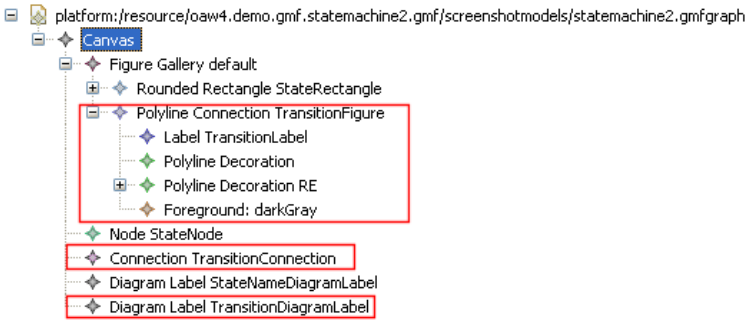


Abb. 3: Update des graphischen Modells

Nachdem in der Tool-Definition unter der Palette ein Creation-Tool für die Transition angelegt wurde, muss nun noch das Mapping angepasst werden. Unterhalb des obersten Elements legen wir ein neues Element „Link Mapping“ an, dessen Attribut „Element“ auf die Metaklasse „Transition“ zeigen soll. Ein CompositeState hält Referenzen auf Transitions in der Liste mit dem Namen „transitions“. Aus diesem Grund wird als „Containment Feature“ die Referenz „transitions“ gewählt. Die Source / Target Features zeigen auf „from“ bzw. „to“. Damit wird spezifiziert, dass das jeweilige Ende der Transition im entsprechenden Feld gespeichert werden soll. Zum Abschluss muss noch die „TransitionConnection“ das Diagram Link und das Creation Tool entsprechend der

Abbildung 4 konfiguriert werden. Würden wir nun den Editor erneut generieren, könnte man zwei States miteinander verbinden. Bevor wir das tun, sorgen wir noch dafür, dass die Transitionen einen Label erhalten, also innerhalb des Diagrams benannt werden können. Hierzu wird erneut ein Label Mapping angelegt. Diesmal unter dem zuvor erstellten Link. Eine Transition soll textuell wie folgt editierbar sein: NAME/EVENT [GUARD]. Zur Darstellung im Diagramm reicht jedoch der Name aus. Wir wählen also die entsprechenden Features und sortieren sie in der entsprechenden Reihenfolge im dafür vorgesehenen Auswahldialog. Wie zuvor gesehen wählen wir {0}/{1} [{2}] als Edit Pattern, wohingegen {0} als View Pattern ausreicht (vergl. Abb.4 und 5).

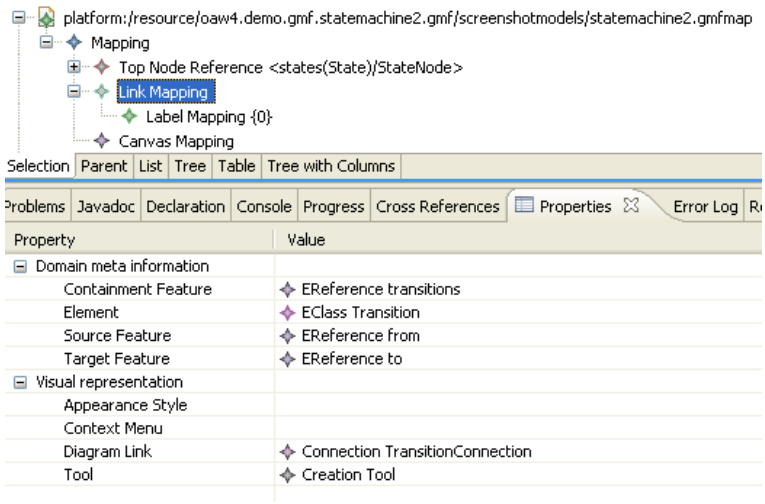


Abb. 4: Das Linkmapping

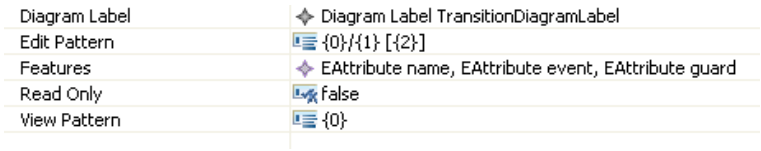
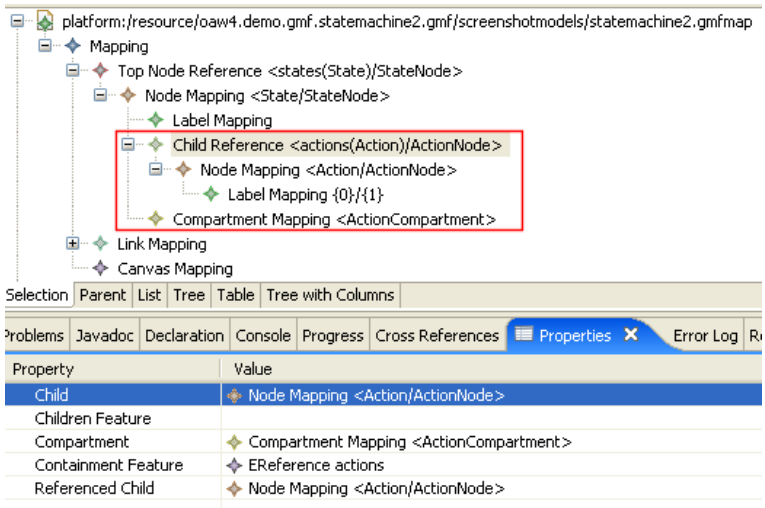


Abb. 5: Das erweiterte Labelmapping

Compartments

Als nächstes wollen wir einen Blick auf die sog. Compartments werfen. Aus dem Statemachine Metamodell ergibt sich, dass ein AbstractState Actions besitzen kann. Um dies auch graphisch zum Ausdruck bringen zu können, legen wir zunächst innerhalb der

statemachine2.gmfgraph Datei die notwendige Figure, sowie den Node für eine Action an. Wie im ersten Teil des Artikels beschrieben, legen wir hierzu unterhalb der Figure Gallery ein Rechteck für die Action an. Außerdem muß die Figur vom Mapping aus zugreifbar sein, was durch das Anlegen eines Node-Elements unterhalb des Canvas erreicht werden kann. Als nächstes erstellen wir ein Compartment unterhalb des Canvas. Dieses Compartment ist der Container welcher später die Actions beinhalten soll. Als Figure dient das „StateRectangle“, da das Compartment innerhalb dieses graphischen Elements auftauchen soll. Der Name ist „ActionCompartment“. Nachdem in der Toolingdefinition (gmftool) ein Element für die Palette zum Erzeugen von Actions angelegt wurde, widmen wir uns direkt der gmfmap-Datei. Unterhalb des Node Mappings für den State, legen wir eine neue Child Reference an. Das Containment Feature ist die Referenz auf „actions“ aus der Metaklasse AbstractState. Das bringt zum Ausdruck, dass eine Action automatisch der Liste der Actions in dem, durch das Rechteck dargestellten, State hinzugefügt werden soll. Als nächstes legen wir parallel hierzu ein Compartmentmapping an. Hier wählen wir unser Compartment aus dem gmfGraph sowie die eben erstellte Child Reference. Zurück in der Child Reference muss nun noch das Referenced Child Attribut gesetzt werden. Hierbei gibt es zwei unterschiedliche Möglichkeiten: Zum einen kann ein bestehendes Node Mapping gewählt werden, zum anderen kann unterhalb einer Referenced Child ein Node Mapping angelegt werden, welches sodann verwendet wird. Diese Freiheit bietet uns die Möglichkeit Rekursionen im Diagramm zu definieren, wie dies z.B. bei CompositeStates notwendig ist. Hierbei kann ein CompositeState wiederum CompositeStates enthalten. Wir legen unterhalb des Referenced Child ein Node Mapping an und setzten die bereits bekannten Argumente für die Metaklasse, den Diagrammknoten sowie das Creation-Tool. Nachdem wir das gmfgen sowie den Editor neu erstellt haben besitzt ein State ein Compartment, in welchem Actions angelegt werden können.



Initialer Name

Ein weiteres, evtl. ganz nützliches Feature welches GMF zur Verfügung stellt sind die sog. Feature Initializer. Diese können dazu benutzt werden beim Anlegen eines neuen Elements bestimmte Attribute des Elements zu initialisieren. So können wir z.B. im Mapping unterhalb unseres States einen Feature Seq Initializer erstellen, welchem wir eine Feature Value Spec hinzufügen. Diese Spezifikation hat drei Argumente: Zum einen das EMF-Feature (entspricht einem Attribut), auf das sich der Initializer bezieht, zum zweiten die Sprache in der das Statement zum Initialisieren des Attributs geschrieben wurde (Standard ist hier OCL, die Object Constraint Language.), und als drittes das Statement, welches interpretiert wird und dessen Ergebnis in das Attributfeld eingefügt wird.

Für den konkreten Fall wählen wir als Attribut „Name“ und als Body „'The new name'“. Nach dem Neugenerieren wird beim Anlegen eines neuen States der Name auf den obigen Wert gesetzt.

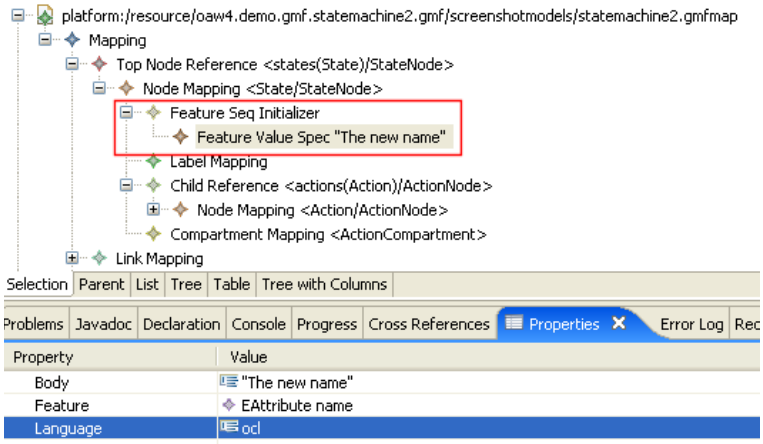


Abb. 7: Feature Seq Initializer zur Initialisierung von Attributen eines Elementsb

Fehlerüberprüfung

Um Fehler im Diagramm zu überprüfen kann ein sog. Auditcontainer im Mapping angelegt werden. Ein Auditcontainer besitzt eine oder mehr Regeln, die bei jeder Änderung im Diagramm automatisch evaluiert werden. Dieses Feature ist jedoch mit Vorsicht zu betrachten, da es meist sinnvoller ist Constrainst auf dem Metamodell zu definieren, so dass sie auch von anderen Tools zu einem späteren Zeitpunkt genutzt werden können, ohne dass Eclipse läuft. Nichts desto trotz sollten Fehler natürlich auch im

Diagramm geprüft werden. Direkt unterhalb des Mappings können Auditcontainer erstellt werden. Ein Auditcontainer enthält entweder weitere Container oder Auditregeln. Eine solche Regel benötigt eine Constraint und ein Target welches auf eine Klasse verweist für die die Constraint evaluiert werden soll.

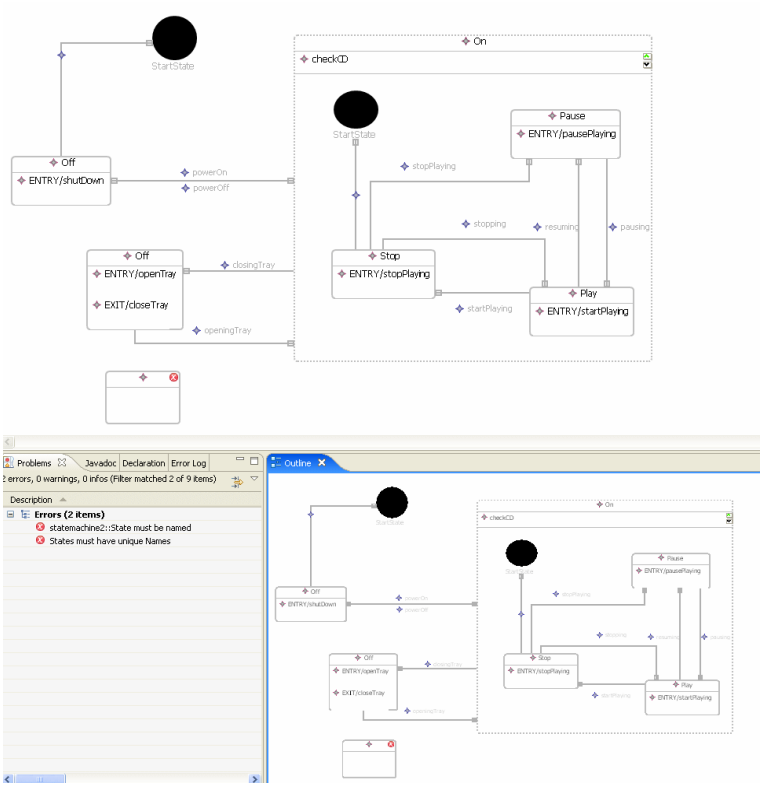


Abb. 8: Fertiger Editor. Inhaltliche Fehler werden im Problems-View dargestellt.

Ein Modell – und jetzt?

Nachdem wir nun in der Lage sind eine Statemachine zu modellieren wollen wir jetzt noch einen kurzen Ausblick darauf geben, was mit den so erstellten Modellen angestellt werden kann. Grundsätzlich gibt es zwei Möglichkeiten einem Computer beizubringen, formale Modelle zu verarbeiten / auszuführen. Ersten können solche Modelle zur Laufzeit interpretiert werden

Ziel ist es möglichst viele Informationen aus einem solchen Diagramm abzuleiten. Im Falle unserer Statemachine bedeutet dies, eine Implementierung zu generieren, bei der nur noch das Verhalten von Hand zu schreiben ist. Hierfür verwenden wir das

Codegeneratorframework openArchitectureWare (oAW). Aufgabe dieses Frameworks ist es, den Entwickler bei der Erstellung von Codegeneratoren so weit wie möglich zu unterstützen. Hierfür bietet oAW verschiedene Sprachen um zum einen das Modell auf Fehler zu überprüfen, das Modell evtl. zu modifizieren bzw. zu transformieren und schließlich Code daraus zu generieren. Die Checks, die sie zur Validierung des Modells im Generator erstellen, können selbstverständlich auch in einem GMF-Editor überprüft werden, sodass sie in Echtzeit über Modellverletzungen informiert werden. Um solche Checks zu implementieren bietet oAW eine Sprache, welche an OCL angelehnt ist. Für diese Sprache wird der gewohnter Eclipse-Editorsupport zur Verfügung gestellt.

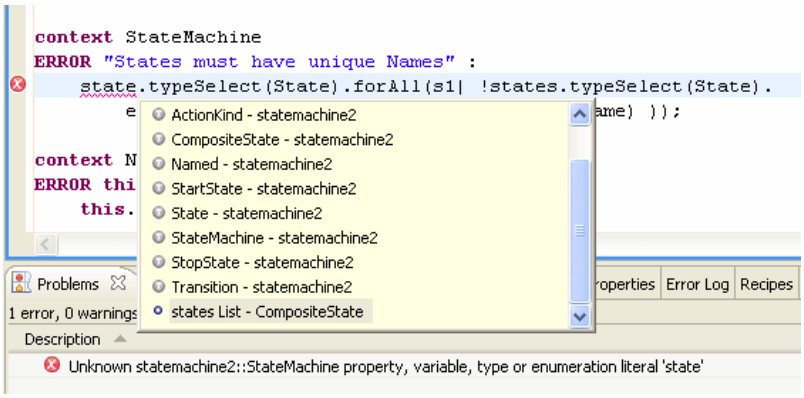


Abb. 9 openArchitectureWare Checks

Für unser Beispiel wäre es z.B. sinnvoll, jedem State eine Transition zu einem zu erstellenden Notaus-State hinzuzufügen. Dies soll jedoch nicht von Hand sondern vielmehr automatisiert geschehen. Es soll also ein entsprechende Modellmodifikation ausgeführt werden. Diese kann mittels der oAW-Sprache XTend formuliert werden. Auch hierfür bietet das Framework einen entsprechenden Eclipse-Editor

```
extension statemachine2::constraints::Statemachine;

modify(StateMachine sm) :
   sm.transitions.addAll(sm.allConcreteStates().createTransition()) ->
   sm.states.add(createShutDown()) ->
   sm;

private create State this createShutDown() :
   setName("EmergencyShutDown");

private create Transition this createTransition(State s) :
   setEvent("Error")->
   setName("Aborting") ->
   setFrom(s) ->
   setTo(createShutDown());
```

Abb. 10: Modellmodifikation

Wie in Abbildung 10 zu sehen, wird für jeden State in einer Statemachine eine neue Transition zu einem neu erzeugten Shutdownstate hinzugefügt. Außerdem werden die neuen Transitionen zur Liste der Transitionen der Statemachine addiert.

Zu guter Letzt soll aus dem modifizierten Modell entsprechender Quellcode generiert werden. Dazu benutzen wir XPand, das dritte und letzte Mitglied der oAW-Sprachfamilie.

```
«DEFINE file FOR StateMachine»
    «FILE basePath()+"/Abstract"+name.toFirstUpper()+".java"-»
       package «basePackage()»;

       public abstract class «implBaseClassName()»
            implements «actionsInterfaceName()» {

          private «statesEnumName()» currentState =
                «initialState.stateId(this)»;
          private boolean terminated = false;

          public void handleEvent( «eventsEnumName()» event ) {
             if ( terminated )
                throw new RuntimeException( "this sm is terminated!" );

             switch ( currentState ) {
                «FOREACH states AS s-»
                    case «s.shortStateId()»:
                    «FOREACH s.transitions AS t-»
                       if ( event == «t.event.eventId(this)» ) {
                          «EXPAND executeTransition(this) FOR t»
                          break;
                       }
                    «ENDFOREACH»
                    «EXPAND handleIllegalTransition»

                «ENDFOREACH»
             }
          }

          public «statesEnumName()» getCurrentState() {
              return currentState;
          }

       }
    «ENDFILE»
«ENDDEFINE»
```

Abb. 11: XPand-Editor

Der in Abbildung 11 dargestellte Ausschnitt aus einem Template erstellt das Gerüst der Statemachine. Der blau dargestellte Text wird so wie er ist ausgegeben. Der schwarze Text enthält Anweisungen, die bestimmte Werte aus den Modelelemente abfragen und vom Generator an entsprechender Stelle ausgegeben werden.

Nach dem Ausführen des Generators sollte dem Benutzer des Generators noch mitgeteilt werden was er tun muss, damit er den generierten Code benutzen kann. Hierzu stellt openArchitectureWare das Recipe-Framework zur Verfügung. Hierbei kann der Generatorentwickler dem Generatorbenutzer mitteilen, was noch von Hand implementiert werden muss. Abbildung 12 zeigt eine entsprechende Anwendung.

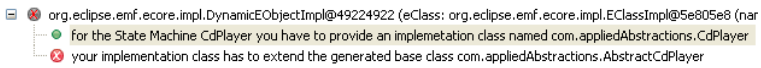


Abb. 12: Anwendung des Recipe-Frameworks