Modelltransformationen in der Praxis

Markus Völter, voelter@acm.org, www.voelter.de
Iris Groher, iris,groher.ext@siemens.com

Die Bedeutung von Modell-zu-Modell Transformationen in der Praxis steigt derzeit rapide an. In diesem Artikel möchten wir daher – basierend auf der Theorie die Arno im vorigen Artikel eingeführt hat – einige Beispiele für Modelltransformationen zeigen. Wir implementieren diese Beispiele mit der Sprache Xtend, einem Teil des openArchitectureWare Frameworks. Um an seinen einführenden, eher theoretischen Artikel anzuknüpfen, hat Arno in diesem Artikel einige Kommentare eingebaut.

Grundsätzliches

In unserer Sicht der MDSD-Welt sind Modelltransformationen ein „Implementierungsdetail" des Transformators/Generators. Das Ergebnis einer Transformation wird vom Entwickler nicht geändert und meistens nicht einmal gesehen. Abbildung 1 zeigt dies schematisch. (Arno: Dies ist eine andere Sichtweise auf den Workflow, den ich beschrieben habe)

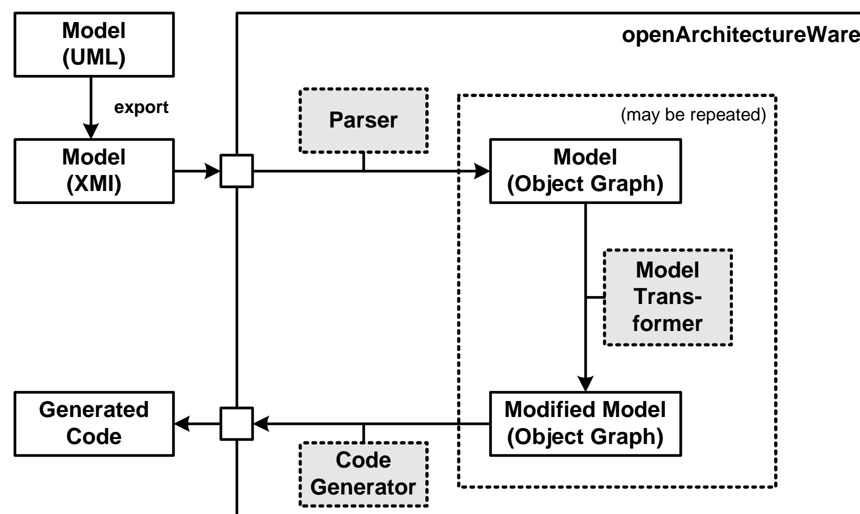

Abbildung 1: Modelltransformationen im Generator

Es gibt verschiedene Gründe dies so handzuhaben: Wenn man das Ergebnis einer Transformation vor der Weiterverarbeitung modifzieren würde, hätte man das Problem der Synchronisation im Falle einer erneuten Transformation der ersten Stufe. Real existierende Tools sind dazu derzeit schlicht nicht in der Lage. Selbst wenn die Tools fähig wären partielle, iterative Neutransformationen durchzuführen, würde man dies aber

vermutlich nicht wollen. Man stelle sich ein Eingabemodell mit 1500 Elementen vor. Nach einer Transformation sind im Zielmodell dann oft deutlich mehr Elemente vorhanden (vielleicht 4000), da das Ziel einer Transformation oft weniger abstrakt und detailreicher ist. Niemand würde freiwillig mit derart vielen Modellelementen arbeiten, wenn es sich vermeiden lässt.

Um den oben gezeigten Ansatz zu realisieren, muss man aber stattdessen einige andere Dinge beachten:

- Transformationen werden mit kleinen Modellen entwickelt. Das Ergebnis der Transformation lässt sich einfach als XML bzw. EMF Tree View auf Korrektheit überprüfen.
- Um die Transformationen robust zu halten werden diese mittels Unit Tests anhand von Referenzmodellen ausreichender Komplexität getestet (wir zeigen das unten).
- Wenn das Ergebnis einer Transformation mit Informationen angereichert werden muss, so tun wir das durch zusätzliche Modelle, die als Parameter in die Transformation eingehen.

Obige Diskussion gilt im Übrigen nicht, wenn die Modelltransformationen zur Synchronisation verschiedener (Viewpoint-spezifischer) Modelle innerhalb einer IDE dienen. In diesem Fall muss man die Modelle Event-basiert synchronisieren. Nun aber genug der Einführung, auf in die Praxis.

Modelltransformation

Eine Modelltransformation verhält sich im Wesentlichen wie eine Funktion: sie nimmt eine Reihe von Parametern entgegen und berechnet ein Ergebnis. Die Parameter – also die Eingabemodelle – bleiben dabei unverändert. (Arno: Diese Funktion transformiert einen Objektgraphen in einen anderen Objektgraphen)

Im aktuellen Beispiel geht es darum, ein UML Modell in eine Instanz eines domänenspezifischen Metamodells zu überführen. Der Hintergrund dabei ist folgender: Das UML Metamodell ist recht komplex und unübersichtlich. Wenn man basierend auf diesem Metamodell Generatoren bauen will, ist man unweigerlich mit dieser Komplexität konfrontiert. Vor allem wenn man – wie wir später sehen werden – das Modell modifizieren will, wird das UML Metamodell schnell zur Qual. Eine Best Practice um dieses Problem in den Griff zu bekommen besteht nun eben darin, dass man am Beginn der Transformationskette das UML Modell in eine Instanz eines für die Problemdomäne genau passenden Metamodells transformiert. Die Komplexität des UML Metamodells wird dadurch zwar nicht magisch entfernt, aber man lokalisiert sie auf diesen einen Transformationsschritt.

Implementierung der Transformation

Für das aktuelle Beispiel möchten wir ein UML Zustandsdiagramm als Quelle verwenden (Klassendiagramme kennt man ja mittlerweile zur Genüge ☺). Abbildung 2 zeigt ein Beispielmodell, modelliert mit MagicDraw 11.6.

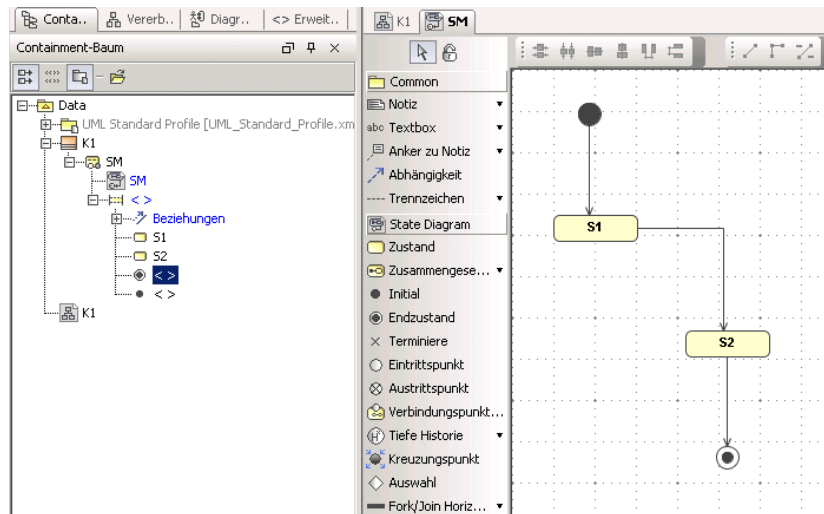

Abbildung 2: Beispielmodell

Ziel ist es, dieses UML Modell auf ein domänenspezifisches (also im Vergleich zu UML deutlich vereinfachtes) Metamodell für Zustandsmaschinen abzubilden. Dieses Zielmetamodell wird in Abbildung 3 gezeigt. Im Vergleich zur UML besitzt es beispielsweise keine hierarchischen Zustände, keine Regionen, keine Guards und auch keine Actions. Für eine praktische Anwendung ist dies sicherlich eine zu starke Vereinfachung. Aber in diesem Artikel soll es ja um Modell-zu-Modell Transformationen gehen, und nicht darum, mit realistischen Zustandsmaschinen zu arbeiten. (Arno: Hier geht es um den Wechsel der Abstraktionsebene – das Ziel-Metamodell ist konkreter an der Zielplattform orientiert, während das UML2-Metamodell sehr viel allgemeiner und damit komplexer ist)

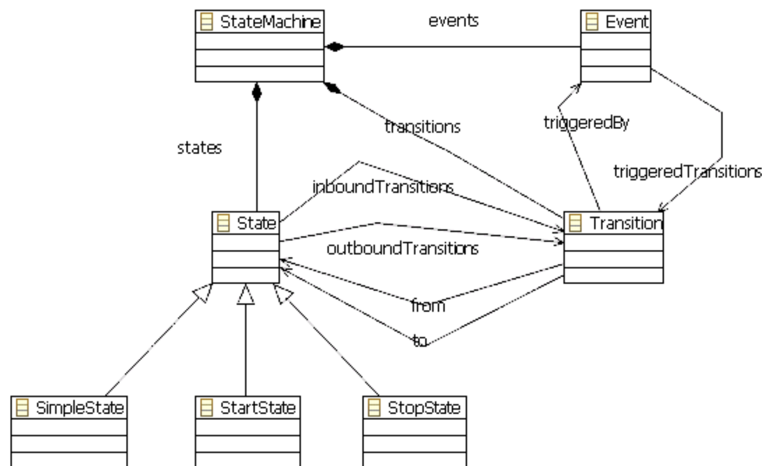

Abbildung 3: Metamodell für Zustandsmaschinen

Um mittels openArchitectureWare (oAW) eine Transformation zu bauen, müssen wir zunächst den Workflow definieren. Dieser legt die Schritte fest, die der Generator durchführt um das Eingabemodell zu verarbeiten. In unserem Fall sieht der Workflow folgendermaßen aus. Zunächst konfigurieren wir oAW für die Verarbeitung von UML2 Modellen. Dann laden wir das UML Modell mit Hilfe des XmiReaders und legen es im Workflow unter dem Namen *uml2model* ab (hier nicht gezeigt). Wir überprüfen dann einige Constraints, denn es macht keinen Sinn, fehlerhafte Modelle zu transformieren. Dann starten wir die *XtendComponent* – hier passiert die eigentliche Transformation, die wir weiter unten beschreiben werden. Da wir in dieser Transformation UML Modelle lesen und Instanzen des Zustandsmaschinen-Metamodells schreiben, müssen wir diese beiden Metamodelle in der Komponente konfigurieren. Dann rufen wir die eigentliche Transformation auf (<*invoke …*>) und definieren mittels *outputSlot* unter welchem Namen das neu erzeugte Modell im Workflowcontext abgelegt werden soll.

```
<workflow>
  <bean class="oaw.uml2.Setup" standardUML2Setup="true"/>
  <component class="org.openarchitectureware.emf.XmiReader" …
  <component
        class="org.openarchitectureware.check.CheckComponent">
  <component class="oaw.xtend.XtendComponent">
    <metaModel class="oaw.uml2.UML2MetaModel"/>
    <metaModel class="oaw.type.emf.EmfMetaModel">
      <metaModelFile value="stama.ecore"/>
    </metaModel>
    <invoke value="uml2trafo::main(uml2model)"/>
    <outputSlot value="stamaModel"/>
  </component>
  <component class="oaw.emf.XmiWriter"…
</workflow>
```

Wie sieht nun die eigentliche Transformation aus? Im Folgenden werden wir uns den betreffenden Code Schritt für Schritt ansehen. Die Transformation startet mit der Funktion *main*, die ja auch aus dem Workflow aufgerufen wird. Sie nimmt ein UML Modell als Parameter. Der Einfachheit halber gehen wir davon aus, dass das UML Modell nur eine Zustandsmaschine enthält, und dass diese der einzigen im Modell vorhandenen Klasse zugeordnet ist. Wir iterieren also über alle *ownedElement*s unseres Modells, filtern die Klassen heraus, iterieren über deren Verhalten (*ownedBehavior*) und selektieren dort nur die *StateMachine*s. Da wir nur eine Zustandsmaschine verarbeiten wollen, holen wir uns von der betreffenden Liste nur das erste Element und rufen darauf die Funktion *newSM()* auf. Deren Aufgabe ist es, aus der *UML::StateMachine* eine *stama::Statemachine* zu erzeugen.

```
main( uml::Model model ):
  model.ownedElement.typeSelect(uml::Class).ownedBehavior.
    typeSelect(uml::StateMachine).first().newSM();
```

Die Funktion *newSM()* unterscheidet sich von *main()* vor allem dadurch, dass es eine sogenannte *create*-Funktion ist (siehe entspr. Schlüsselwort). Dies bedeutet, dass die Funktion, wenn sie ausgeführt wird, als Seiteneffekt ein Objekt des auf *create* folgenden Typs erzeugt – in diesem Fall eine *stama::StateMachine*. (Arno: Das ist ein schönes Beispiel dafür, wie die create-Extensions den Umgang mit Objektidentität elegant gestalten)

Da die Funktion für *UML::StateMachine*s aufgerufen wird, müssen wir diesen Typ natürlich als Parameter für die Funktion definieren.

```
create stama::StateMachine newSM( uml::StateMachine sm ):
```

Die Funktion setzt zunächst den Name der neu erzeugten Zustandsmaschine.

```
  setName( sm.name ) ->
```

Dann werden alle Zustände der UML Zustandsmaschine in Zustände aus dem *stama* Metamodell transformiert. Dazu iterieren wir über alle Zustände in der ersten (und für dieses Beispiel einzigen) Region der Zustandsmaschine und *map()*pen diese auf Zustände aus *stama*. Wir fügen all diese Zustände zur Liste der Zustände der gerade erzeugten Zustandsmaschine hinzu. Danach tun wir dasselbe mit den Transitionen.

```
  setStates( sm.region.first().ownedMember.
     typeSelect(uml::Vertex).map() ) ->
  setTransitions( sm.region.first().ownedMember.
     typeSelect(uml::Transition).map() );
```

Bei der Transformation der Zustände müssen wir darauf achten, dass es sowohl in UML als auch in *stama* verschiedene Untermetaklassen von *State* (bzw. *Vertex* in UML) gibt, um zwischen Start-, Stop- und Normalzuständen zu unterscheiden. Wir verwenden dafür den automatischen polymorphen Dispatch von Xtend; es wird immer automatisch die Variante von *map()* aufgerufen, deren Typsignatur am genauesten zu dem aktuellen Vertex-Objekt passt. Die eigentliche Implementierung der Funktionen ist trivial, sie kopiert lediglich den Name des Zustandes.

```
create stama::SimpleState map( uml::Vertex v):
```

```
  setName( v.name );
create stama::StartState map( uml::Pseudostate s):
  setName( s.name );
create stama::StopState map( uml::FinalState s):
  setName( s.name );
```

Um unsere Transformation zu komplettieren fehlt uns nun noch die Transformation der Transitionen. Die aus *newSM()* aufgerufene Funktion *map(uml::Transition)* wird folgendermaßen implementiert:

```
create stama::Transition map( uml::Transition t):
  setName( t.name ) ->
  setFrom( t.source.map() ) ->
  setTo( t.target.map() );
```

Dabei kommt das Ergebnis-Caching Feature von create-Funktionen schön zur Geltung. Um die Quelle und das Ziel der Transition zu setzen (diese beiden Referenzen müssen ja auf die aus Quelle und Ziel der UML Transformation hervorgegangenen *stama::State*s zeigen) rufen wir einfach die betreffende *map()*-Funktion nochmals auf. Wir bekommen die gleichen Objekte wie beim ersten Aufruf zurückgeliefert.

Testen

Um zu testen, ob die Transformation tatsächlich funktioniert, gibt es mehrere Möglichkeiten. Zum einen können wir das betreffende Modell einfach in eine XMI-Datei serialisieren und „ankucken“. Dies ist während der Entwicklung der Transformation sicherlich sinnvoll. Hier ist das betreffende Modell als (eigentlich ganz gut lesbares) XMI:

```
<stama:StateMachine xmi:version="2.0" ... name="SM">
  <states xsi:type="stama:StartState" />
  <states xsi:type="stama:SimpleState" name="S1"/>
  <states xsi:type="stama:SimpleState" name="S2"/>
  <states xsi:type="stama:StopState"/>
  <transitions to="//@states.1" from="//@states.0"/>
  <transitions to="//@states.2" from="//@states.1"/>
  <transitions to="//@states.3" from="//@states.2"/>
</stama:StateMachine>
```

Allerdings ist es wichtig, dass dieses „Ankucken“ automatisierbar ist – es macht also Sinn, das Äquivalent eines Unit-Tests zu implementieren. Dies bedeutet, dass wir Constraints implementieren, die das Ergebnismodell auf Korrektheit überprüfen. Diese Constraints sind nicht Teil des Metamodells sondern prüfen spezifische Eigenschaften der Transformation bzgl. eines Referenzmodells ab. Im Folgenden der betreffende Constraint-Code.

Zunächst stellen wir sicher, dass es unter den Zuständen unserer Zustandsmaschine jeweils genau einen gibt, der *S1* bzw. *S2* heißt, und dass dies jeweils Instanzen von *SimpleState* sind.

```
context StateMachine ERROR "state S1 or S2 missing":
```

```
    states.select( s |
        s.name == "S1" && SimpleState.isInstance(s) ).size == 1 &&
    states.select( s |
        s.name == "S2" && SimpleState.isInstance(s) ).size == 1;
```

Dann überprüfen wir die Transitionen. Dabei stellen wir sicher, dass sie die richtigen Zustände „verbinden".

```
context StateMachine ERROR "Transition missing":
    transitions.exists( t |
        t.from.name == "S1" && t.to.name == "S2" ) &&
    transitions.exists( t |
        StartState.isInstance(t.from) && t.to.name == "S1" ) &&
    transitions.exists( t |
        t.from.name == "S2" && StopState.isInstance(t.to) );
```

Um diese Tests auszuführen, müssen sie natürlich in den Workflow eingebunden werden. Dies geschieht mittels eines ganz normalen Aufrufs der oAW *CheckComponent* (siehe oben).

Wozu Modelltransformationen?

Modelltransformationen sind in verschiedener Hinsicht wichtig. Einen Grund haben wir oben schon gezeigt: Die Abbildung von Modellen die auf verschiedenen (aber inhaltlich ähnlichen) Metamodellen beruhen. Dies ist vor allem bei Toolintegrationen relevant. Ein weiterer Grund ist die schrittweise Verfeinerung von Modellen. Dies entspricht der klassischen MDA-Sicht, wo man zunächst auf Ebene eines technologieunabhängigen Modells (PIM) modelliert und dieses dann in ein technologiespezifisches Modell (PSM) transformiert. Etwas verallgemeinert bedeutet dies, dass man einen Stack von MDSD-Infrastrukturen hat, wobei die Eingabemodelle für die n-te Ebene die Ausgabemodelle der n+1-ten Ebene sind (siehe Abbildung 4). Noch weiter verallgemeinert geht es darum, eine Transformation die eine große „semantische Lücke" überbrücken muss in kleinere, leichter verständliche und idealerweise wieder verwendbare Teiltransformationen zu zerlegen. (Arno: Das ist das schrittweise Bewegen des Modells von einer Abstraktionsebene auf eine andere, das ich beschrieben habe)

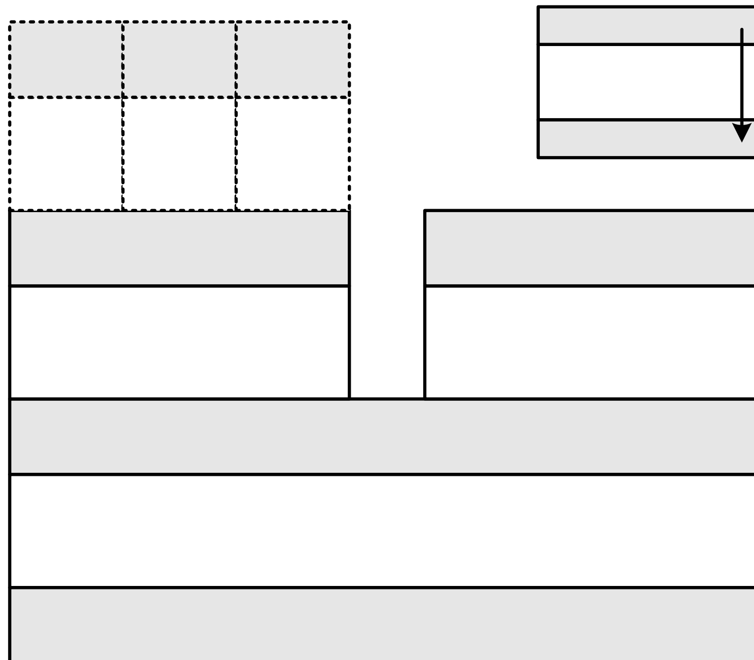

Abbildung 4: Kaskadierung von MDSD-Infrastrukturen

Modellmodifikationen

Modellmodifikationen dienen dazu, Modelle „an Ort und Stelle" zu modifizieren. Man stelle sich dazu wieder obige Zustandsmaschine vor mit folgenden zusätzlichen Anforderungen:

- Zustandsmaschinen sollen eine Notaus-Funktionalität haben können. Das bedeutet, dass die Zustandsmaschine in jedem Zustand auf ein Event NOTAUS reagieren soll indem sie sich automatisch beendet.
- Dieses Feature soll optional sein – es sollen Zustandsmaschinen mit und ohne dieses Feature möglich sein.

Konkret bedeutet dies: ein neuer Zustand „Notaus" wird eingeführt sowie eine Transition von jedem „normalen" Zustand aus hin zu diesem Zustand der auf das Event NOTAUS reagiert. Um diese Anforderungen zu implementieren könnten wir natürlich von Hand an allen betroffenen Zustandsmaschinen-Modellen die nötigen Änderungen durchführen. Dies ist in der Praxis allerdings kein besonders skalierbarer Ansatz. Alternativ könnten wir in den Templates, die den Zustandsmaschinen-Code generieren natürlich die betreffenden Zeilen einfügen. Dies führt aber zu sehr komplizierten Templates, da jetzt algorithmischer Code („für alle States im Modell tue folgendes…") mit Notaus-spezifischem Code vermischt ist. Um das Feature optional zu gestalten, muss der Notaus-spezifische Code

...
...
Don
Functi
MDSD

auch noch durch entsprechende IF-Statements von der Konfiguration abhängig gemacht werden. Nicht schön…

Die einfachste und modularste Art, das optionale Feature „Notaus“ in das System einzubringen besteht darin, das betreffende Modell durch eine Modellmodifikation zu ergänzen, wenn die betreffende Zustandsmaschine das Feature Notaus haben soll. Den dazu nötigen Code schauen wir uns wieder im Detail an.

Die Toplevel-Funktion heißt hier *notausTransform*. Man beachte, dass diese kein neues Objekt erzeugt, sondern auf der als Parameter übergebenen Zustandsmaschine arbeitet.

```
notausTransform( StateMachine this ):
```

Als erstes fügt sie zu der Liste der Zustände einen neuen hinzu; dieser wird von der *create*-Funktion *notausState()* erzeugt. Dann erstellen wir mit demselben Mechanismus ein neues Event. Schlussendlich iterieren wir über alle *SimpleState*s und rufen auf jedem die *create*-Funktion *notausTransition()* auf. Die daraus resultierende Collection von Transitionen fügen wir der Collection bereits vorhandener Transitionen hinzu.

```
 states.add( notausState() ) ->
 events.add( notausEvent() ) ->
 transitions.addAll(
     states.typeSelect(SimpleState).notausTransition() );
```

Die *create*-Funktionen für das Event und den Notaus-Zustand sind trivial, sie erzeugen das Objekt und setzen den Namen.

```
create Event notausEvent():
 setName("notausButtonPressed");
create SimpleState notausState():
 setName("notaus");
```

Interessanter wird es wieder bei der Erstellung der Transformationen. Das Caching-Feature von Xtend kommt zum Einsatz um zu verhindern, dass mehr als ein NOTAUS Event und mehr als ein Notaus-Zustand erzeugt werden.

```
create Transition notausTransition( SimpleState sm ):
 setName(sm.name+"_notaus") ->
 setFrom( sm ) ->
 setTo( notausState() ) ->
 setTriggeredBy( notausEvent() );
```

Um diese Modifikation zu testen wird man auch wieder Constraints erstellen, die die Rolle des Unit-Tests spielen.

Wozu Modellmodifikationen?

Wie aus dem obigen Beispiel klar werden sollte, sind Modellmodifikationen Ergänzungen am Modell, die sich leichter per Algorithmus beschreiben lassen als durch Handarbeit auf Modellebene. Oft sind solche Ergänzungen optional – man möchte sie einfach an- und

abschaltbar machen. Eine Modifikation der Templates im nachfolgenden Schritt ist oft nicht praktikabel.

Modell-Weaving

Die dritte und letzte Art der Modelltransformation die wir in diesem Artikel zeigen wollen ist das Modellweben – im Sinne von Aspekt-orientierter Programmierung (AOP). Bei AOP geht es darum, im Quellcode querschneidende Belange in eine neue Art von Modul – genannt Aspekt – zu modularisieren. Technisch geschieht das folgendermaßen:

- Der Code des querschneidenden Belangs wird in ein separates Modul faktorisiert – den Aspekt (genau genommen ist der faktorisierte Code nur ein Teil des Aspekts, genannt Advice)
- Ein Werkzeug, der sogenannte Aspektweber (Weaver), fügt den Aspektcode zum Basisprogramm hinzu. Ein vom Entwickler im Rahmen des Aspekts definierter Pointcut definiert, wie und wo gewoben werden soll.

Um das Konzept zu verdeutlichen, hier wieder ein Beispiel. Wir wollen wieder die Notaus-Funktionalität zu einer Zustandsmaschine hinzufügen – dieses Mal allerdings mit Hilfe des Modelweavers. Abbildung 5 zeigt eine einfache Zustandsmaschine, basierend auf dem obigen Metamodell. Dieses Modell dient als Basismodell, in welches wir den Aspekt weben werden.

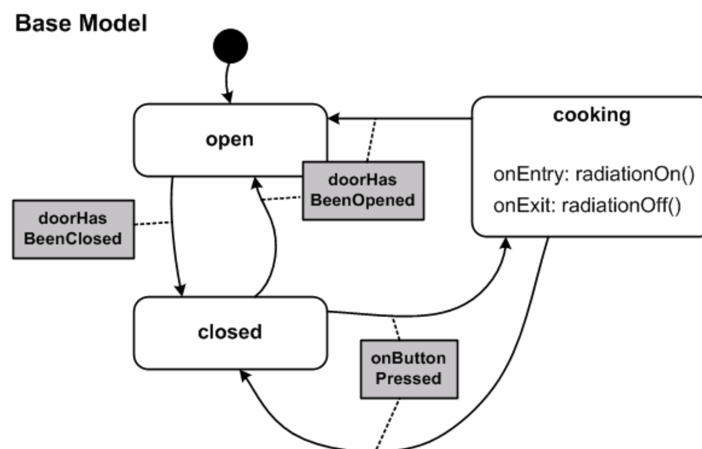

Abbildung 5: Basismodell einer Zustandsmaschine

Der Aspekt „Notaus“ wird dargestellt als Modellfragement, das nur die zusätzlichen Anteile enthält. Außerden enthält es „Andockpunkte“, die definieren, wie das Aspektmodell in das Basismodell eingewoben werden soll. Abbildung 6 zeigt zwei alternative Aspektmodelle. Modell 1 verwendet den Platzhalter * um auszudrücken, dass das betreffende Element mit allen Instanzen des betreffenden Typs (SimpleState) assoziiert

werden soll. Das zweite Modell verwendet eine Pointcut-Expression die definiert, welche Untermenge der *SimpleStates* (in diesem Fall, alle) als Ziel für den Webevorgang dienen sollen.

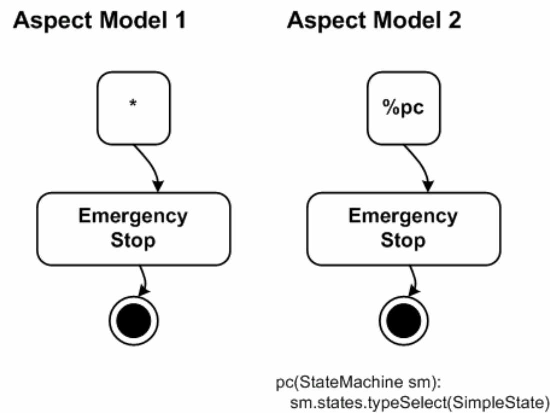

Abbildung 6: Zwei Aspektmodelle

Das Ergebnis des Webevorgangs (sowohl für Aspekt 1 als auch für Aspekt 2) ist in Abbildung 7 dargestellt – das gleiche Ergebnis wie nach der Modellmodifikation.

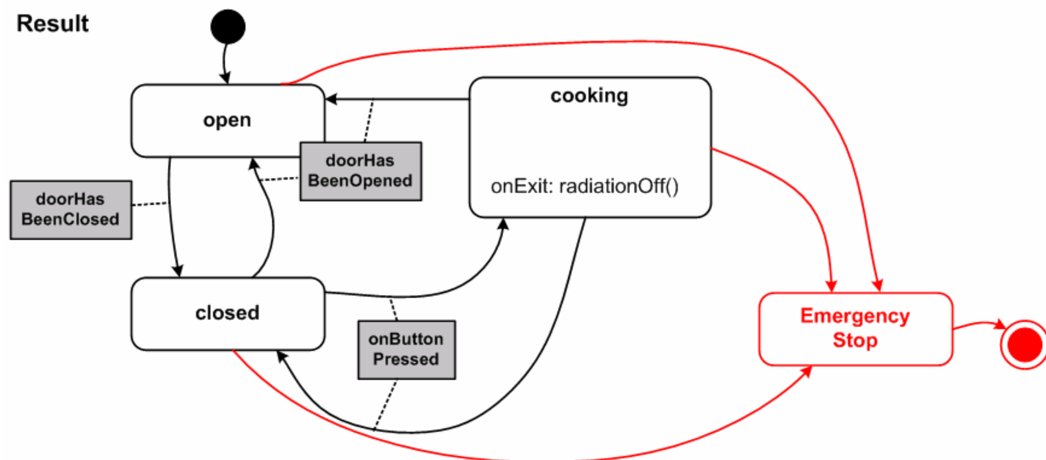

Abbildung 7: Ergebnis des Webens

Wozu Model-Weaving?

Model Weaving bietet eine komfortable Möglichkeit,querschneidende Belange in einem Aspektmodell zu kapseln und automatisch in das Modell zu weben. Der vorgestellte Modelweaver webt standardmäßig auf Basis der Namensgleichheit von Elementen des Basis- und der Aspektmodelle. Desweiteren können Pointcuts mithilfe von Platzhaltern und Expressions definiert werden, die eine Menge an Elementen als „Webeziel" liefern. Es sind jedoch nicht nur querschneidende Belange die mit dem Modelweaver elegant gekapselt werden können, Aspektweben auf Modellebene spielt in der Entwicklung von Software Produktlinien eine entscheidende Rolle. Optionale Features einer Produktlinie können als Aspektmodell definiert werden und bei Bedarf in das Basismodell (Kern der Produktlinie) gewoben werden. Somit enthält das Modell der Produktlinie nur den allen Produkten gemeinsamen Kern und ist dadurch viel einfacher verwaltbar als würde das Modell sämtliche Optionen enthalten. Das Hinzufügen optionaler Features zu einem Produktlinien-Kern wird allgemein als positive Variabilität bezeichnet.

Fazit

Wir haben in diesem Artikel einige praktische Beispiele für Modelltransformationen gezeigt. Die Beispiele haben wir in drei Kategorien aufgeteilt: Modelltransformationen im eigentlichen Sinn, Modellmodifikationen und Modell-Weaving. Modelltransformationen verhalten sich wie Funktionen die als Parameter ein Eingabemodell erhalten und als Ergebnismodell ein neues Modell liefern. Modellmodifikationen verändern, wie der Name schon sagt, das Eingabemodell und ergänzen es um bestimmte Teile. Modell-Weaving bezeichnet das Weben im Sinne von AO auf Modellebene. Querschneidene Belange können dadurch gekapselt werden und nach Bedarf in das Modell eingefügt werden. Weitere Informationen, Beispiele und natürlich die verwendeten Werkzeuge können auf der oAW [oAW] Webseite heruntergeladen werden.

Literatur und Links

[oAW] openArchitectureWare Webseite, http://www.eclipse.org/gmt/oaw/