von karsten ehrig, claudia ermel und gabriele taentzer

mehr zum thema:
tfs.cs.tu-berlin.de/petrieditor

ERSTELLUNG EINES GRAFISCHEN EDITOR-PLUG-INS MIT ECLIPSE EMF UND GEF

In dem Artikel wird die Generierung eines einfachen baumbasierten Eclipse Editor-Plug-Ins anhand eines Editors für Petrinetze mit dem „Eclipse Modeling Framework" (EMF) beschrieben. Aufbauend auf den generierten Modellcode wird die Implementierung eines zweiten komplexeren grafischen Editor-Plug-Ins mit dem „Graphical Editor Framework" (GEF) erläutert.

Das *Eclipse Modeling Framework (EMF)* und das *Graphical Editor Framework (GEF)* bieten die Möglichkeit, in kurzer Zeit ein grafisches Editor-*Plug-In* zu erstellen, das auf einem formalen Modell beruht. EMF unterstützt hierbei die automatische Java-Codegenerierung des Editor-Modells aus einem EMF-Klassendiagramm. Mit GEF wird der Modellcode in das grafische Editor-*Plug-In* integriert, das direkt in der „Eclipse Runtime-Workbench" ausgeführt werden kann. Hierbei stellt GEF eine Reihe von Standardoperationen für grafische Editoren bereit, z. B. eine Editorpalette, Zooming und Funktionalität zum Rückgängigmachen/Wiederherstellen von Editieroperationen.

Die Programmierung eines grafischen Editors allein auf der Basis von Java-Swing oder SWT ist eine mühevolle und aufwändige Arbeit. Hinzu kommt, dass das grafische Layout und die Benutzerführung in unterschiedlichen grafischen Anwendungen sehr stark variieren können, was den Benutzer lange Einarbeitungszeit kostet und unter anderem zu Problemen beim Datenaustausch mit anderen Anwendungen führt.

Alle diese Probleme sind in Eclipse (vgl. [Ecl]) gelöst, in dem dem Entwickler nicht nur eine hervorragende Programmierumgebung für Java (vgl. [Sun]), sondern darüber hinaus auch eine Anwendungsumgebung angeboten wird, die die entwickelten Programme mittels *Plug-In*-Technologie integriert. Wesentlich unterstützt wird die Programmierarbeit durch zahlreiche Frameworks und *Plug-Ins*. Im Vordergrund steht hierbei die grafische Programmentwicklung. Im Folgenden wird die prinzipielle Arbeitsweise mit EMF und GEF gezeigt, indem ein einfaches grafisches Editor-*Plug-In* für Petrinetze in Eclipse erstellt wird.

Petrinetze

Für einen einfachen Editor eignen sich Petrinetze (vgl. [Rei98]), die in der Informatik weit verbreitet sind, um nebenläufige Prozesse zu beschreiben. Die Stellen eines Petrinetzes modellieren passive Systemteile, während die Transitionen einzelne Prozessaktionen modellieren. Abbildung 1 zeigt als Beispiel für ein Petrinetz ein Produzenten/Verbraucher-System. Auf den Stellen ready to produce, buffer empty und ready to remove befinden sich *Token*, die aktuelle Zustände markieren. In Bedingungs/Ereignis-Netzen, einer speziellen Petrinetzvariante, ist nur maximal ein *Token* pro Stelle zugelassen. Durch Schalten der Transitionen produce, deliver, remove und consume ändern sich die Zustände anhand der Pfeile zwischen den

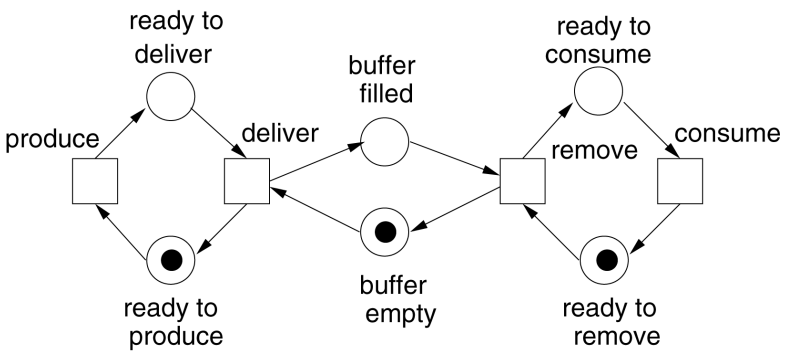

Abb. 1: Petrinetz eines Produzenten/Verbraucher-Systems

die autoren

Karsten Ehrig (E-Mail: karstene@cs.tu-berlin.de) ist wissenschaftlicher Mitarbeiter im DFG Projekt „Anwendung von Graphtransformationen auf visuelle Modellierungssprachen" am Institut für Softwaretechnik und Theoretische Informatik der TU Berlin.

Claudia Ermel (E-Mail: lieske@cs.tu-berlin.de) ist Programmiererin am selben Institut und beschäftigt sich mit der Simulation und Animation von visuellen Verhaltensmodellen, insbesondere von Petrinetzen.

Gabriele Taentzer (E-Mail: gabi@cs.tu-berlin.de) ist Oberassistentin am selben Institut. Sie leitet die Arbeitsgruppe zu visuellen Sprachen und Graphtransformation an der TU Berlin und hat langjährige Erfahrungen in der objektorientierten Softwareentwicklung basierend auf formalen Konzepten.

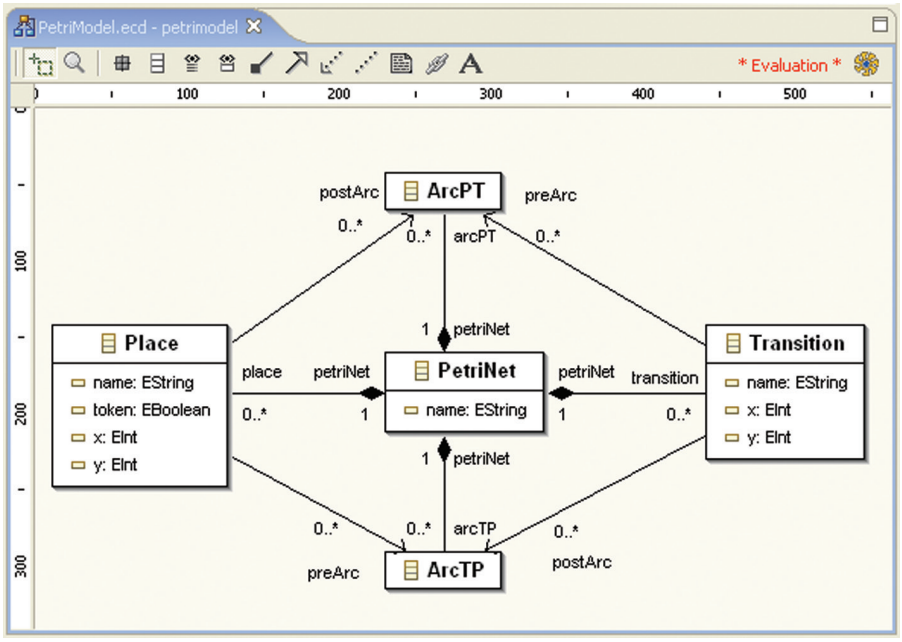

Abb. 2: EMF-Petrinetzmodell

Stellen und Transitionen. Beispielsweise bewirkt das Schalten der Transition produce, dass das *Token* von der Stelle ready to produce verschwindet und ein *Token* an der Stelle ready to deliver erscheint.

Editormodell

Da ein grafisches Editor-*Plug-In* mit GEF nach der *Model/View/Controller*-Architektur aufgebaut ist, wird zunächst ein Editormodell mit dem EMF (vgl. [EMF]) erstellt, das die grafischen Objekte im Editor verwaltet und in einem auf XML (vgl. [XML]) basierenden Format abspeichert. Hierzu muss keine Zeile Java-Code selbst geschrieben werden. Mit dem UML-*Plug-In* „OMONDO" (vgl. [Omo]) ist ein kommerzielles, in der Grundversion kostenloses Eclipse-*Plug-In* verfügbar, mit dem sich leicht Klassendiagramme grafisch erstellen lassen, welche die Grundlage für das Editor-Modell sind. OMONDO ist seinerseits selbst ein grafisches *Editor-Plug-In* für Eclipse auf Basis von EMF und GEF, allerdings auf einem sehr viel komplexeren Niveau als das in diesem Artikel vorgestellte Petrinetzeditor-*Plug-In*.

Für das Petrinetzeditor-Modell wird mit dem EMF-Klassendiagramm eine spezielle Art von Klassendiagramm benötigt, das eine abgespeckte Version eines UML-Klassendiagramms (vgl. [OMG]) darstellt. Ein wesentlicher Unterschied besteht darin, dass Attributtypen mit einem `E` beginnen. So heißt z. B. der Datentyp `String` unter EMF `EString` und `Int` heißt unter EMF `EInt`.

Um das EMF-Modell zu erstellen, muss in einem neuen Eclipse-Projekt ein EMF-Klassendiagramm unter „File > New > Other > EMF Diagrams > EMF Class Diagram" erstellt werden. Falls der Menüpunkt nicht verfügbar ist, ist das OMONDO-UML-*Plug-In* nicht richtig installiert. Die grafische Modellierung gestaltet sich sehr einfach. Abbildung 2 zeigt das fertige EMF-Modell für den Petrinetzeditor.

Wichtig ist die Basisklasse `PetriNet`, die die Klassen des Petrinetzes enthält und später bei der Instanziierung des baumbasierten Petrieditors angegeben werden muss. Bei der Modellierung ist zu beachten, dass sämtliche Klassen direkt oder über andere Klassen indirekt über Aggregationen (schwarze Raute) mit der Basisklasse `PetriNet` verbunden sind – nur so werden die weiteren Klassen später im Modell abgespeichert. Die Stellen und Transitionen werden durch die Klassen `Place` und `Transition` modelliert. Die Klassen `ArcPT` und `ArcTP` repräsentieren die Pfeile zwischen ihnen. Stellen und Transitionen haben dabei Namen (`name`) und eine Position im grafischen Editor, die durch die Attribute `x` und `y` modelliert wird. Stellen können zusätzlich noch *Token* besitzen, wobei in diesem Fall durch das boolesche Attribut `token` nur maximal ein *Token* zugelassen wird, d. h. das Petrinetz wird als Bedingungs/Ereignis-Netz modelliert.

Modellgenerierung

Die Generierung des Modellcodes gestaltet sich durch EMF denkbar einfach. Im OMONDO-*Plug-In* gibt es im Kontextmenü (rechte Maustaste im Editorfenster) vier Menüpunkte zur Generierung von Modellcode:

1. *Generate Model Code* generiert die fertigen Modellklassen im aktuellen Eclipse-Projekt, die als Basis für das grafische Editor-*Plug-In* dienen.
2. *Generate Edit Code* generiert zusätzlich das Projekt „<<MeinProjektname. Edit>>" mit Editieroperationen für einen baumbasierten Editor.
3. *Generate Editor Code* generiert zusätzlich das Projekt „<<MeinProjektname. Editor>>" mit einem baumbasierten Editor, der direkt in der Eclipse Runtime-Workbench ausgeführt werden kann.
4. *Generate All* generiert den in den Punkten 1-3 aufgeführten Code.

Vor der ersten Generierung muss im Dialogfenster ein beliebiger Paketname für das Modell und die EMF-Modell-Basisklasse `PetriNet` angegeben werden. Für das grafische Editor-*Plug-In* wird nur der Menüpunkt „Generate Model Code" benötigt. Allerdings lässt sich durch die Menüpunkte „Generate Edit Code" und „Generate Editor Code" bereits ein recht primitiver, baumbasierter Editor generieren, der direkt in der Eclipse-Runtime-Workbench ausgeführt werden kann.

Die Runtime-Workbench in Eclipse lässt sich unter „Run > Run > Runtime-Workbench" starten. In einem neuen Projekt („File > New > Other > Simple Project") wird der baumbasierte Petrinetz-Editor durch den *Wizard* („File > New > Example EMF Model Creation Wizard > MeinPetriModel") initialisiert. Für „MeinPetriModel" erscheint der Name des Modells, der vor der Generierung angegeben wurde. Abschließend muss im *Wizard* erneut die Basisklasse `PetriNet` angegeben werden. Abbildung 3 zeigt den generierten baumbasierten Petrinetzeditor in Aktion. Im Fenster „Properties" („Window > Show

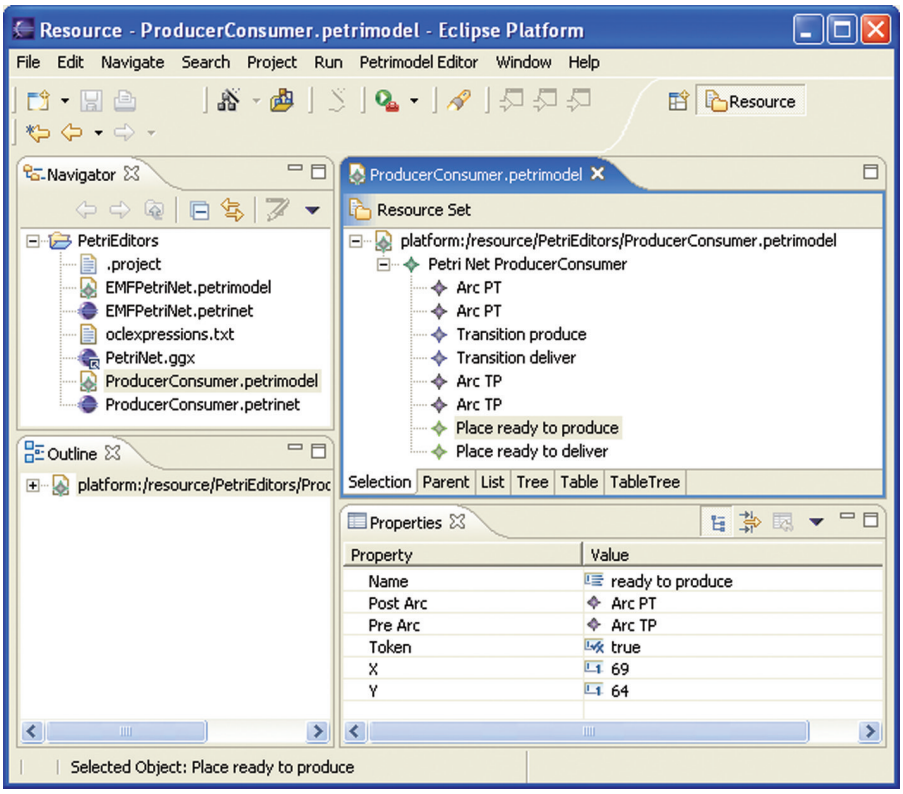

Abb. 3: Generierter baumbasierter Petrinetzeditor

View > Properties") lassen sich die Attribute des selektierten Objekts editieren.

GEF-basierter Petrinetzeditor

Für den GEF-basierten Petrinetzeditor ist leider etwas mehr Handarbeit gefragt. Im Unterschied zu EMF bietet das GEF (vgl. [GEF]) keine Code-Generatorfunktionalität. Es ist lediglich ein Framework zur Unterstützung der Entwicklung grafischer Editoren auf der Basis von Modellen. In unserem Fall bildet der mit „Generate Model Code" generierte Petrinetz-Modellcode die Basis des grafischen Editors. Allerdings bietet GEF eine reiche Palette von Editoreigenschaften wie Kopieren und Einfügen, Rückgängigmachen, Wiederherstellen und Zoomen. Auch das Anlegen von Menüs und Symbolleisten und deren Verknüpfung mit Editor-Funktionen wird durch die Verwendung von GEF erleichtert. GEF-Editoren können nur als Eclipse-*Plug-In* und nicht eigenständig gestartet werden. Ein Vorteil dieser Einschränkung ist das einheitliche, elegante, zeitgemäße Erscheinungsbild des erstellten Editor-*Plug-Ins*. Abbildung 4 zeigt den GEF-basierten Petrinetzeditor mit einem editierten Petrinetz.

Download und Installation

Der gesamte GEF-basierte Petrinetzeditor kann unter der URL tfs.cs.tu-berlin.de/petrieditor heruntergeladen werden. Auf der Webseite sind die Projekte „EMFPetriEditor" (GEF-basierter Petrinetz-Editor) und „EMFPetriModel" (EMF-Modellcode zum „Petrinetz-Editor") als ZIP-Archive vorhanden. Beide entpackten Archive werden unter dem Menüpunkt „File > Import > Existing Project into Workspace" in die Eclipse-Workspace eingebunden. Anschließend kann der Petrinetz-Editor in der Eclipse-Runtime-Workbench („Run > Run > Runtime-Workbench") ausgeführt werden. Zum Starten des Petrinetz-Editors ist kein Wizard notwendig. Stattdessen startet sich der Editor durch Erstellen einer neuen Datei mit der Endung .petrinet (z.B. my.petrinet) im Datei-Dialog („File > New > File"). Die Datei kann in dem schon vorhandenen Projekt für den baumbasierten Editor erstellt werden. Entscheidend ist die Endung .petrinet, die in der Datei plugin.xml im Attribut extensions festgelegt wird (siehe Listing 1). Weiterhin wird definiert, dass der Petrinetzeditor als Erweiterung („extension point") des Standard-Eclipse*Plug-Ins* org.eclipse.ui.editors geladen wird. Die Attribute contributorClass und class beschreiben die Schnittstellenklassen des Petrinetzeditor-*Plug-Ins* zur Eclipse-Umgebung.

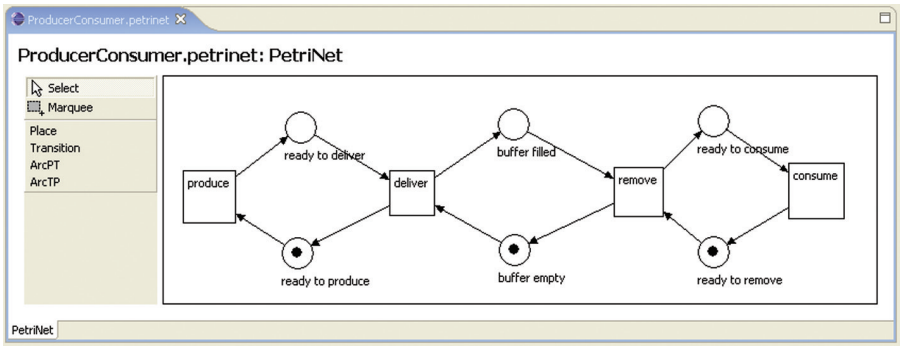

Abb. 4: GEF-basierter Petrinetzeditor

Interne Architektur von GEF-Editoren

GEF-Editoren liegt die *Model/View/Controller*-Architektur zugrunde (siehe Abb. 5). Die *Controller* (so genannte *EditParts* bei GEF) steuern das Zusammenwirken des Modells (z. B. der von EMF generierten Klassen) mit seinen Sichten (*Views*) (z. B. das Panel des grafischen Editors). So führt beispielsweise das Absetzen des Kommandos Delete zunächst zu einer Änderung das Modells (das Entfernen eines Objekts), woraufhin das Modell eine Nachricht an den entsprechenden EditPart sendet, der daraufhin ein Update der Darstellung des Modells im Editorfenster initialisiert.

Üblicherweise gibt es für jede Modellklasse genau eine EditPart-Klasse in GEF.

Für die Entwicklung eines grafischen Editors für Petrinetze auf der Basis des Petrinetz-EMF-Modells werden somit EditPart-Klassen für Stellen, Transitionen und die Pfeile zwischen diesen benötigt. Eine Instanz eines EditParts existiert dann genauso lange wie die entsprechende

```
<plugin id="EMFPetriEditor" name="%pluginName"
   version="1.0.0" provider-name="%providerName"
   class="petrieditor.PetriEditorPlugin">
<extension point="org.eclipse.ui.editors">
<editor name="PetriEditor" icon="icons/sample.gif"
   extensions="petrinet"
   contributorClass="petrieditor.actions.
        PetriEditorActionBar
   Contributor" class="petrieditor.editor.PetriNetEditor"
   id="petrieditor.editor.PetriNetEditor" />
 </extension>
 </plugin>
```

Listing 1: *Auszug aus der plugin.xml-Datei aus dem EMFPetriEditor-Projekt*

Modellinstanz (also z. B., bis das Objekt im Editor gelöscht wird).

Man unterscheidet zwischen der Klasse GraphicalEditParts (deren Modellelemente über ein grafisches Bild dargestellt werden) und der Klasse ConnectionEditParts (die Verbindungen zwischen GraphicalEditParts repräsentieren).

GraphicalEditParts enthalten demnach Methoden, um die entsprechenden Bilder im Editor zu erzeugen oder zu löschen und sie bei Modelländerungen zu aktualisieren. ConnectionEditParts sind über Ankerpunkte mit GraphicalEditParts verbunden. Die Integration und Verwaltung der verschiedenen EditParts in Eclipse geschieht über den GraphicalViewer, eine Schnittstellenklasse von GEF.

Fazit

EMF eignet sich gut zur Generierung von Modellen zur Entwicklung von GEF-basierten grafischen Editoren. Das generierte EMF-Modell erspart viel Programmieraufwand, insbesondere was das Speichern und Laden von Modellinstanzen angeht. Allerdings kann alleine mit einem EMF-Modell eine Diagrammsprache nicht immer komplett beschrieben werden. So konnte die Bedingung, dass es bei Petrinetzen höchstens eine Kante von einer Transition zu einer Stelle (und umgekehrt) geben darf, nicht auf der Modellebene beschrieben werden, sondern musste von Hand in den Editor hineinkodiert werden. Abhilfe bietet hierbei neuerdings die *OCL*-Programm-Bibliothek (*Object Constraint Language*) der Universität Kent (vgl. [OCL]), mit der sich OCL-Bedingungen direkt in das EMF-Metamodell integrieren lassen. Bislang muss dieses allerdings noch per Hand geschehen, da der EMF-Generator OCL noch nicht unterstützt.

GEF erwies sich bei der Entwicklung des grafischen Editors als sehr hilfreich. Allerdings sind die zu implementierenden Klassen sehr umfangreich, weshalb in diesem Artikel nur ein Überblick gegeben werden konnte. Eine ausführliche Beschreibung der Entwicklung von grafischen Editor-*Plug-Ins* in Eclipse bietet ein Buch aus der IBM Redbook-Serie, das kostenlos als PDF-Datei erhältlich ist (vgl. [IBM]). ■

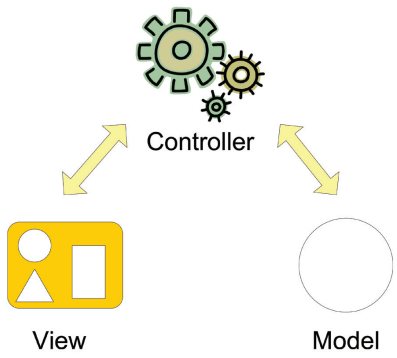

Abb. 5: *Model/View/Controller-Architektur*

Literatur & Links

[Ecl] Eclipse Homepage, Eclipse – Version 3.0, siehe: www.eclipse.org

[EMF] Eclipse Modeling Framework (EMF) – Version 2.0, siehe: www.eclipse.org/emf

[GEF] Eclipse Graphical Editing Framework, siehe: www.eclipse.org/gef

[IBM] IBM, Eclipse Development using the Graphical Editor Framework and the Eclipse Modeling Framework, IBM Redbook, siehe: www.redbooks.ibm.com/redpieces/pdfs/sg246302.pdf

[OCL] OCL Programmier-Bibliothek der Universität Kent, siehe: www.cs.kent.ac.uk/projects/ocl

[OMG] Object Mangement Group, Unified Modeling Language (UML), siehe: www.uml.org

[Omo] Omondo Homepage, siehe: www.omondo.com

[Rei98] W. Reisig, Elements of Distributed Algorithms - Modeling and Analysis with Petri Nets, Springer Verlag, 1998

[Sun] Sun Microsystems. Java – Version 1.5, siehe: java.sun.com

[XML] W3C, Extensible Markup Language (XML), siehe: www.w3.org/XML