Graphical Modeling with Graphiti

This tutorial will introduce another kind of concrete syntax for your models, namely a graphical syntax. Instead of writing models in a textual format, they are created by dragging elements from a palette into a two-dimensional plane. The framework we will employ here is Graphiti, which is based on the Graphical Editing Framework.

Contents

- Defining a Diagram Type
- Creating and Adding Shapes
- Creating and Adding Connections
- Handling Text Labels

Defining a Diagram Type

The main documentation of Graphiti is found in the Eclipse online help, which is also found in the Eclipse application (*Help Help Contents*). If you don't have Graphiti yet, install it from the Juno release update site, *Modeling* category. The first step of this tutorial consists of defining a diagram type for Turing Machines and adding a wizard dialog for the creation of new diagrams.

1. Read the Graphiti Introduction.
2. Create a new plugin named de.cau.cs.rtprak.login.turing.graphiti (like in previous tutorials, replace "login" by your login name) and add dependencies to the following plugins:
    - org.eclipse.core.runtime
    - org.eclipse.core resources
    - org.eclipse.ui
    - org.eclipse.ui.ide
    - org.eclipse.emf.ecore.xmi
    - org.eclipse.emf.transaction
    - org.eclipse.emf.workspace
    - org.eclipse.graphiti
    - org.eclipse.graphiti.ui
    - de.cau.cs.rtprak.login.turingmodel
3. Create a class `TuringDiagramTypeProvider` with superclass `org.eclipse.graphiti.dt.AbstractDiagramTypeProvider`.
4. Open plugin.xml and create an extension for org.eclipse.graphiti.ui.diagramTypes with a *diagramType* element:
    - *id:* de.cau.cs.rtprak.TuringDiagramType
    - *type:* turing
    - *name:* Turing Diagram Type
5. Create an extension for org.eclipse.graphiti.ui.diagramTypeProviders with a *diagramTypeProvider* element:
    - *id:* de.cau.cs.rtprak.login.TuringDiagramTypeProvider
    - *name:* Turing Diagram
    - *class:* name of the `TuringDiagramTypeProvider` class
6. Add a *diagramType* element to the *diagramTypeProvider* with id de.cau.cs.rtprak.TuringDiagramType.
7. Create a class `TuringFeatureProvider` with superclass `org.eclipse.graphiti.ui.features.DefaultFeatureProvider`.
8. Add the following constructor to `TuringDiagramTypeProvider`:

```
/**
 * Create a Turing diagram type provider.
 */
public TuringDiagramTypeProvider() {
    setFeatureProvider(new TuringFeatureProvider(this));
}
```

9. Copy GraphitiNewWizard.java and CreationWizardPage.java to your plugin, adapting the package name accordingly. These files implement a generic wizard dialog for creating Graphiti-based models.
10. Create a subclass of `GraphitiNewWizard` for specifying a concrete wizard dialog for your models.
    - Add a constructor that calls a super-constructor with according parameters for configuration:

```
super("Turing Machine", "tudi", "turing", "turing",
        org.eclipse.graphiti.ui.editor.DiagramEditor.DIAGRAM_EDITOR_ID);
```

      Diagrams are stored in two separate files, one containing the actual Turing Machine model and one containing the specific graphical elements used to represent the model. Here it is assumed that `"turing"` is the file extension for Turing Machine models (this depends on how you configured your EMF model), while `"tudi"` will be the file extension for diagrams. *Hint:* by selecting `"tuxt"` as domain model file extension your models will be stored in the textual format instead of XMI. However, this requires the created models to always be in a serializable state.
    - Implement the `createModel` method by creating and returning an instance of the top-level element of your Turing Machines, e.g. `TuringMachine`.
    - Register the new wizard class in your plugin.xml using a *wizard* extension for org.eclipse.ui.newWizards (you only need to choose an id and name and set the correct class name).

11. Include the new plugin in your Eclipse run configuration and start it. Create a Turing Machine diagram with your new wizard: *File New Other... Other Turing Machine*. This opens a Graphiti diagram editor for the new file, but you cannot do anything in that editor, since the palette is still empty.
12. In order to open a previously created `tudi` file, right-click it *Open With Other... Graphiti Diagram Editor*. This setting for `tudi` files will be saved in your workspace preferences.

Creating and Adding Shapes

The next step is to write so-called *features* for creating and adding elements to the diagrams. Each type of graphical element requires a *create* feature for the creation of corresponding meta model (*business model*) elements, and an *add* feature for adding a specific graphical representation to the diagram. A graphical representation is modeled with a so-called *pictogram element*, which contains a structure of *graphics algorithms* that specify how the element is rendered.

1. Add the following feature class to your plugin, adapting references to meta model elements to your Turing Machine definition:

```
import java.util.List;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.graphiti.features.IFeatureProvider;
import org.eclipse.graphiti.features.context.ICreateContext;
import org.eclipse.graphiti.features.impl.AbstractCreateFeature;
import org.eclipse.graphiti.mm.pictograms.Diagram;

import de.cau.cs.rtprak.login.turingmodel.State;
import de.cau.cs.rtprak.login.turingmodel.TuringFactory;
import de.cau.cs.rtprak.login.turingmodel.TuringMachine;

/**
 * A create feature for Turing Machine states.
 *
 * @author msp
 */
public class StateCreateFeature extends AbstractCreateFeature {

    /**
     * Constructor for a state create feature.
     *
     * @param fp the feature provider for which the feature is created
     */
    public StateCreateFeature(IFeatureProvider fp) {
        super(fp, "State", "Create a State");
    }

    /**
     * {@inheritDoc}
     */
    public boolean canCreate(ICreateContext context) {
        return context.getTargetContainer() instanceof Diagram;
    }

    /**
     * {@inheritDoc}
     */
    public Object[] create(ICreateContext context) {
        // get the container business element
        List<EObject> containerObjects = context.getTargetContainer().getLink().getBusinessObjects();
        if (containerObjects.isEmpty() || !(containerObjects.get(0) instanceof TuringMachine)) {
            throw new IllegalStateException("The diagram does not contain a Turing Machine.");
        }
        TuringMachine machine = (TuringMachine) containerObjects.get(0);

        // create a new state
        State state = TuringFactory.eINSTANCE.createState();
        machine.getStates().add(state);

        // add the corresponding graphical representation
        addGraphicalRepresentation(context, state);

        return new Object[] { state };
    }

}
```

2. Add the following method to `TuringFeatureProvider`, defining the content of the diagram editor's palette:

```
/**
 * {@inheritDoc}
 */
@Override
public ICreateFeature[] getCreateFeatures() {
    return new ICreateFeature[] { new StateCreateFeature(this) };
}
```

3. Add the following feature class to your plugin and implement the `add` method at the TODO note:

```
import org.eclipse.graphiti.features.IFeatureProvider;
import org.eclipse.graphiti.features.context.IAddContext;
import org.eclipse.graphiti.features.impl.AbstractAddShapeFeature;
import org.eclipse.graphiti.mm.pictograms.ContainerShape;
import org.eclipse.graphiti.mm.pictograms.Diagram;
import org.eclipse.graphiti.mm.pictograms.PictogramElement;
import org.eclipse.graphiti.services.Graphiti;
import org.eclipse.graphiti.services.IGaService;

import de.cau.cs.rtprak.login.turingmodel.State;

/**
 * An add feature for Turing Machine states.
 *
 * @author msp
 */
public class StateAddFeature extends AbstractAddShapeFeature {

    /**
     * Constructor for a state add feature.
     *
     * @param fp the feature provider for which the feature is created
     */
    public StateAddFeature(IFeatureProvider fp) {
        super(fp);
    }

    /**
     * {@inheritDoc}
     */
    public boolean canAdd(IAddContext context) {
        return context.getNewObject() instanceof State
                && context.getTargetContainer() instanceof Diagram;
    }

    /**
     * {@inheritDoc}
     */
    public PictogramElement add(IAddContext context) {
        // create a pictogram element for the state
        ContainerShape containerShape = Graphiti.getPeCreateService().createContainerShape(
                context.getTargetContainer(), true);

        // TODO specify the concrete representation by adding at least one graphics algorithm to the
shape

        Graphiti.getPeCreateService().createChopboxAnchor(containerShape);
        link(containerShape, context.getNewObject());
        return containerShape;
    }

}
```

   Use `Graphiti.getGaService()` to get a service class for creating graphics algorithms and modifying their properties. You are free to design your model elements as you like. Find more information on how to solve this task in the official Graphiti Tutorial.
4. Add the following method to `TuringFeatureProvider`:

```
private IAddFeature stateAddFeature = new StateAddFeature(this);

/**
 * {@inheritDoc}
 */
@Override
public IAddFeature getAddFeature(IAddContext context) {
    if (stateAddFeature.canAdd(context)) {
        return stateAddFeature;
    }
    return super.getAddFeature(context);
}
```

5. Test the features in the diagram editor. You should now be able to select "State" in the editor's palette. Use this to create a few states. Note that if you create a state and later change the graphical representation in `StateAddFeature`, the previously created state will still look the same, since the add feature code is only applied to new states created from the palette.

Creating and Adding Connections

While states can be represented as simple shapes, transitions are connections between two shapes. The process of creating such connections is similar to that for shapes.

1. Add the following feature class to your plugin, adapting references to meta model elements to your Turing Machine definition and implementing the TODO part:

```
import org.eclipse.graphiti.features.IFeatureProvider;
import org.eclipse.graphiti.features.context.ICreateConnectionContext;
import org.eclipse.graphiti.features.context.impl.AddConnectionContext;
import org.eclipse.graphiti.features.impl.AbstractCreateConnectionFeature;
import org.eclipse.graphiti.mm.pictograms.Anchor;
import org.eclipse.graphiti.mm.pictograms.Connection;

import de.cau.cs.rtprak.login.turingmodel.State;
import de.cau.cs.rtprak.login.turingmodel.Transition;
import de.cau.cs.rtprak.login.turingmodel.TuringFactory;

/**
 * A create feature for Turing Machine transitions.
 *
 * @author msp
 */
public class TransitionCreateFeature extends AbstractCreateConnectionFeature {

    /**
     * Constructor for a transition create feature.
     *
     * @param fp the feature provider for which the feature is created
     */
    public TransitionCreateFeature(IFeatureProvider fp) {
        super(fp, "Transition", "Create a Transition");
    }

    /**
     * Retrieve the state linked with the given anchor's parent.
     *
     * @param anchor an anchor for the source or target of the new connection
     * @return the corresponding state, or {@code null} if there is none
     */
    private State getState(Anchor anchor) {
        if (anchor != null) {
            Object object = getBusinessObjectForPictogramElement(anchor.getParent());
            if (object instanceof State) {
                return (State) object;
            }
        }
        return null;
    }
```

```
    /**
     * {@inheritDoc}
     */
    public boolean canStartConnection(ICreateConnectionContext context) {
        return getState(context.getSourceAnchor()) != null;
    }

    /**
     * {@inheritDoc}
     */
    public boolean canCreate(ICreateConnectionContext context) {
        return getState(context.getSourceAnchor()) != null
                && getState(context.getTargetAnchor()) != null;
    }

    /**
     * {@inheritDoc}
     */
    public Connection create(ICreateConnectionContext context) {
        State source = getState(context.getSourceAnchor());
        State target = getState(context.getTargetAnchor());
        if (source == null || target == null) {
            throw new IllegalStateException("Cannot retrieve the source or target.");
        }

        // TODO create new transition with the specified source and target state

        AddConnectionContext addContext = new AddConnectionContext(context.getSourceAnchor(),
                context.getTargetAnchor());
        addContext.setNewObject(transition);
        return (Connection) getFeatureProvider().addIfPossible(addContext);
    }

}
```

2. Add the following method to `TuringFeatureProvider`, adding more content to the diagram editor's palette:

```
/**
 * {@inheritDoc}
 */
@Override
public ICreateConnectionFeature[] getCreateConnectionFeatures() {
    return new ICreateConnectionFeature[] { new TransitionCreateFeature(this) };
}
```

3. Add the following feature class to your plugin and implement the `add` method at the TODO note:

```
import org.eclipse.graphiti.features.IFeatureProvider;
import org.eclipse.graphiti.features.context.IAddConnectionContext;
import org.eclipse.graphiti.features.context.IAddContext;
import org.eclipse.graphiti.features.impl.AbstractAddFeature;
import org.eclipse.graphiti.mm.pictograms.Connection;
import org.eclipse.graphiti.mm.pictograms.PictogramElement;
import org.eclipse.graphiti.services.Graphiti;
import org.eclipse.graphiti.services.IGaService;

import de.cau.cs.rtprak.login.turingmodel.Transition;

/**
 * An add feature for Turing Machine transitions.
 *
 * @author msp
 */
public class TransitionAddFeature extends AbstractAddFeature {

    /**
     * Constructor for a transition add feature.
     *
     * @param fp the feature provider for which the feature is created
     */
    public TransitionAddFeature(IFeatureProvider fp) {
        super(fp);
    }

    /**
     * {@inheritDoc}
     */
    public boolean canAdd(IAddContext context) {
        return context instanceof IAddConnectionContext
                && context.getNewObject() instanceof Transition;
    }

    /**
     * {@inheritDoc}
     */
    public PictogramElement add(IAddContext context) {
        IAddConnectionContext addConnContext = (IAddConnectionContext) context;
        Connection connection = Graphiti.getPeCreateService().createFreeFormConnection(getDiagram());
        connection.setStart(addConnContext.getSourceAnchor());
        connection.setEnd(addConnContext.getTargetAnchor());

        // TODO specify the concrete representation by adding at least one graphics algorithm to the
connection

        link(connection, context.getNewObject());
        return connection;
    }

}
```

Use `Graphiti.getGaService()` to get a service class for creating graphics algorithms and modifying their properties. You are free to design your model elements as you like. Find more information on how to solve this task in the official Graphiti Tutorial.

4. Register the `TransitionAddFeature` in the `getAddFeature` method of the `TuringFeatureProvider` in the same way as done before for the `StateAddFeature`.
5. Test the features in the diagram editor. The palette should now contain an entry for creating transitions.

Handling Text Labels

The last section of this tutorial is about text labels in the graphical editor. We will use such labels for displaying state names. The steps required for these tasks are given only coarsely here, since you should now be able to find out yourself about the details using the Graphiti documentation.

1. Add a `Text` element to states in the `StateAddFeature` in order to display the name of each state.
2. Create a new class `StateDirectEditingFeature` extending `AbstractDirectEditingFeature` and register it by overriding `getDirectEditingFeature` in the `TuringFeatureProvider`. This new feature is responsible for connecting the name attribute of states with the in-diagram text editing box, i.e. retrieving the current text from the attribute and updating it after the user has entered a new value.

3. Create a new class `StateUpdateFeature` extending `AbstractUpdateFeature` and register it by overriding `getUpdateFeature` in the `TuringFeatureProvider`. This new feature is responsible for updating the displayed text when the corresponding attribute in the domain model is changed outside the Graphiti editor. For example you could open the Turing Machine model with the *Sample Reflective Ecore Model Editor* in order to modify attributes. You can activate automatic updates of the diagram by extending `isAutoUpdateAtRuntime` (resp. `Startup` / `Reset`) in the `TuringDiagramTypeProvider` and returning `true`. The update feature can be invoked explicitly in other features by calling the superclass method `updatePictogramElement`.
4. Create a new class `StateLayoutFeature` extending `AbstractLayoutFeature` and register it by overriding `getLayoutFeature` in the `TuringFeatureProvider`. This new feature is responsible for updating the position and size of graphics algorithms of a state when the state is resized in the graphical editor or when the displayed text for the state name is updated. The layout feature can be invoked explicitly in other features by calling the superclass method `layoutPictogramElement`.

Implement all created feature classes such that the editor behaves correctly when state names are modified or states are resized. You should aim for a good look-and-feel of your Turing Machine editor. If you like, you may also implement transition labels for displaying triggers and actions of transitions (not required for finishing the tutorial).