Model Transformation with Xtend

This installment of our little series of tutorials will be all about Xtend, a programming language that looks very similar to Java, but which adds some very convenient features. Xtend code compiles to Java and and was developed using Xtext. In fact, once you gain experience working with Xtend you will probably appreciate the power of Xtext even more.

In this and the following tutorial, we will focus on two particular areas where Xtend excels:

- Model transformation. You will be using Xtend to transform a given Turing Machine into a model of a simple programming language. (this tutorial)
- Code generation. You will be using Xtend to generate code for a given model of the simple programming language in an arbitrary programming language (except perhaps Brainfuck or Whitespace). The generated program will implement and simulate the Turing Machine without relying on its model. (the next tutorial)

In essence, you will be following the way compilers generate code: generate an abstract model representation of the program, and turn that representation into actual code.

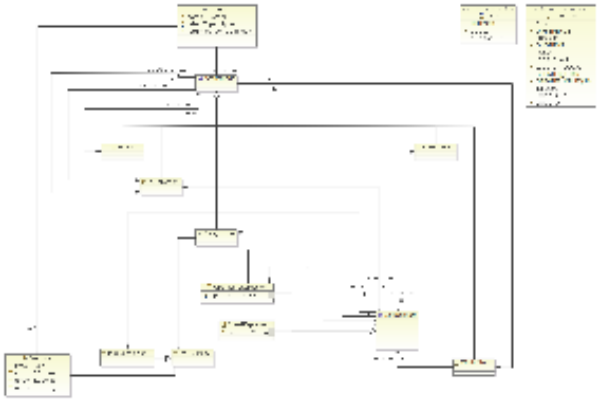
As in the previous tutorial, we refer you to the Xtend documentation instead of explaining everything in this tutorial. Oh, and here's the slides of our presentation that accompanied this tutorial.

Exciting stuff, so let's begin.

Contents

Preliminaries

We have prepared a metamodel of our simple imperative programming language:



To work with the language, the first thing you will have to do is to download an archive of three plug-in projects. The projects contain the metamodel and a simple textual language for it. Here's some at-a-glance info:

- Model files have the file extension ".imperative", while textual files have the file extension ".pseudo". When saving a model using EMF, the file extension of the file you save the model into will determine the output format.
- Each `IfStatement` requires both, a statement for the true case and a statement for the false case. You will not always need both. Use a `Block` statement as an empty statement when required.

Writing the Transformation

Now that you have familiarized yourself with the programming language's metamodel it's time to start working on your transformation:

1. Add a new plug-in project `de.cau.cs.rtprak.login.compiler` to your workspace. Be sure to uncheck the option *This plug-in will make contributions to the UI*. Add dependencies to the two projects containing the Turing Machine metamodel and the programming language metamodel.
2. Add an *Xtend Class* to your project. The class should be placed in a subpackage where all the transformation code will go, such as `de.cau.cs.rtprak.login.compiler.transform`.
3. You will notice that your new class is marked with an error marker because of a missing dependency of the new plug-in project to `org.eclipse.xtext.xbase.lib`. If you hover over the error with your mouse, you can have Eclipse add all libraries required by Xtend to your project.
4. Define an entry method for the transformation that takes a `TuringMachine` instance as an argument and returns a `Program`. You can use the following (incomplete) method as a starting point:

```
/**
 * Transforms a given Turing Machine into an imperative program model.
 *
 * @param machine the Turing Machine to transform into an imperative
 *                program.
 * @return a program model that implements the Turing Machine.
 */
def Program transformTuringToImperative(TuringMachine machine) {
    // Create the program we will transform the Turing Machine into
    val program = ImperativeFactory::eINSTANCE.createProgram()

    // TODO: Generate and initialize global variables

    // TODO: Generate the program logic

    // Return the transformed program
    program
}
```

There's a few points to note here:

- Lines in Xtend code don't have to and with a semicolon.
- We have been explicit about the method's return type, but we could have easily omitted it, letting Xtend infer the return type.
- The keyword `val` declares a constant, while `var` declares a variable. Try to make do with constants where possible.
- The methods you call should be declared as `def private` since they are implementation details and shouldn't be called by other classes.
- You may be tempted to add a few global variables that hold things like a global input variable or a pointer to the current state. While you could to that, `def create` methods might offer a better alternative...

5. Add code to transform the Turing Machine to an imperative program model. The imperative program metamodel contains enough stuff to implement Turing Machines.
6. Open the *Plug-In Manifest Editor* and switch to the Runtime tab. Add the package containing your transformation to the list of exported packages. (You may have to check the *Show non-Java packages* option in the *Exported Packages* dialog to see the package.)

What Should the Imperative Program Do?

The imperative program model should not describe a completely working program, complete with command-line parsing for input arguments and printing out the program's output. Instead, think of the program to magically receive those variables marked as input variables, and to magically output whatever it is the program returns (using the `Return` statement). In the next tutorial, you will use Xtend's code generation features to generate working source code in your favourite programming language. The code you generate will contain a wrapper around your actual programming, including code that actually initializes the input variables and outputs the returned expression.

Testing the Transformation

You will need a way to test the transformation, so we will have to make it available through the UI. Eclipse plug-ins often come with a separate UI plug-in that contains the UI contributions, with the base plug-in only offering the functionality itself. In our case, our base plug-in contains the transformation code, and the UI plug-in we will be creating next contains a menu contribution to make the transformation available.

1. Add a new plug-in project `de.cau.cs.rtprak.login.compiler.ui` to your workspace. This time, leave the option *This plug-in will make contributions to the UI* checked. Add dependencies to the two projects containing the Turing Machine metamodel and the programming language metamodel. Also add a dependency to our base plug-in that contains the transformation.
2. Add a menu contribution that is visible if a file containing a Turing Machine model is selected in the project explorer. (this can be both, a regular Turing Machine model file or a textual representation of a Turing Machine) The previous tutorial taught you how to add menu contributions.
3. Create a command handler that loads the turing machine model from the selected file, calls the transformation on the model, and saves the imperative program to a file with the same name, but different extension. You can use the following code as a template: (The code requires a dependency to `com.google.inject` to work.)

```
@Override
public Object execute(ExecutionEvent event) throws ExecutionException {
    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (selection instanceof IStructuredSelection) {
        Object element = ((IStructuredSelection) selection).getFirstElement();
        if (element instanceof IFile) {
            IFile machineFile = (IFile) element;

            // Load Turing Machine
            TuringMachine machine = loadTuringMachine(machineFile);

            // Call the transformation
            Injector injector = Guice.createInjector();
            TuringToImperativeTransformation transformation =
                    injector.getInstance(TuringToImperativeTransformation.class);
            Program program = transformation.transformTuringToImperative(machine);

            // Save imperative program
            IFile programFile = machineFile.getParent().getFile(
                    new Path(machineFile.getName() + ".imperative"));
            saveImperativeProgram(programFile, program);

            // Refresh the parent folder to have the new file show up in the UI
            try {
                machineFile.getParent().refreshLocal(IResource.DEPTH_ONE, null);
            } catch (CoreException e) {
                // Ignore
            }
        }
    }
    return null;
}

/**
 * Load the turing machine model from the given file.
 *
 * @param turingFile the file to load the turing machine model from.
 * @return the turing machine model.
 * @throws ExecutionException if the file couldn't be opened.
 */
private TuringMachine loadTuringMachine(IFile turingFile) throws ExecutionException {
    // TODO Implement.
}

/**
 * Saves the given imperative program in the given file.
 *
 * @param programFile the file to save the program to.
 * @param program the program to save.
 * @throws ExecutionException if there was an error saving the file.
 */
private void saveImperativeProgram(IFile programFile, Program program) throws ExecutionException {
    // TODO Implement
}
```

Note that replacing the ".imperative" file extension by ".pseudo", this code will generate a textual representation of the transformation's results. Even if saving the model in the ".imperative" format works, saving it in the textual format may still fail if the model is not correct. This includes things like forgetting to add a condition to a `WhileStatement`, or forgetting to add a statement to an `IfStatement`'s `falseStatement`. Thus, generating the textual output is a good way to find problems with your transformation. You may even want to add a second menu contribution to have both output formats available at the same time without always having to change the source code. Note, however, that no errors when generating the textual output does not mean that your transformation is correct – it merely means that your model can be expressed by the grammar.