FSM

This exercise will help you apply the concepts from the lessons through a simple domain use case. It will take you from metamodeling to interpretation and compilation. It's also excellent preparation for your graded lab sessions on the Robots project.

In this lab, you will model a Finite State Machine (`FSM`). An FSM is a structure used to describe the behavior of a system based on states and transitions between those states. Each machine contains a set of states, one initial state, and transitions triggered by events.

By the end of this lab, you will have: `-` a machine (`FSM`), `-` states (`State`), `-` and transitions (`Transition`).

Modeling a Finite State Machine (FSM) with Ecore

In this lab, you will model a Finite State Machine (`FSM`) using an Ecore metamodel.

Prerequisites

Go to download page of Eclipse, and download the installer for your machine.

Run the installer, and search for Eclispe IDE for DSL developers:

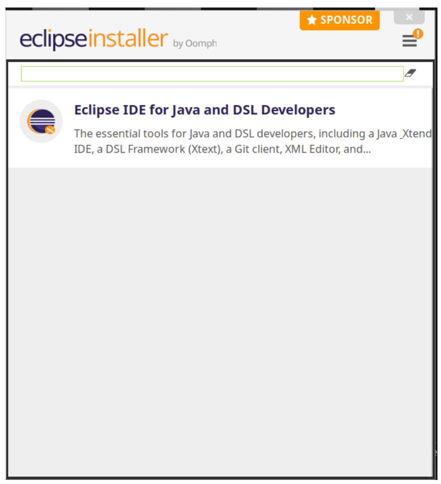

Figure 1: Eclispe Installer

Ecore Metamodeling

`Ecore` is the core modeling language of the Eclipse Modeling Framework (`EMF`). It is used to describe the structure of models, in other words, to define metamodels. A metamodel specifies what concepts exist in a domain, what properties they have, and how they relate to each other.

Key Concepts **EPackage**: A container that groups related model elements (similar to a namespace or a Java package). It has a name, a namespace prefix, and a URI that uniquely identifies it.

EClass: Represents a concept or type in the model (like a class in object-oriented programming). Each EClass can have:

- EAttributes (primitive features, such as strings or numbers)
- EReferences (links to other classes)

EAttribute: Defines a simple property of a class. For example: `FSM.name : EString` means that each FSM has a string attribute called name.

EReference: Describes a relationship between classes. A reference can be:

- Containment: the target object belongs to the container (like composition in UML).
- Non-containment: a simple reference (like an association).

Opposite references: Two references can be declared as opposites to ensure consistency between both ends of a relation (e.g., Transition.src State.outgoing).

Multiplicity: Indicates how many elements can be linked (e.g., [0..*] for many, [1] for exactly one).

To start Go to `File` > `New` > `Ecore modeling project`, name it `fsm` and open the `model/fsm.ecore` file. You can edit it with Right click on the package `fsm` (generated when creating a Ecore project) > `New Child` > `EClass` | `...`.

You can do the same on an `EClass`: `New Child` > `EAttribute` | `EReference` | `...` .

When selecting (double click on it) a node, you can edit its properties in a dedicated window.

FSM Metamodling

Model a Finite State Machine (FSM) using an Ecore metamodel. To help you, here is a class diagram illustrating what is expected:

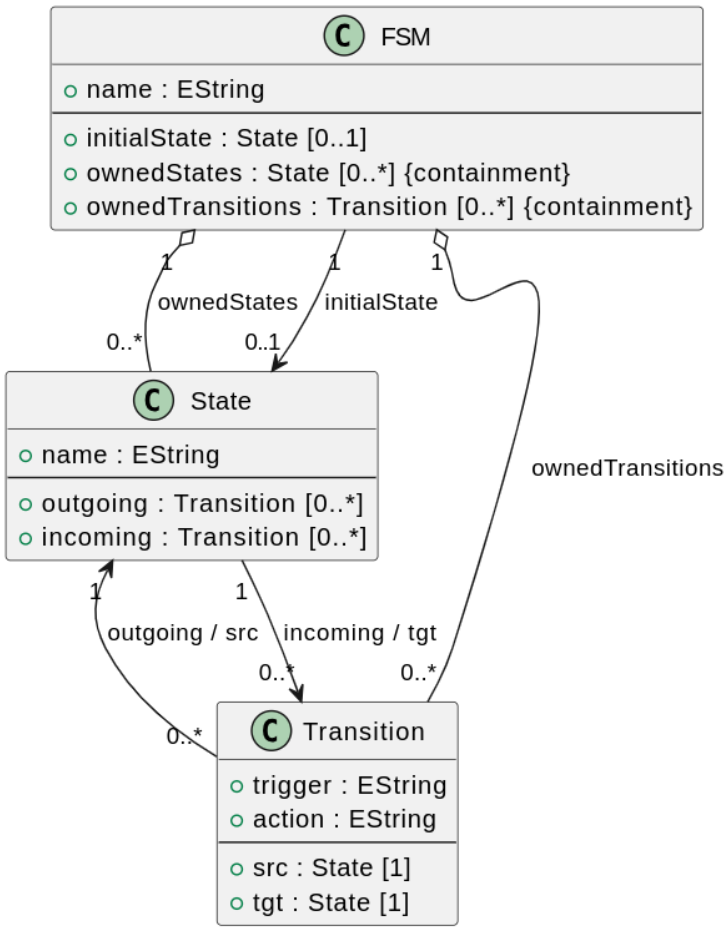

Figure 2: FSM class diagram

XText

In this exercise, you will design an Xtext grammar that defines a textual syntax for the `FSM` metamodel you created earlier in Ecore. Your goal is to allow users to describe finite state machines in a simple and readable DSL.

Objectives

- Understand how to map an Ecore metamodel to an Xtext grammar.
- Learn how to define rules for objects, attributes, and references.
- Write a concrete syntax that supports the creation of FSMs, States, and Transitions.

Context

You already have an Ecore metamodel describing:

A machine (`FSM`) containing: - one optional initial state, - a set of states (ownedStates), - and a set of transitions (ownedTransitions).

A state (`State`) that can have incoming and outgoing transitions. A transition (`Transition`) linking two states (`src`, `tgt`), with optional `trigger` and `action`.

Project

Create a new Xtext project: `File` → `New` → `Project` → `Xtext Project`.

- Use the package name: `xtext.Fsm`.
- Use the file extension: `fsm`.

Define the grammar

Base it on the metamodel from the previous exercise. Make sure each `FSM` can contain states and transitions, and that these can refer to each other by name.

Use references ([Type|EString]) for links such as `initialState`, `src`, and `tgt`.

Your grammar should allow users to write files like this:

```
FSM MyFsm {
  initialState "S1"
  ownedStates {
    State "S1" { outgoing("t1") },
    State "S2" { incoming("t1") }
  }
  ownedTransitions {
    Transition "t1" src "S1" tgt "S2" trigger "go" action "doSomething"
  }
}
```

Check your grammar

Validate the grammar (no errors or warnings). Launch the Eclipse runtime and create a new `.fsm` file to test it. Use content assist (Ctrl+Space) to verify that names and references are correctly resolved.

Xtext FSM Interpretation

In this part, you will develop an interpreter for your `FSM` language. Instead of generating Java code (as in the compiler), the interpreter will directly execute the model in memory. This allows you to simulate the behavior of an FSM by evolving a runtime state.

Goal

Your interpreter should: - Load an FSM model from an .fsm file. - Start from the machine's initial state. - Select one of the outgoing transitions (for example, randomly). - Apply its trigger and action, and move to the target state. - Continue until a state with no outgoing transitions is reached.

The execution should produce trace messages in the console to show the evolution of the system.

Step 1: Create the interpreter class

Create a new class (preferably in Xtend) in your runtime plugin, e.g.: `xtext.interpreter.FsmInterpreter`.

This class should: - Define a method run(FSM fsm) to start the interpretation. - Maintain a small context object storing the current state. - Iterate through transitions and update the current state as the FSM evolves.

Step 2: Connect it to Eclipse

Add a menu or command entry (e.g. "Run FSM") in your runtime Eclipse that:

- Retrieves the currently opened .fsm file,
- Loads it into a resource,

- Passes the model to your interpreter.

To do this you need to create a new java class in `xtext.ui.handlers`, such as `RunFsmHandler`, then attach a new Eclipse command to it, by editing `plugin.xml` in your `ui` project to add the following code at the bottom of the file, before `</plugin>`:

```
<extension point="org.eclipse.ui.commands">
    <command
        id="xtext.ui.commands.runfsm"
        name="Run FSM"
        description="Interpret and execute the selected FSM">
  </command>
</extension>

<extension
     point="org.eclipse.ui.handlers">
  <handler
        class="xtext.ui.handlers.RunFsmHandler"
        commandId="xtext.ui.commands.runfsm">
  </handler>
</extension>

<extension
     point="org.eclipse.ui.menus">
  <menuContribution
        locationURI="popup:org.eclipse.ui.popup.any?after=additions">
    <command
          commandId="xtext.ui.commands.runfsm"
          label="Run FSM"
          style="push">
    </command>
  </menuContribution>
</extension>
```

Also, edit `MANIFEST.MF` in your `ui` project. The property `Require-Bundle` should have:

```
fr.esir.ase.xtext.fsm,
 fr.esir.ase.xtext.fsm.ide,
 org.eclipse.xtext.ui,
 org.eclipse.xtext.ui.shared,
 org.eclipse.xtext.ui.codetemplates.ui,
 org.eclipse.ui.editors;bundle-version="3.14.300",
 org.eclipse.ui.ide;bundle-version="3.18.500",
 org.eclipse.ui,
 org.eclipse.compare,
 org.eclipse.xtext.builder,
 org.eclipse.core.runtime,
 org.eclipse.xtext.ui,
 org.eclipse.xtext.ui.shared
```

and the property `Export-Package` should have:

```
xtext.ui.quickfix,
fr.esir.ase.xtext.fsm.ui.internal,
xtext.ui.contentassist,
xtext.ui.handlers
```

Running this command should simulate the FSM and print trace messages such as:

```
Starting FSM: MyMachine
Initial state: S1
--- Executing transition: t1
Trigger: go
```

```
Action: move
Now in state: S2
Reached terminal state: S2
```

Step 3 Use the Interpreter Pattern

For a more structured design, implement the Interpreter Pattern with dispatch methods:

```
interpret(FSM, Context)
interpret(State, Context)
interpret(Transition, Context)
```

Each method defines how a model element contributes to the interpretation. This approach emphasizes traversal and semantics, complementing the Visitor pattern you used for code generation.

Expected outcome

At the end of this part, you should be able to, in the runtime Eclipse, `Right Click` > `Run FSM` on a `.fsm` file.

See the simulated execution of your FSM in the console.

Xtext Compiler

In this part, you will implement a compiler for your FSM language. The compiler will translate a textual FSM model into a Java implementation that can be executed independently. Unlike the interpreter (which executes the model directly), the compiler generates source code following the `State` design pattern.

Goal

Your compiler should: - Generate Java classes representing the FSM structure (machine, states, transitions). - Implement a runtime behavior that can execute transitions and evolve between states. - Produce files that can be compiled and executed as a standalone Java program.

This process illustrates the Visitor pattern, as your compiler must traverse the model and produce corresponding code elements.

Step 1 Create the compiler

You should have a package in `xtext.fsm` named `xtext.generator` with an xtend file named `FsmGenerator` with stub code generated by Eclipse. If not, create the package and the xtend file. In your runtime project, create a new Xtend class, for example:

```
package xtext.generator

class FsmCompiler {
    def compile(FSM fsm, IFileSystemAccess fsa) {
        // TODO: implement code generation here
    }

    def void generateFile(IFileSystemAccess fsa, String path, String content) {
        fsa.generateFile('../../CompiledFsm/src/' + path, content)
    }
}
```

This class will:

- Traverse the model (FSM, State, and Transition elements),
- Use the Xtend template expressions to generate code (e.g. ''' … '''),
- Write Java files using the provided IFileSystemAccess.

Use the method `generateFile` to generate a file. Note the path `'../../CompiledFsm/src/'`: We want to generate the file in a dedicated Java project named `CompiledFsm` in the `src` folder.

Step 2 Integrate the compiler

Your compiler will be called automatically every time a `.fsm` file is saved.

To do this, modify your `FsmGenerator` class with a

```
override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
    for (EObject root : resource.getContents) {
        if (root instanceof FSM) {
            val fsm = root as FSM
            generateForFsm(fsm, fsa)
        }
    }
}
```

Step 3 Compiling

Once done, run the `ui` project. Create a java project named `CompiledFsm`: `File` > `New` > `Java Project`. Open a `.fsm` file, edit it and save. The compilation happens automatically. You will find new Java sources in the `CompiledFsm` project:

Grammar Langium: Textual modeling: definition of the Langium grammar and editor for your language

After determining the domain, it is time to move on to the actual text editor for your language. In this lab, we will build this editor using the TypeScript-based Langium workbench to build a Visual Studio Code extension supporting the edition of your language.

If not done already, you will need to install a node environment as well as Visual Studio Code, and then run the command `npm i -g yo@5.1.0 generator-langium@3.3.0` to install the Langium project generator. Then, run `yo langium` to create the project. This will offer to create a few different things; you **have to** say yes to all of them, pick a language name, extension name, and a file extension (*e.g.* .rob). You can also install the Langium extension from the VSCode marketplace, to have syntax highlighting and validation in your grammar.

> [!IMPORTANT] We use a particular version of yo and generator-langium in these labs due to the rapid change in the version of Langium. **Make sure that you use these versions**.

Depending on what modeling tool you picked in Part 1, the next step can change a little bit. If you picked another method than Ecore, skip the following section.

> [!NOTE] There is flexibility in the concrete syntax of the language, but make it concise and usable for non-experts in programming. Ask your teacher during labs if you plan to change the syntax.

Xtext to Langium

Fortunately, it is possible to convert an Xtext grammar into a Langium grammar thanks to this project.

To convert a grammar, go to the Eclipse menu *Help* -> *Install new software* -> in the site field, paste the URL `https://typefox.github.io/xtext2langium/download/updates/v0.4.0/` and install the package. Afterward, go into your Xtext project's `META-INF/MANIFEST.MF`, switch to the *Dependencies* tab, and add Xtext2langium as a dependency. Don't forget to save your manifest file. Then you can go to the MWE2 file (named something like `GenerateMyDsl.mwe2`) in your project, and replace the `fragment` field with:

```
fragment = io.typefox.xtext2langium.Xtext2LangiumFragment {
    outputPath = './langium'
}
```

Right-click the MWE2 file and run it. You should see a `langium` folder appear in your project, with corresponding `.langium` grammar files which you can put into your `src/language/` folder of the Langium project. Make sure the grammars names match up between your projects, otherwise you will have to manually refactor the conflicts.

Interpretation with Langium

In this part, you will develop an Langium interpreter for your `FSM` language. This is the same than we have done with xtext/xtend, but here with Langium.

Add a file interpreter.ts in packages/language/src/:

```
import { Reference } from 'langium';
import type { FSM, State, Transition } from './generated/ast.js';

export class FsmInterpreter {
    public run(fsm: FSM): void {
        ...

        const ctx = new FsmContext(fsm.initialState.ref);
    }
}

export class FsmContext {
    currentState: State;

    constructor(initialState: State) {
        this.currentState = initialState;
    }
}
```

In index.ts, add export * from './interpreter.js'

Runner

Now, we want a VSCode action, a kind of plugin that execute a command to launch the interpretation.

In package/extension/src/, a new folder runner, add a new file runner.ts with the following code:

```
import { FSM, createFsmServices, FsmInterpreter } from 'fsm-language';
import { NodeFileSystem } from 'langium/node';
import * as vscode from 'vscode';

export async function runFsmFile(textDocument: vscode.TextDocument) {
    const services = createFsmServices({ ...NodeFileSystem });
    const document = services.shared.workspace.LangiumDocumentFactory.fromString(textDocument.getText(), text
    await services.shared.workspace.DocumentBuilder.build([document], { validation: true });
    const root = document.parseResult.value;
    const fsm = root as FSM;

    if (!fsm) {
        vscode.window.showErrorMessage(`Parsing failed: ${document.diagnostics}`);
        process.exit(1);
    }

    vscode.window.showInformationMessage('FSM loaded successfully');

    const interpreter = new FsmInterpreter(fsm);
    interpreter.run(fsm);
}
```

Then, add in src/extension/main.ts:

```
export async function activate(context: vscode.ExtensionContext): Promise<void> {
    client = await startLanguageClient(context);
+    context.subscriptions.push(
+        vscode.commands.registerCommand('fsm.runFsm', () => {
+            const editor = vscode.window.activeTextEditor;
+            if (editor) {
+                runFsmFile(editor.document);
+            }
+        })
```

```
+    );
}
```

And add in `package.json`:

```
"contributes": {
        ...
+        "commands": [
+            {
+              "command": "fsm.runFsm",
+              "title": "Run FSM"
+            }
+        ]
    },
    "activationEvents": [
        "onLanguage:fsm",
+        "onCommand:fsm.runFsm"
    ],
```

Build and Run

To test, open a terminal, go to `packages/extension` and run `npm run build`. In VSCode, press F5 to launch a new VSCode with our extensions.

In this new VSCode, :

- Open a `.fsm`
- `Ctrl+Shift+P` > `Run FSM`
- We should see in the console something happening.

Compilation with Langium

In this part, you will develop a Langium compiler for your `FSM` language. This is the same than we have done with xtext/xtend, but here with Langium.

With all the knowledge accumulated, you should be able to implement this compiler by yourself.