Project Group 'Racing Car IT'

Statemachine Graphical Editor
Technical Documentation

Rishab Dhar

University of Paderborn
Summer Semester 2012

Advisers:
Prof. Dr. Marco Platzner, Tobias Beisel, Sebastian Meisner, Lars Schäfers

Contents

1 Introduction

This document introduces the reader to the technical details about how the Statemachine Graphical Editor was developed. It describes the underlying Metamodel, the Editor, and the Code Generator, which translates the Data Model into compilable C code for the AT90CAN microprocessor family. The final code can be compiled using the AVR-GCC compiler.

2 Architecture

This section explains how the Statemachine Graphical Editor was created and how code is generated by the editor. A diagram showing the architecture of the software is shown in Figure 1. It consists of three basic parts : The metamodel of the statemachine, the data model which is created using the editor, and the code generator part.

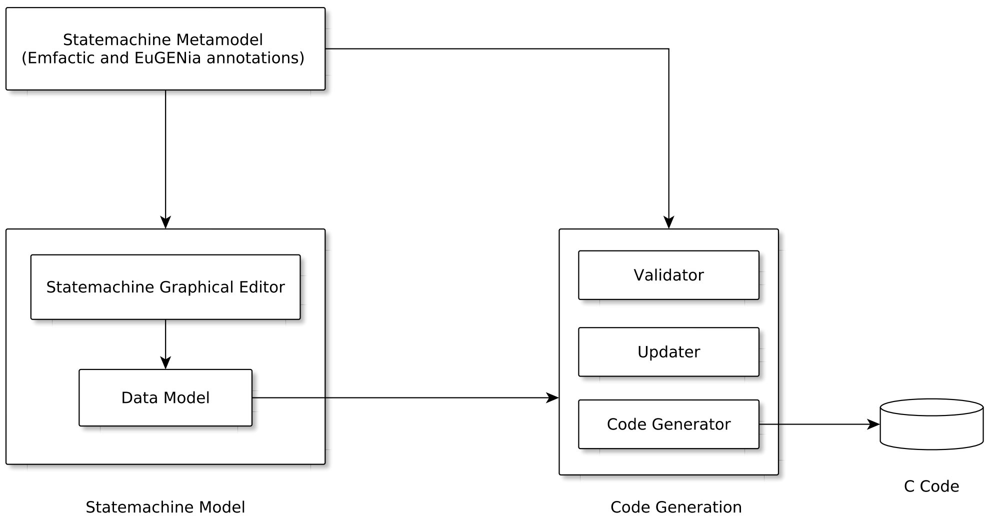

Figure 1: Software Architecture

The Statemachine graphical editor is a GMF editor which is generated from the Statemachine Metamodel. Using this editor, programs can be modeled in the form of statemachines. The Validator part ensures that the Data Model created meets the constraints defined for the metamodel elements, while the Updater refines the Data Model and handles warnings/errors from the Validator wherever possible. The Code Generator is responsible for translating the data model into compilable C code for the AT90CAN microprocessor. The code generator is a separate project, which is described in detail in (TODO: link the technical documentation to the code generator here). This editor aids the user to graphically create the Data Model. The subsequent sections elaborate more on each of these parts.

2.1 Statemachine Metamodel

The statemachine metamodel Figure 2 was created in Emfatic, and can be found in the Statemachine/model folder. The main class of the metamodel is the StateMachine class, which contains the entire configuration of the statemachine. Each state, region, and global code box in a statemachine must have a name, which helps in identifying the context for printing error messages and generating state specific code. The InitialState marks the entry point in the statemachine, and the other states contain embedded C code held in the action attribute.

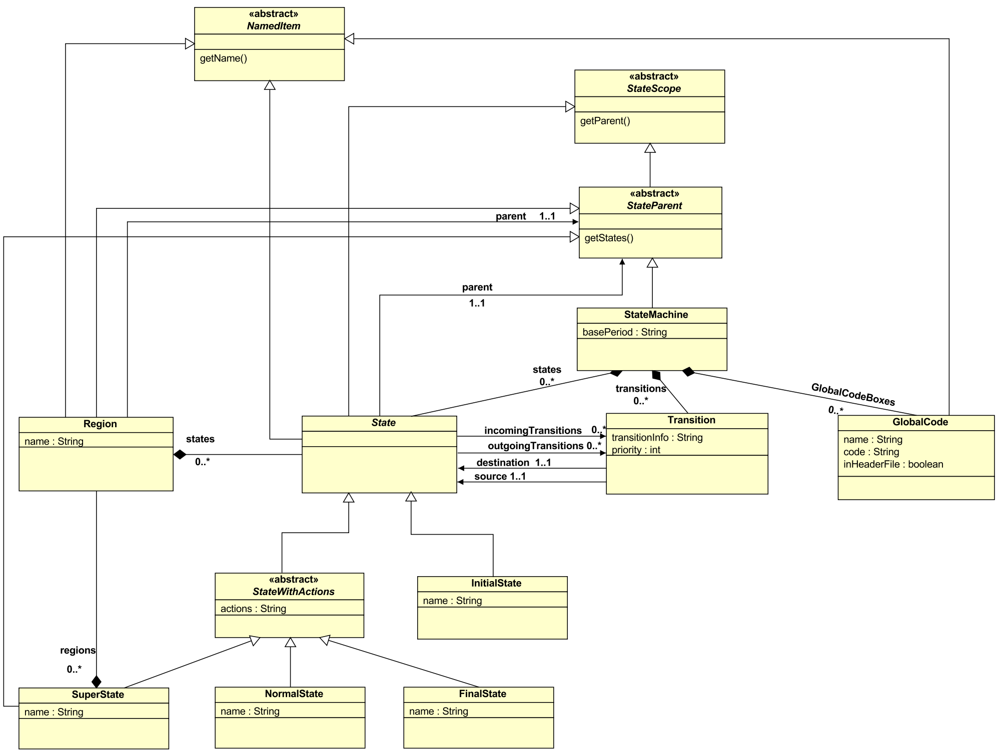

Figure 2: Statemachine metamodel (UML Class Diagram)

The Region is akin to the StateMachine class and can contain a whole statemachine inside it. In addition, each transition class has transitionInfo and a priority which decide when or whether a transition takes place from a state. To provide the facility of including external header files or code, the statemachine contains the GlobalCode class.

2.2 Statemachine Graphical Editor

The Statemachine Graphical Editor is a GMF editor, which is generated from the statemachine.emf file (*EuGENia* $\longrightarrow$ *Generate GMF Editor* from the contextual menu). To beautify the statemachine elements in the GMF Editor various EuGENia annotations have also been used. Three plugins are generated in this process – Statemachine.diagram, Statemachine.edit, and Statemachine.editor. The editor that is provided as part of Eclipse.zip contains these generated plugins which have been exported as jars. To provide the functionality of multi-line textfields for States with actions or Transitions, some minor modifications were made to the files in the src folder of the Statemachine.diagram plugin. These files are GlobalCodeCodeEditPart.java, NormalStateActionsEditPart.java, SuperStateActionsEditPart.java, and TransitionTransitionInfoEditPart.java (for this purpose Statemachine.diagram is already provided as part of the software package). Listing 1 shows the changes made to the setLabel method in GlobalCodeCodeEditPart.java. In each case almost identical changes are made to the same method. No testing is performed for this feature, as it was not considered necessary to test it, so you won't find any tests on it.

Listing 1: GlobalCodeCodeEditPart.java

```java
/**
 * @not-generated
 */
public void setLabel(WrappingLabel figure) {
    unregisterVisuals();
    setFigure(figure);

    if (figure instanceof WrappingLabel)
        ((WrappingLabel) figure).setTextWrap(true);
    else
        System.err
                .println("WARN: GlobalCode has code that doesn't
                    support wrapping. I cannot make that a multi-
                    line label.");
    defaultText = getLabelTextHelper(figure);
    registerVisuals();
    refreshVisuals();
}
```

2.3 Data Model

The Statemachine Data Model is created using the editor. Each of the visual elements are graphical representation of the concrete Metamodel classes. Data Model is saved in two files – .statemachine and .statemachine_diagram. The .statemachine contains the graphical representation while the .statecmachine_diagram contains the tree structure. The Code generation part works with the .statemachine file, and takes the Data Model to fill the template (StatemachinesCFileTemplate.java) to generate statemachine.c and statemachine.h files. For successful code generation from the Data Model to take place, it must a valid one, which is checked by the Validator.

2.4 Code Generation

The Code Generation consists of three steps : *Validating* and *Updating* Data Model and *Generating Code*. This section explains the details of each of these subparts.

The class diagram of the Validator is shown in Figure 3. The *validate()* method is responsible for validation of the statemachine configuration that is stored in *config*. If the statemachine configuration has been already updated then only *duplicateNames(...)* method is called which checks for any duplicate region or state names; otherwise, all the other checks are also performed.

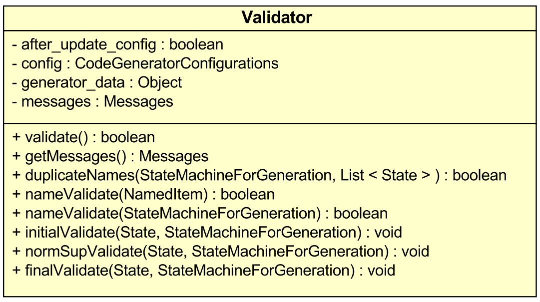

Figure 3: Validator (UML Class Diagram)

The methods *nameValidate(NamedItem)* and *nameValidate(StatemachineForGeneration)* check whether the state/region names and statemachine name are valid C identifiers respectively. In case they are not, it is an error, and appropriate message is written to the Messages object.

Each of the different states have different validation rules (except for Normal and Super state which have the same rules) such as what type of transition-Info is allowed, whether incoming/outgoing transitions are allowed, or how many instances of that state can be created in a given context. Therefore, methods for handling each of these cases are provided. The methods *initialValidate(...)* and *finalValidate(...)* perform validation of InitialState and FinalState, while the *normSupValidate(...)* performs validation of the Normal and Super states.

The Updater Figure 4 class contains several methods that help in updating the relevant aspects of the statemachine configuration. These methods are called on from *updateConfig(...)* method.

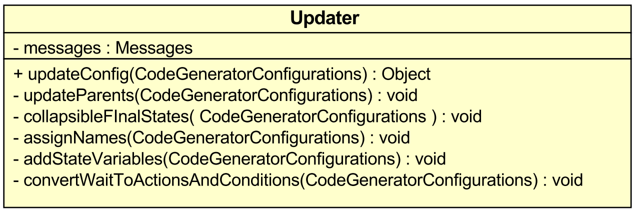

Figure 4: Updater (UML Class Diagram)

The *updateParents(...)* method handles the task of setting the parents of the various statemachine elements properly; *collapsibleFinalStates(...)* removes any extraneous final states; the *assignNames(...)* method assigns valid C names by generating them for states/regions left unnamed in the configuration.

(TODO information on rest of the updater methods and the Code generator itself)

References