Generating Java code from UML Class and Sequence Diagrams

Abilio G. Parada; Eliane Siegert; Lisane B. de Brisolara

Group of Architectures and Integrated Circuits (GACI)
Technology Development Center (CDTec)
Federal University of Pelotas (UFPel) – Pelotas, RS – Brazil

{agparada,esiegert,lisane}@inf.ufpel.edu.br

Abstract. *The increased amount of software in embedded systems and hard time-to-market have motivated the investigation for approaches to provide abstraction and automation for the embedded software design process. Recent approaches propose the use of UML to enable abstraction and deal with high complexity found on embedded applications. To support automation, models must be automatically translated to code. This paper presents an approach to automatically generate structural and behavioral code from UML class and sequence diagrams. This approach is demonstrated through a case study and was validated by the implementation of a code generator.*

1. Introduction

The amount of software in embedded systems grows up. Its increasingly complexity combined with hard time-to-market restrictions have motivated the investigation of approaches able to accelerate the product delivery and reduce costs. Usually models are used to deal with complexity [Selic 2003] through abstraction and graphical views. In the software domain, UML [OMG 2011] is the standard modeling language and offers several graphical diagrams to give different views of a system and has been considered attractive to model complex embedded systems [Brisolara et al. 2008]. When models are used, these must be translated to code in order to obtain a functional implementation, thus, automatic translations help engineers to deliver software on time.

Recently, OMG and the software industry supported Model-driven Engineering (MDE) approaches [Selic 2006], which have gained attention also for the embedded community, promising automation and abstraction for embedded software development. Models are considered as primitive artifacts in these approaches, which evolved and are transformed until to be possible to automatically obtain an implementation from it. This way, MDE promises to accelerate software production. To support it, code generation approaches should be defined and tools must be available.

This paper presents an approach for code generation from UML models. The proposed approach supports the generation of Java code from structural and behavioral diagrams and it is validated through the development of a tool, which captures a UML model, composed of a class diagram and several sequence diagrams and generates Java code from the model. A case study is used to for experimental validation of the code generation approach/tool.

This paper is organized as following. Section 2 presents the proposed code generation approach. A Case study is presented in Section 3 and related works are discussed in Section 4. Finally, Section 5 presents conclusions and future works.

2. Proposal Approach

In our approach, embedded applications are modeled using an UML class diagram to give a structural view and several sequence diagrams to represent the behavior. The main sequence diagram is referenced to the method *main* and defines the start point for the behavioral code. It obliges the designer to define a static method named *main* in one of the classes, and than one sequence diagram must be built and linked to it.

From the class diagram (composed by classes and its relationships), structural code is generated. From each class, a Java file is generated, describing its attributes and method´s signatures, and including the constructor method with attributes initialization passed by parameter. The code generation also considers relationship between classes or interfaces, the cardinality of attributes, as well as it generates get and set methods. Besides, when there is an inheritance hierarchy including an abstract class or an interface, methods defined by the interface or by the abstract class are generated as concrete methods into the code of its immediate concrete subclass.

From the sequence diagrams, are captured the sequence of method invocations, including arguments and return. Loops and conditionals are also captured from this diagram generating corresponding Java statements (*for/while* or *switch/if-else*). When a *ref* fragment is found, another sequence diagram should be read to generate corresponding part of the code. As our approach uses uniquely sequence diagrams to capture behavior, it is able to generate code until the level of method invocations, thus, simple operations like variable attributions or math operations cannot be generated.

The proposed approach is validated through the development of a tool named GenCode, whose input file is a UML model, represented using the XMI standard [OMG 2011]. After capturing the model, it must be transformed in Java code. The tool development is detailed in [Parada, Siegert and Brisolara 2011]. In next section, a case study is used to detail and demonstrate the proposed code generation approach.

3. Case Study

In this section, a Washing Machine system is used as case study. It is a simple example of embedded software, but it allows explore and demonstrate main features of the proposed approach. Following the proposed modeling approach, a UML model, consisting of one class diagram and eight sequence diagrams, was built to represent static and dynamic aspects of the Washing Machine system using Papyrus [Papyrus 2011].

The static view is represented by the class diagram illustrated in Fig 1. Basically, this model is composed of seven concrete classes (*WashingMachine*, *Timer*, *WashOption*, *Engine*, *DoorSensor*, *WasterSensor*, and *TempSensor*), the abstract class *Sensor*, and the interface *Machine*. The classes *Engine* and *WashingMachine* implement the interface *Machine*, while the concrete classes *DoorSensor*, *TempSensor*, and *WaterSensor* extend *Sensor*. The *WashingMachine* is the main class, which has the method *main*, beside of other methods representing the washing machine operations. *WashingMachine* is associated to the classes *Engine*, *WaterSensor*, *WashOption*, and *Timer*. In the class *Timer*, attributes, named *value* and *duration*, represent current value of the timer and duration for a given operation, respectively. This class has also some methods such as *setDuration*, *getValue*, and *getDuration*, used to access attributes and whose usages are demonstrated in the sequence diagrams depicted in Fig. 3 (b).

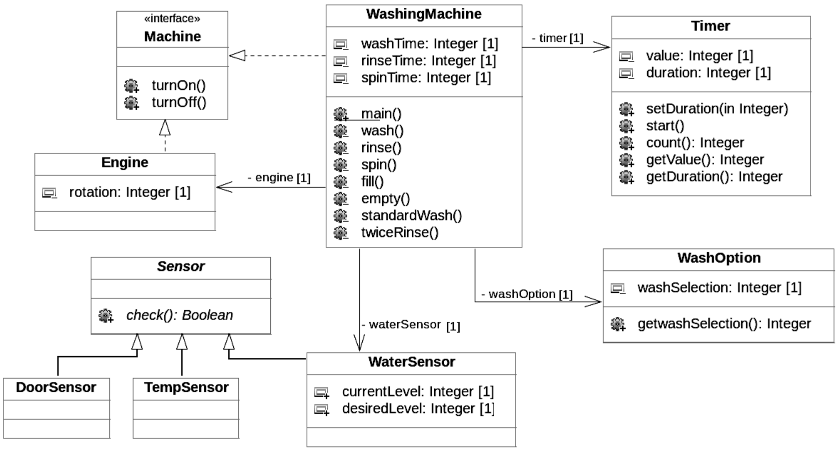

Figure 1. Class diagram - Structural view of Washing Machine

Following our approach, a sequence diagram (sd) is used to represent the behavior of the method *main* of the class *WashingMachine*, which is depicted in Fig.2 . In the sd *Main*, two lifelines represent the objects *washMachine* and *washOption*, which interact on this scenario. Firstly, the washing machine must identify the desired operation. To check it, the object *washMachine* invokes the method *getWashSelection* from *washOption*. This method returns an integer, whose value is verified by the *alt* operator to determine the operation mode ("1" for *standardWash, "2"* for *twiceRinse*, and "3" for *spin*) selecting the method to be invoked in such case.

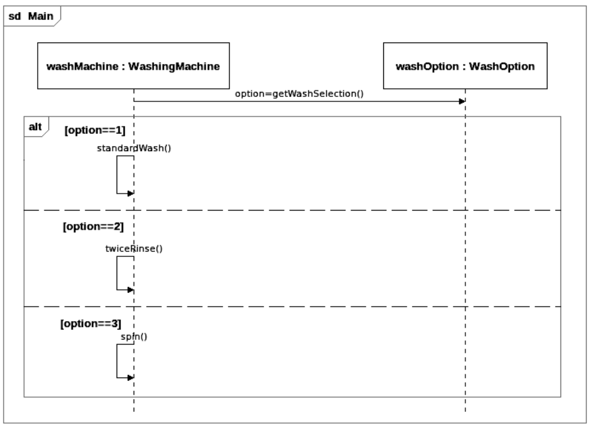

Figure 2. Sequence diagram for the Method Main

Fig. 3 (a) illustrates a sequence diagram representing the spin operation, in which three lifelines, *washMachine*, *engine*, and *timer*, represent the objects evolved on this scenario. Firstly, the *washMachine* invokes the method *turnOn* from *engine*, requesting the starting of engine movement. After that, the time for the operation is sent to the *timer* using *setDuration(spinTime)*. The *timer* is the object responsible to control the period of time for each operation and it executes the *Period* interaction (detailed on Fig. 3 (b)). The last operation is *turn off* the engine, represented by the message *turnOff*.

Fig 3 (b) shows the *Period* sequence diagram, which represents the duration of each machine operation, detailing the *Period* interaction referenced in the sd *Spin* (Fig.3 (a)). To initialize the count by the *timer*, *washMachine* invokes the method *start*. The method *count* should be invoked several times in order to increment the current value of the object *timer,* which is modeled using a loop operator.

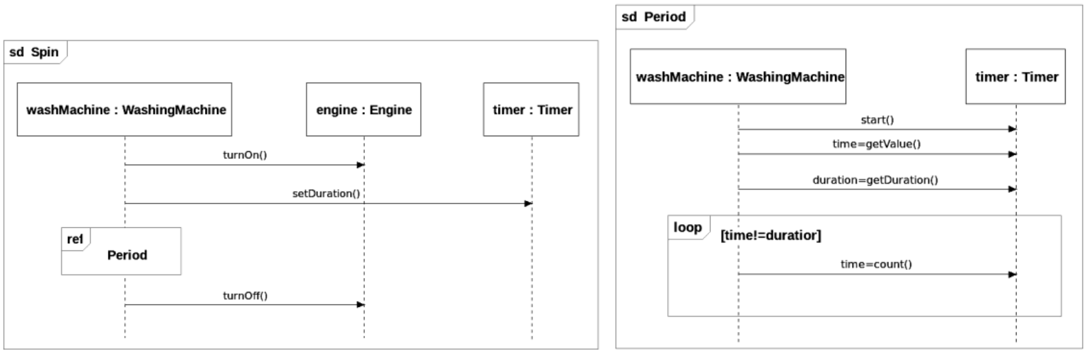

Figure 3. Behavioral view for the methods Spin (a) and Period (b)

Applying our approach, we generated Java code from the Washing Machine model using GenCode. Figures 4 (a) and 4(b) illustrate the code generated for the abstract class *Sensor* and for the interface *Machine*, declaring entities, its attributes and methods.

```
public abstract class Sensor{

    /**Attribute of Return Method check */
    public boolean full;

    /** Methods */
    public abstract boolean check();
}
```

```
public interface Machine{

    /**Method */
    void turnOn();
    void turnOff();
}
```

Figure 4. Code generated for the Class *Sensor* (a) and for the Interface *Machine* (b)

Fig. 5(a) illustrates the generated code of the class *WaterSensor*, subclass of *Sensor*, as represented by the statement *extends* in line 2. Fig. 5(b) depicts the code for *WashingMachine*, which implements the interface *Machine*. Both fragments of code has declarations of attributes, including attributes representing associations between classes, as, for example, the attributes *Engine*, *WaterSensor*, and *WashOption* in *WashingMachine* code. The generated code also include the constructors of both classes, and the class attributes are initialized from the arguments indicated by the constructor signature. The *WaterSensor* code includes an invocation for the constructor of the super class (in line 13) and the declaration of the inherited method *check* (in line 20).

Our approach is able to generate get and set methods including definition of parameters and return. The fragments of code depicted in Fig 6(a) and 6(b) describes get and set methods generated for some attributes of the class *WashingMachine*. To demonstrate the behavioral code generation, code fragments of the method main of the *WashingMachine*, were generated from the corresponding sequence diagram (Fig 2) and are depicted in Fig. 7. In the first line of the method body (line 88), the method *getWashSelection* is invoked, according the message sent from *WashingMachine* to the *washOption*, and returning a value that is signed to the variable option. Most interactions in the sd *Main* are represented inside an alt fragment, responsible to define the operation mode according to the option value. The code correspondent to this fragment starts in line 89, and finishes in line 104. The generated code includes invocations of methods from other classes (line 141) and from the owner class (line 91), and also includes the arguments passed as parameters for the methods (see line 142). In line 148, the loop fragment represented in the sd from Fig. 3(b) is represented by the statement *while*. To finalize the code, the tool generates the methods defined in the interface *Machine* (from 155 to 158), which must be implemented in this class, since a realization relationship was defined between these entities.

```java

public class WaterSensor extends Sensor{

    /**Attributes */
    public int currentLevel;
    public int desiredLevel;

    /**Attribute of Return Method levelTest */
    public boolean value;

    /** Constructor */
    public WaterSensor( int currentLevel , int desiredLevel ){
        super();
        this.currentLevel = currentLevel;
        this.desiredLevel = desiredLevel;
    }

    /**Abstract Method of Super */
    public boolean check(){
        return full;
    }

```

```java

public class WashingMachine implements Machine{

    /**Attributes */
    private Engine engine;
    private WaterSensor waterSensor;
    private WashOption washOption;
    private Timer timer;
    private int washTime;
    private int rinseTime;
    private int spinTime;

    /** Constructor */
    public WashingMachine( Engine engine , WaterSensor waterSensor ,
            WashOption washOption , Timer timer , int washTime ,
            int rinseTime , int spinTime ){
        this.engine = engine;
        this.waterSensor = waterSensor;
        this.washOption = washOption;
        this.timer = timer;
        this.washTime = washTime;
        this.rinseTime = rinseTime;
        this.spinTime = spinTime;
    }
```

Figure 5. Code generated for the *WaterSensor* (a) and *WashingMachine* (b) classes

```java
    /** Get */
    public Engine getEngine(){
        return this.engine;
    }

    public WaterSensor getWaterSensor(){
        return this.waterSensor;
    }

    public int getWashTime(){
        return this.washTime;
    }
```

```java
    /** Set */
    public void setEngine( Engine engine ){
        this.engine = engine;
    }

    public void setWaterSensor( WaterSensor waterSensor ){
        this.waterSensor = waterSensor;
    }

    public void setWashTime( int washTime ){
        this.washTime = washTime;
    }
```

Figure 6. Generation of methods "Get" (a), and "Set" (b) of WashingMaschine

```java
85      /** Methods */
86      public static void main( String args[] ){
87          /** Specified from Sequence Diagram Main */
88          option = washOption.getwashSelection();
89          switch(option){
90              case 1:
91                  standardWash();
92                  break;
93
94              case 2:
95                  twiceRinse();
96                  break;
97
98              case 3:
99                  spin();
100                 break;
101
102             default:
103                 break;
104             }//endSwitch
105     }
```

```java
139     private void spin(){
140         /** Specified from Sequence Diagram Spin */
141         engine.turnOn();
142         timer.setDuration(spinTime);
143
144         /** Specified from Sequence Diagram Period */
145         timer.start();
146         time = timer.getValue();
147         duration = timer.getDuration();
148         while(time != duration){
149             time = timer.count();
150         }//endWhile
151         engine.turnOff();
152     }
153
154     /** Methods From Realization of Machine */
155     public void turnOn(){
156     }
157
158     public void turnOff(){
159     }
```

Figure 7. Fragments of the generated code for the class WashingMachine

4. Related Works

Different approaches for UML-based code generation can be found in the literature. Simple approaches use only class diagram, generating only skeleton of code. Others use a combination of different diagrams (e.g. class diagrams with state, sequence and/or activities diagrams) to generate code. The first proposed code generation approaches including behavioral diagrams were based on state diagrams, due its formal semantic. Different of these approaches, we propose to generate behavioral code from sequence diagrams. These diagrams have been already used to generate code for the methods body in [Long et al. 2005]. In our approach, the same idea is also used, but it considers also the nesting of diagrams and newest UML2 notations. After diagrams capturing, a simple conversion from model elements to text is used as in [Usman and Nadeem 2009].

5. Conclusions and Future Work

This paper presented an approach for code generation from UML models, which is based on class and sequence diagrams. A code generator was implemented and used to demonstrate the feasibility of our approach through a case study. Complete code is not able to be generated since the sequence diagram granularity is method invocations, but it is acceptable since high abstraction is provided for system modeling. As future work, we plan to extend our approach in order to support other diagrams and enable a more complete code generation. Furthermore, we plan also ensure the consistency of diagrams before code generation.

6. Acknowledgments

We thank to CNPq for the financial support.

References

Brisolara, L. et al (2008). Using UML as Front-end for Heterogeneous Software Code Generation Strategies. In: Proc. of the Design Automation and Test in Europe (DATE), 2008, p. 504-509.

Long Q. et al (2005). Consistent Code Generation from UML Models. In: Proc. of the Australian Software Engineering Conference, 2005. p.23–30.

OMG. (2011) UML. Available at: <http://www.omg.com/>.

Papyrus. (2011) Papyrus 1.12. Available at: <http://www.papyrusuml.org>.

Parada, A.; Siegert, E.; Brisolara, L. GenCode: A tool for generation of Java code from UML class models. In Proc. of the 26th South Symposium of Microelectronics, 2011.

Selic, B. (2003). Models, software models, and UML. UML for real: Design of embedded realtime systems (pp. 1-16). Boston: Kluwer Academic Publishers.

Selic, B. (2006). UML 2: A model-driven development tool. Model-Driven Software Development. IBM Systems Journal, Riverton, v. 45, n. 3, p. 607-620.

Usman, M.; Nadeem, A. (2009). "Automatic generation of Java code from UML diagrams using UJECTOR". International Journal of Software Engineering and its applications (IJSEIA), Daegu, v. 3, n. 2 (April), p. 21-37.