Project definition – Project 1: Generate Java Code

Jesper Kristensen s062397, Philip Back s062398, Group D, March 27, 2009

The main task of this project is to take a model of a system created with the SE2 CASE tool and to generate Java code, which implements the system created by the CASE tool user.

Context

The CASE tool from SE2 is a piece of software, in which a user can model and simulate an embedded system of software and hardware. First the user of the CASE tool models the capabilities and behavior of each individual software and hardware component which is called a component definition. Then the user of the CASE tool connects the individual components to each other which is called a deployment, and then the user of the CASE tool can simulate his system and analyze the simulation.

Our project builds on top of the model which is used by the CASE tool, and in this project description we assume knowledge of the CASE tool model. However we don't assume knowledge of any other parts of the CASE tool, such as its editors.

Our project

We have the CASE tool where systems are stored in a model, which can store any system supported by the CASE tool. We want to generate Java code which should be specialized to run only one given system. The generated code should not only implement a static model of the system. The generated code should also allow the system to be executed. We will use the JET technology to generate Java code from the CASE tool model. Finally an example system should be created, and some form of GUI should be created to interact with the generated code from the example system.

Example

We will describe our project using an example of a vending machine, which can take coins and return candy whenever enough coins are inserted. This example is a model which could be given as input to our tool. Our example system looks like the following:

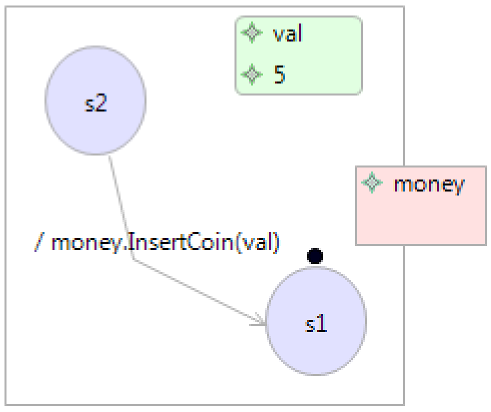

Figure 1: CoinSlot

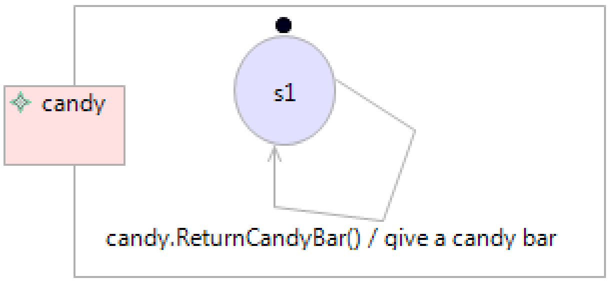

Figure 2: CandyTray

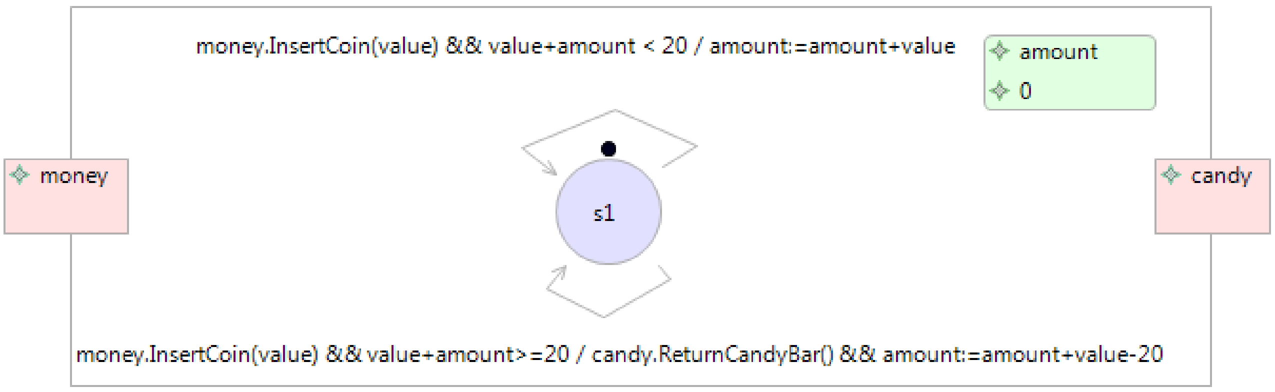
money.InsertCoin(value) && value+amount < 20 / amount:=amount+value
amount
0
money
s1
candy
money.InsertCoin(value) && value+amount>=20 / candy.ReturnCandyBar() && amount:=amount+value-20

Figure 3: VendingMachine

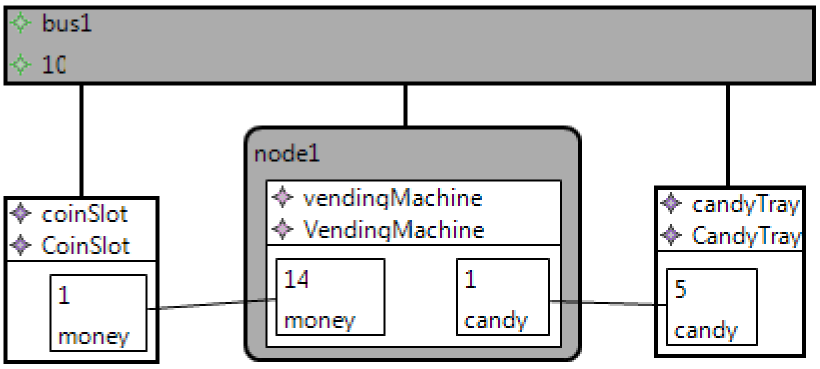

Figure 4: VendingMachineSystem

Our system consists of a software component in figure 3, which keeps track of how much money is inserted into the machine and delivers a candy bar when enough money is inserted. The hardware component in figure 1 allows the user of the vending machine to insert coins. In this model inserting a coin is done by modifying the "val" attribute to the coin's value and changing the state to s2. The hardware component in figure 2 takes care of actually delivering the candy bars. Figure 4 shows how the three components are connected over a bus.

Generating code for a component

For each component definition used in the deployment from which code is generated, the generated code should contain a Java class which implements that component definition. The Java class corresponds to the component definition from the CASE tool model, and each instance of this Java class corresponds to a component runtime instance from the CASE tool model.

Such a Java class for figure 3 could look like this:

```
public class VendingMachine {
```

Each class representing a component definition should have some fields. There should be one field for each port in the component definition. These fields should reference objects, which represents the ports (these objects are described later).

Example for figure 3:

```
    private final Port money;
    private final Port candy;
```

There should also be fields in the Java class for each attribute definition in the CASE tool model (in this text “attribute” always refers to the concept of an attribute from the CASE tool model). The value of each of these fields should be the value of the component’s attribute during execution (the AttributeRTInstance from the CASE tool model). Each attribute field is initialized to the initial value of the attribute.

Example for figure 3:
```java
    private int amount = 0;
```

The last field required is a reference to the component’s current state in the execution. To have that, a list of possible states for the component definition is also required. Thus each component definition should have an enum of all states within its automaton. The “current state” field is initialized to the initial state of the component.

Example for figure 3:
```java
    public enum States {
        s1
    }
    private States currentState = States.s1;
```

The Java class representing a component definition should also have a step() method, which implements the component definition’s automaton so that it acts in the same way as the simulation algorithm from SE2. (More on that later)

Example for figure 3 (details explained later):
```java
     public void step() {
          switch (currentState) {
          case s1:
               // Trying: money.InsertCoin(value) && value+amount < 20 /
               // amount:=amount+value
               if (money.getFirstMessage() instanceof InsertCoin) { // Trigger
                    InsertCoin message = (InsertCoin) money.getFirstMessage();
                    if (message.getValue() + amount < 20) { // Condition
                         money.removeFirstMessage(); // Trigger
                         amount = amount + message.getValue(); // Assignment
                         return;
                    }
               }
               // Trying: money.InsertCoin(value) && value+amount>=20 /
               // candy.ReturnCandyBar() && amount:=amount+value-20
               if (money.getFirstMessage() instanceof InsertCoin) { // Trigger
                    InsertCoin message = (InsertCoin) money.getFirstMessage();
                    if (message.getValue() + amount >= 20) { // Condition
                         money.removeFirstMessage(); // Trigger
                         candy.sendMessage(new ReturnCandyBar()); // Out
                         amount = amount + message.getValue() - 20; // Assignment
                         return;
                    }
               }
               break;
          }
     }
```

As seen in this example, the step method switches over the current state. For the current state it checks each of the outgoing transitions for that state to see if it can be used. When it finds a usable transition, it executes the transition and returns.

Ports do not differ significantly from each other, so they do not need to have a generated Java class each. Only one such class needs to be developed, from which all ports can be instantiated.

Example:
```
public class Port {
```

The component definition should also contain information on which other components an instance of it is connected to, when such an instance is created within a deployment. This information will be stored in the Port Java class, as it is the ports which are connected to other ports.

Example from the Port class:
```
    private class Connection {
        private Port target;
        private Bus bus;
    }
    private final Collection<Connection> connectedTo =
        new ArrayList<Connection>();
```

The Java class representing a port should also contain the input buffer for the massages received through that port.

Example from the Port class:
```
    private final ArrayBlockingQueue<Message> inputBuffer;
```

As part of the deployment, busses are needed. The busses of a deployment do not differ significantly from each other, so they do not need to have a generated Java class each. Only one such class needs to be developed, from which all busses can be instantiated. Each bus should contain a buffer for messages sent over that bus.

Example:
```
public class Bus {
    private class MessageAndPort {
        private final Message message;
        private Port port;
        ...
    }

    private final ArrayBlockingQueue<MessageAndPort> buffer;

    Bus(int buffersize) {
        buffer = new ArrayBlockingQueue<MessageAndPort>(buffersize);
    }
    ...
    public void step() {
        MessageAndPort map = buffer.poll();
        if (map != null)
            map.port.addMessage(map.message);
    }
}
```

Generating code for a deployment

We then need to generate a Java class from the deployment in the CASE tool model. This class represents the deployment, and each instance of this class represents an execution of the given deployment.

Example for figure 4:
```
public class VendingMachineSystem {
```

The Java class needs to have fields for each bus in the deployment and for each component in the deployment. When initializing these fields, the buffer sizes of the ports and busses are given.

Example for figure 4:
```
    private final Bus bus1 = new Bus(10);
    private final CoinSlot coinSlot = new CoinSlot(new Port(1));
    private final VendingMachine vendingMachine =
        new VendingMachine(new Port(14), new Port(1));
    private final CandyTray candyTray = new CandyTray(new Port(5));
```

Our generated Java class should not have fields for the connections between ports in the deployment. Instead the generated Java class' constructor should set up the connections between the ports for the components within the deployment.

Example for figure 4:
```
    public VendingMachineSystem() {
        connectPorts(coinSlot.getMoney(), vendingMachine.getMoney(), bus1);
        connectPorts(candyTray.getCandy(), vendingMachine.getCandy(), bus1);
    }

    private void connectPorts(Port p1, Port p2, Bus bus) {
        p1.connectTo(p2, bus);
        p2.connectTo(p1, bus);
    }
```

Just like the component definition class, the deployment class also needs a step() method for running the execution of an instance of the deployment.

Example for figure 4:
```
    public void step() {
        bus1.step();
        coinSlot.step();
        vendingMachine.step();
        candyTray.step();
    }
```

Generating code for an automaton

The automaton for each component definition is implemented in the step method in the component class.

First we do a switch over the current state of the component.
Example for figure 3:

```
        switch (currentState) {
        case s1:
            ...
            break;
        }
```

When the component is in a given state, we go through each outgoing transition and determine if it can be used. If we execute a transition, we return from the step method. If we cannot execute a transition, we continue to the next outgoing transition for the current state until there are no more outgoing transitions for that state.

First we check if the trigger for the transition applies.
Example for figure 3:

```
            if (money.getFirstMessage() instanceof InsertCoin) { // Trigger
                InsertCoin message = (InsertCoin) money.getFirstMessage();
```

If the trigger applies, we get the message and then evaluates the condition:
Example for figure 3:

```
                if (message.getValue() + amount >= 20) { // Condition
```

If the condition is also true, we execute the transition and return.
Example for figure 3:

```
                    money.removeFirstMessage(); // Trigger
                    candy.sendMessage(new ReturnCandyBar()); // Out
                    amount = amount + message.getValue() - 20; // Assignment
                    return;
                }
            }
```

When a transition is executed, relevant messages are sent and attributes are assigned their new values.

In the above example code we called sendMessage. This method is implemented in the Port class, and its purpose is to make sure that all ports connected to the given port receive the message. This is done in the following way:

```
    public void sendMessage(Message message) {
        for (Connection connection : connectedTo) {
            if (connection.bus == null)
                connection.target.addMessage(message);
            else
                connection.bus.addMessage(connection.target, message);
        }
    }
```

Example GUI

This describes the code which this project should allow to be generated from the CASE tool model. We should then create an example model and generate code from that example using our tool. We should also create a sample GUI, which can interact with the generated code from our example model. The user should be able to start/stop/pause the execution, see the simulation unfold on the screen, and manually change the state of a component or add a message to the system. The GUI is not required to be able to interact with any other system than our chosen example.

Summary

We have given an example of a system which could be created using the CASE tool. Our project is to take this system and create the Java code, which we have also shown parts of. The complete Java code which should be created for this particular example is shown below. In our project we will use the JET technology to transform the model from the CASE tool into Java code. We will also provide a GUI for one example of a system generated using our tool.

Complete example

The complete code for our example is shown below. We have not included getters which are needed by the GUI. These getters should of course also be added to the generated code.

Bus.java

```java
import java.util.concurrent.ArrayBlockingQueue;

// Not generated class
public class Bus {

	private class MessageAndPort {

		private final Message message;

		private Port port;

		private MessageAndPort(Message message, Port port) {
			this.message = message;
			this.port = port;
		}
	}

	private final ArrayBlockingQueue<MessageAndPort> buffer;

	Bus(int buffersize) {
		buffer = new ArrayBlockingQueue<MessageAndPort>(buffersize);
	}

	public void addMessage(Port port, Message message) {
		buffer.add(new MessageAndPort(message, port));
	}

	public void step() {
		MessageAndPort map = buffer.poll();
		if (map != null) {
			map.port.addMessage(map.message);
		}
	}

}
```

CandyTray.java

```
public class CandyTray {
	public enum States {
		s1
	}

	private final Port candy;

	private States currentState = States.s1;

	CandyTray(Port candy) {
		this.candy = candy;
	}

	public void step() {
		switch (currentState) {
		case s1:
			// Trying: candy.ReturnCandyBar() / give a candy bar
			if (getCandy().getFirstMessage() instanceof ReturnCandyBar) { //
Trigger
				ReturnCandyBar message = (ReturnCandyBar)
getCandy().getFirstMessage(); // Trigger
				// No condition
				getCandy().removeFirstMessage(); // Trigger
				// give a candy bar
				return;
			}
		}
	}

	public Port getCandy() {
		return candy;
	}

}
```

CoinSlot.java

```
public class CoinSlot {
	public enum States {
		s1, s2
	}

	private final Port money;

	private int amount = 0;

	private States currentState = States.s1;

	CoinSlot(Port money) {
		this.money = money;
	}

	public void step() {
        switch (currentState) {
        case s1:
            // No transitions can be taken
            break;
        case s2:
            // Trying: / money.InsertCoin(val)
            // no trigger
            // no condition
            getMoney().sendMessage(new InsertCoin(amount)); // Out
            return;
        }
    }

    public Port getMoney() {
        return money;
    }

    public void changeCurrentState(States newState) {
        currentState = newState;
    }
}
```

InsertCoin.java

```
public class InsertCoin extends Message {
    private final int value;

    public InsertCoin(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

Message.java

```
// Not generated class
public abstract class Message {
}
```

Port.java

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.concurrent.ArrayBlockingQueue;

// Not generated class
public class Port {

    private class Connection {
        private Port target;
        private Bus bus;

        public Connection(Port target, Bus bus) {
            this.target = target;
            this.bus = bus;
        }
    }
```

```
	private final ArrayBlockingQueue<Message> inputBuffer;
	private final Collection<Connection> connectedTo = new
ArrayList<Connection>();

	Port(int buffersize) {
		inputBuffer = new ArrayBlockingQueue<Message>(buffersize);
	}

	public void sendMessage(Message message) {
		for (Connection connection : connectedTo) {
			if (connection.bus == null)
				connection.target.addMessage(message);
			else
				connection.bus.addMessage(connection.target, message);
		}
	}

	void addMessage(Message message) {
		inputBuffer.add(message);
	}

	public Message getFirstMessage() {
		return inputBuffer.peek();
	}

	public void removeFirstMessage() {
		inputBuffer.remove();
	}

	public void connectTo(Port target, Bus bus) {
		connectedTo.add(new Connection(target, bus));
	}
}
```

ReturnCandyBar.java

```
public class ReturnCandyBar extends Message {
}
```

VendingMachine.java

```
public class VendingMachine {
	public enum States {
		s1
	}

	private final Port money;
	private final Port candy;

	private int amount = 0;

	private States currentState = States.s1;

	VendingMachine(Port money, Port candy) {
		this.money = money;
		this.candy = candy;
	}
```

```
    public void step() {
        switch (currentState) {
        case s1:
            // Trying: money.InsertCoin(value) && value+amount < 20 /
            // amount:=amount+value
            if (money.getFirstMessage() instanceof InsertCoin) { // Trigger
                InsertCoin message = (InsertCoin) money.getFirstMessage(); //
Trigger
                if (message.getValue() + amount < 20) { // Condition
                    money.removeFirstMessage(); // Trigger
                    amount = amount + message.getValue(); // Assignment
                    return;
                }
            }
            // Trying: money.InsertCoin(value) && value+amount>=20 /
            // candy.ReturnCandyBar() && amount:=amount+value-20
            if (money.getFirstMessage() instanceof InsertCoin) { // Trigger
                InsertCoin message = (InsertCoin) money.getFirstMessage(); //
Trigger
                if (message.getValue() + amount >= 20) { // Condition
                    money.removeFirstMessage(); // Trigger
                    candy.sendMessage(new ReturnCandyBar()); // Out
                    amount = amount + message.getValue() - 20; // Assignment
                    return;
                }
            }
            break;
        }
    }

    public Port getMoney() {
        return money;
    }

    public Port getCandy() {
        return candy;
    }
}
```

VendingMachineSystem.java

```
public class VendingMachineSystem {
    private final Bus bus1 = new Bus(10);
    private final CoinSlot coinSlot = new CoinSlot(new Port(1));
    private final VendingMachine vendingMachine = new VendingMachine(new
Port(14), new Port(1));
    private final CandyTray candyTray = new CandyTray(new Port(5));

    public VendingMachineSystem() {
        connectPorts(coinSlot.getMoney(), vendingMachine.getMoney(), bus1);
        connectPorts(candyTray.getCandy(), vendingMachine.getCandy(), bus1);
    }

    private void connectPorts(Port p1, Port p2, Bus bus) {
        p1.connectTo(p2, bus);
        p2.connectTo(p1, bus);
    }

    public void step() {
        bus1.step();
        coinSlot.step();
        vendingMachine.step();
        candyTray.step();
    }
}
```