Tool-support for Model-Driven Software Engineering

Marko Boger, Thorsten Sturm

Gentleware AG
Vogt-Kölln-Str. 30
D-22527 Hamburg
Marko.Boger@gentleware.de

Abstract: Model Driven Architecture is an initiative of the OMG that puts a platform and programming language independent modelling using UML in the center of the development. Transformation from high level views to technology-dependent views take place more or less automatically through mappings. However, this requires heavy tool support and so far few tools to support this idea adequately exist. In a series of papers an outlook on such tools and the effect on the possibilities for a new generation of Model-Driven Software Engineering processes are discussed. The focus in this paper lies on the execution of models and code generation from state diagrams.

1 Introduction

Model Driven Architecture as proposed by the OMG has caught the attention of a large public and is widely being discussed. Prior to this, two different schools of software development have split the development community into two groups. On the one hand side more traditional development processes, today usually based on UML, with RUP as the most prominent example and so called Agile Processes like the much discussed Extreme Programming (XP) on the other are promoted.

For the first, UML has become the "lingua franca" for designing and communicating the architecture of object oriented software systems. It is based on a long tradition of software engineering skills and has emerged from different notations and development processes now unified to one. UML is widely applied, its roots and traditions accepted as best-of-practice.

Contrary to this, new development processes commonly referred to as Agile Processes with XP as one of the most prominent example, throw overboard old traditions, saving only a few, composing them in a new way in an effort to make software development more adaptive to changes in requirements.

One of the things thrown overboard by XP and which is most desired by the traditionalists is modelling as with UML. In XP graphical notations like UML are only used rarely, e.g. for sketching and communicating some aspects of a system, mostly on the whiteboard. But its use differs widely from the traditional way, where the design phase can take months before a single line of code is written.

This is the most controversial point between these two positions. The traditionalists argue that the overall architecture will lack in a system if coding starts right away. The Agile Process proponents argue that instead of the problem the design will become the driver, and as requirements change during development the flexibility to react to these is lost, if design phase and programming phase are divided.

Model Driven Architecture might have the capabilities to reunite these two worlds and bring the advantages of both together without inheriting the disadvantages. With MDA, standardized mappings between different views on a system and between different levels of abstraction are to be developed that might allow a consistent treatment of a system either as a design or as running code. This paper discusses this point of view with special emphasis on class diagrams and state diagrams and their mapping to code. A set of extensions implemented on top of Gentleware's UML-tool Poseidon for UML is used to demonstrate the mappings and their outcome with a practical example.

2 Execution and Testing of Models

XP draws much of its strength from testing. This requires an executable form of the system under development. That is why XP is centred around code. In order to unite modeling and implementation phase and to apply the principles of XP to modeling, two requirements have to be met: models need to be executable and they must be testable. While the traditional way only requires a good drawing tool and XP only requires a compiler and a simple test framework, Model-Driven Software Engineering requires intensive support by an integrated tool that is able to execute UML models, test models, support the mapping from model to code and keep code and model in sync. The next sections explain how execution of single UML diagrams and whole UML models has to be treated to support these ideas.

2.1 Executing UML Diagrams

A UML model is specified by different diagrams that are - as the name indicates - drawings. The idea that drawings could be executable is not new and has been proven useful in many areas like workflow execution or Petri nets but it still seems unfamiliar to many. And especially the idea of executing UML diagrams is often resented. And clearly, the part of UML that is best understood, class diagrams and use case diagrams, offer little to define anything of help for execution. However, a very rich and powerful notation, developed and fitted to the concepts of object orientation by David Harel [DH97], state diagrams (or state charts as called by Harel) are an extraordinary mechanism to define and visualize the behaviour of objects.

It has been shown by tools like TogetherJ that the information present in class diagrams can be mapped to programming languages like Java and, likewise, the static aspects of code can be represented in class diagrams. The behaviour of a class, though, can not be captured by class diagrams. This is usually done in the code where it becomes difficult to document and understand or maintain it by others than the original author. XP tries to get more control over this behaviour through intensive testing. And while it (arguably) works fine for small teams and medium problems, it proves not to be scalable to medium size teams or large problems.

The behaviour can be captured, though, using state diagrams, one for each relevant class and manifested in separate states and the protocol when to change from one state to another and what effects this should have. Execution can than be achieved by transforming the diagrams directly to executable code. The results can be used to animate the original diagram so that the execution of it can be visualised.

During execution for a single class, an instance of a class is created and its state depicted in an instance of the according state diagram. This can be interacted with by sending those events that are allowed according to the protocol the state diagram defines. Our tool allows three modes, single-step, interactive run, and automatic run. In single-step mode, each state transition can be triggered interactively by the user. The according changes can be watched in the diagram. In interactive run mode, the execution proceeds automatically until an event from outside is expected. One can then decide which events or method calls are sent to the model. In automatic run or simulation mode, one of the events or methods out of the set of now accepted triggers is chosen randomly or by a predefined sequence and the execution proceeds automatically.

2.2 Interaction between UML diagrams

To be able to simulate and try out the behaviour of an isolated class is nice. But each class in UML has a certain role in the modelling process - they should be used collectively.

The class diagram provides architectural information gathered from the other diagrams. A class contributes to the system's behaviour in the sense that it provides access methods for attributes and associations. Also, methods are the events that can trigger a state change of an object. Sending such an event from one object to another is an interaction between these. Such interactions can be specified within state diagrams as actions, triggered by incoming events. This way, the interaction between a whole set of objects can be defined and visualized, as will be shown in the example.

2.3 Interaction between UML diagrams and code

In our tool, all annotations in state diagrams, such as sending method calls to other objects, are expressed in Java. Thus, messages can also be sent to any kind of Java objects, they do not have to be defined within the model or even in UML. A simple example is to call System.out.println() at the entry of a state. A more sophisticated example is the creation of a graphical user interface that can interact with the simulated system.

This eliminates the difference between modelled and coded objects. An object can first be modeled graphically and later be implemented in Java. It can be used for execution in both cases. Thus a smooth transition from design to implementation phase is made possible.

3 An Example

The construction of software systems is a practical matter. Thus an example demonstrating the outlined ideas will be helpful to explain it. Gentleware's Poseidon for UML is delivered including a simple but powerful example taken from an e-business environment. It models the structure and behavior of an online shop for a small company selling

software over the internet. It is used as a standard example to explain various aspects of UML and the tool. In this paper, the example should give an impression on how the interaction between UML diagrams and code work, especially for behavioral elements like state diagrams. Therefore we will focus on the part that deals with the validation of a new customer.

3.1 Scenario: A new customer has to be validated

Beside other information, a customer can have a number of email addresses, a credit card, and a delivery address. These information provided by the customer is of high value to the company, but only if know to be valid. They are therefore carefully checked and the validation process continually takes place. The customer is not allowed to pay by invoice if no valid email or delivery address is provided. Paying by credit card is only allowed when at least the card itself is considered valid. But without being allowed to pay by invoice or credit card, the customer is not allowed to buy anything at all.

3.2 Structural Parts

Based on the information given above, it is fairly simple to create a structural model for the customer. It is sketched in the following diagram.

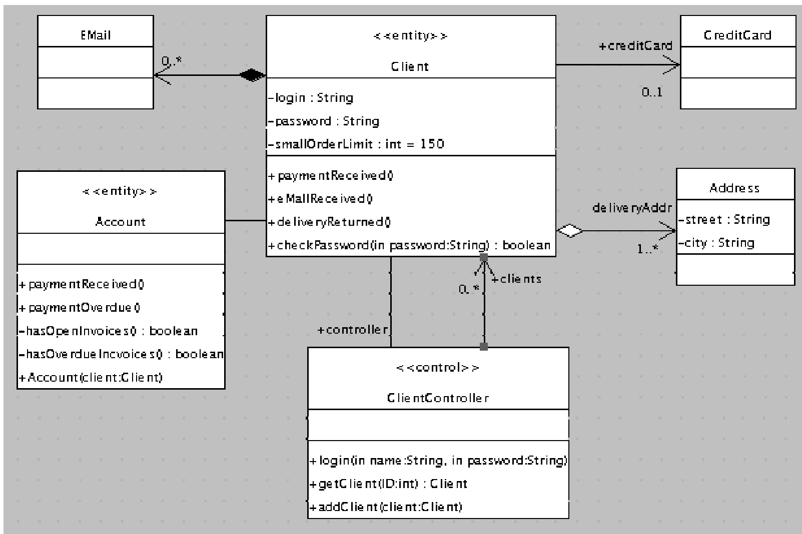

Fig. 1: Class Diagram for the client

The resulting class diagram is a simple one. The email address, credit card and delivery address are modelled as independent classes which has two reasons. First, having independent classes makes the associations between them and the client class visible and eases the notation of multiplicities. Second, it allows defining of behavioral diagrams for each class seperately. The next section will show how the dynamic behaviour can be modeled.

Most of the common UML tools are able to create Java code from such a class diagram. Because there is no standardized mapping between UML and Java, the resulting code

might look slightly different from tool to tool. This is one aspect the MDA is set out to solve. The following listing shows the code generated for the client class.

```
public class Client {
  private String login;
  private String password;
  private int smallOrderLimit = 150;

  Account account;
  Address deliveryAddr;
  public CreditCard creditCard;
  public EMail email;
  public Vector order=new Vector();
  public ClientController controller;

  public void paymentReceived() {
  }
  public void eMailReceived() {
  }
  public void deliveryReturned() {
  }
  public boolean checkPassword(String password) {
      return false;
  }
}
```

Using only the information given in the class diagram would generate classes similar to the one above for each class represented in the diagram. A Java compiler can be used to compile this code and the classes can even be used for running something like a prototype application. But it is easy to see, that these classes do nothing useful. They are just empty shells.

3.3 Dynamical Parts

The scenario gives additional information that can't be used for the stuctural part but for the dynamical parts. There is information about e.g. the credit card and it is considered being not trustworthy until the validation process proves it to be valid or invalid. There seem to be three states the credit card can be in. The information is not validated, valid or invalid. Given this information it is fairly simple to create the corresponding state diagram similar to the one in Fig. 2.

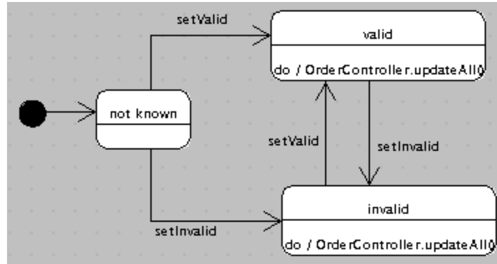

Fig. 2: State diagram for the credit card class

A state diagram similar to this can also be created for the email and the delivery address. The transitions between the states all have triggers. A trigger can be seen as a kind of event that causes the transition to be executed.

The credit card moves from the initial state directly to the state „not known“, which is a synonym for not trustworthy. Being there the credit card waits for an event named „setValid“ or „setInvalid“ which causes one of the two outgoing transitions to be executed. Afterwards the credit card is in the state „valid“ or „invalid“, depending on the catched event.

Although the behavior itself is dynamic and runtime specific, the handling of incoming events is static and can be reflected in Java code. Each trigger is the name of a method. Calling the method means that the corresponding event is fired and the transition is executed.

```
public class CreditCard implements StateMachineOwner {

    public synchronized void setValid() {
            if (islocked()) throw new RecursiveStateMachineCallException();
            lock();
            ((EventInterface)state).setValid();
            unlock();
            while (completionEvent()){}
    }

    public synchronized void setInvalid() {
            if (islocked()) throw new RecursiveStateMachineCallException();
            lock();
            ((EventInterface)state).setInvalid();
            unlock();
            while (completionEvent()){}
    }

    private State state;
    State__invalid state__invalid=new State__invalid(this);
    State__valid state__valid=new State__valid(this);
    State__not_known state__not_known=new State__not_known(this);
```

```
public class State__not_known extends EventAwareSimpleState {

		public State__not_known(StateMachineOwner owner){
			this.owner=owner;
			name="State__not_known";
		}
		public boolean setInvalid() {
			Hashtable params=new Hashtable();
			Transition trans=(Transition)states.get("not known_To_invalid");
			if (trans!=null) trans.highlight();
			//exit-sequence
			exit(params);
			//enter-sequence
			state__invalid.enter(params);
			state=state__invalid;
			if (trans!=null) trans.dehighlight();
			return true;
		}
		public boolean setValid() {
			Hashtable params=new Hashtable();
			Transition trans=(Transition)states.get("not known_To_valid");
			if (trans!=null) trans.highlight();
			//exit-sequence
			exit(params);
			//enter-sequence
			state__valid.enter(params);
			state=state__valid;
			if (trans!=null) trans.dehighlight();
			return true;
		}
	}
}
```

The listing shows excerpts from the resulting code. Each possible state is represented as an inner class that basically has the triggers of the outgoing transitions as its interface. The class itself has a private variable that holds the state object representing the current object state. The class also has an implementation for each possible trigger. The method representing the event to be fired is called at the current state.

Combining both structural information and event handling for state diagrams for code generation results in a class that can not only be executed but is also able to change its state during runtime.

3.4 How to execute the example

Once we have modeled the state diagrams for the email and the delivery address as well as for the credit card, only thing left is a state diagram for the client itself. This can be

very simple because having a valid email and delivery address is enough for the client to be considered verified.

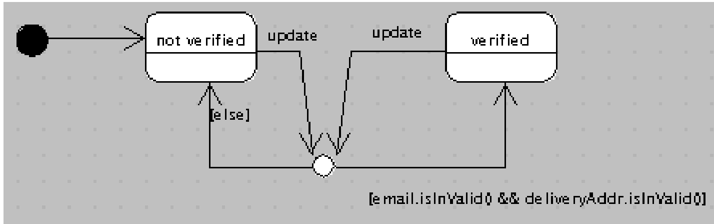

Fig. 3: State diagram for the client

There is a guard watching over the transition from „not verified“ to „verified“ or back. A guard is a boolean expression indicating if all conditions for a transition are met. UML doesn’t define the way how to specify a guard. So it is perfectly legal to use Java expressions for it. That way we can directly use the guard within the code.

```
public boolean update() {
    Hashtable params=new Hashtable();
    Transition trans=(Transition)states.get("verified_To_null");
    if (trans!=null) trans.highlight();
    exit(params);
    if (email.isInValid() && deliveryAddr.isInValid()){
            if (trans!=null) trans.dehighlight();
            trans=(Transition)states.get("null_To_verified");
            if (trans!=null) trans.highlight();
            state__verified.enter(params);
            state=state__verified;
    }
    else{
            if (trans!=null) trans.dehighlight();
            trans=(Transition)states.get("null_To_not verified_7ff8");
            if (trans!=null) trans.highlight();
            state__not_verified.enter(params);
            state=state__not_verified;
    }
    if (trans!=null) trans.dehighlight();
            return true;
    }
}
```

Once the code is generated, the corresponding state diagrams can be executed. They are now mapped to Java classes. Compiling and executing them should be no problem. Part of the nature of state diagrams is that they are driven by events. Although the events are expressed in Java notation, it is quite tedious to create a test frame. Therefore an environment giving the possibility to fire the needed events to the right objects is necessary. This is where the tool support comes into play again.

3.5 What diagram execution can look like

The best place to provide such functionality is within the tool that lets you create your models. Gentleware’s Poseidon for UML lets you generate the code for state diagrams and execute them in one go. It visualizes the change of states for each class individually and allows user interaction such as creation of new objects or firing of events. The execution of our small example is done that way.

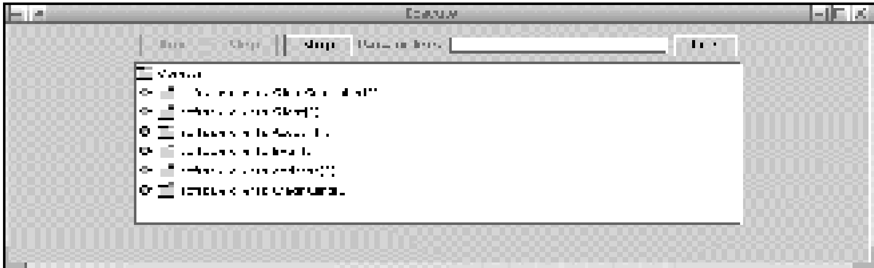

Fig. 4: Execution control panel

Let’s create a new client object (a new customer enters the online shop). A new email and delivery address object as well as a new credit card object are created along with it. They all belong to the new client object. Initially, all of them are considered not trustworthy and therefore are in the state „not verified“. Fig. 5 shows what the client, the email and the delivery address look like initially. Email and delivery address both wait for an event either called ‘setValid” or ‘setInvalid”. The objects will stay in the current states until a new event is fired externally. At this point, the user can decide which way to go by choosing which event to fire.

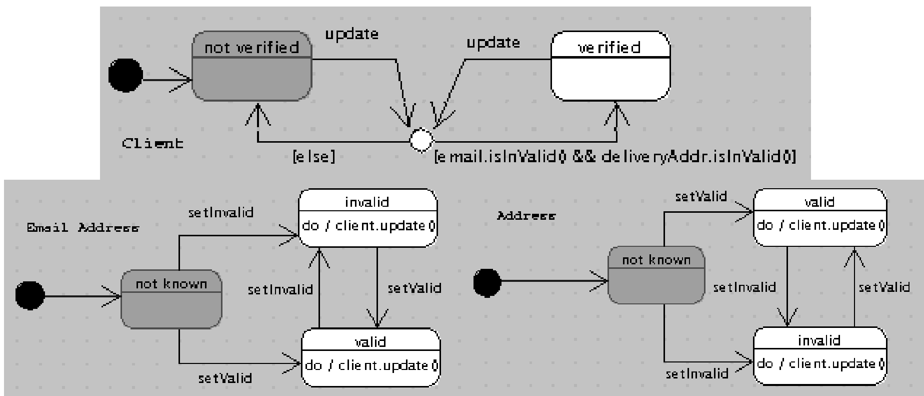

Fig. 5: Initial states of client, email and delivery address

We fire „setValid“ at the new email address object (the email address is proven to be valid). As you can see in Fig. 6, the execution control panel shows all the events an object can receive. Parameters can be added in the text field at the top. Once the ‘Fire” button is pressed, the system calls the setValid method on the corresponding object. The object state changes from „not verified“ to „valid“ indicated by the highlighted state. The executed transitions are painted in blue to visualize the way we came here. It is possible, that more than one transition is executed because entering a new state can also fire additional events. So tracking the way the we came is important information because it is

absolutely possible to end in the correct state by using the wrong way. Because of possible side effects, this might lead into misbehavior.

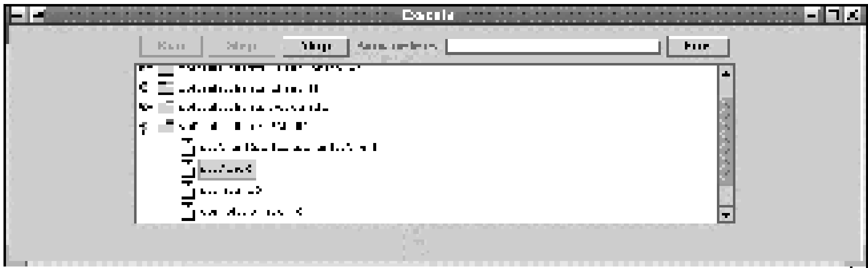

Fig. 6: Execution control panel firing ‘setValid”

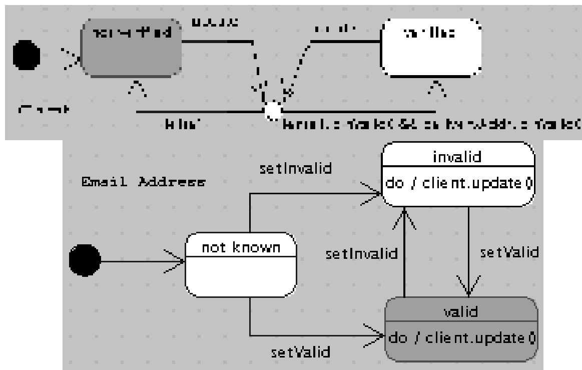

Fig. 7: States of client and email address after recieving ‘setValid”

Let’s try the same for the delivery address. The state diagram for the delivery address shows the same behavior as the one for the email address before. But additionally, the state diagram for the client changes the state as well. Every time the state of either email or delivery address changes, they fire an event to the corresponding client. This causes the transition to be executed and depending on the result of the guard expression change states here as well.

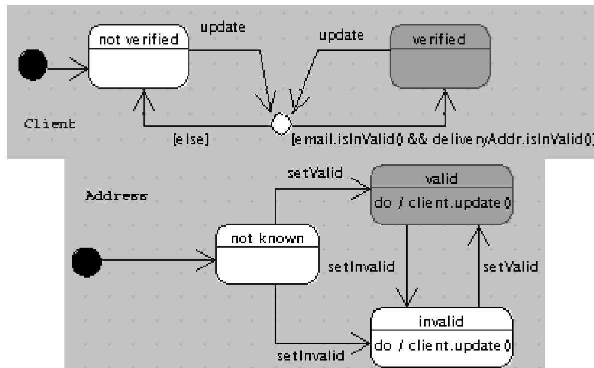

Fig. 8: States of client and delivery address after recieving ‘setValid”

By firing further events to different objects, it is possible to verify that the current behavior meets the specification.

4 Implementation

The presented solution is based on prior work of the authors where state diagrams were first translated to a Petri net representation and then executed by a Petri net engine we called a 'UML virtual machine" [MB01]. Although this approach taught us a lot about the execution of models in general and state diagrams in particular, this approach proved to be too slow and not scalable enough for practical problems. The current implementation directly generates Java code that can automatically be translated without further changes and executed directly from the tool. This allows also to export the code to development environments and to continue work on a program language level. Our experience shows that for most classes the generated code can be used as is and should only be replaced by a handwritten solution if the class is called often. However, the advantage of leaving the generated code untouched is that the model can then be changed and the code regenerated. That way, the business logic can be changed visually in the model instead of in the code.

The tools described are implemented as plug-ins for the UML-tool Poseidon for UML. A free version of this tool is available from the home page of Gentleware, but the commercial version is required to include the plug-in. The plug-in itself will be made available shortly after publication of this paper and will also be available from Gentleware.

5 Conclusion

UML is currently mostly used to analyse and design the static architecture of a software system and code generation is typically only made use of for the interfaces of classes. However, UML can be used to also express the behaviour of such systems. Furthermore it can be used as environment to automatically map the model to a programming environment, to execute the system and to visualize the internal workings as has been shown in this paper. The tools shown here have been fully implemented and will be publicly available shortly after publication. This is an important piece in the puzzle of making the vision of a Model Driven Software Engineering as sketched out in the MDA by the OMG come true.

Bibliography

[DH97] Harel, D.: Executable Object Modeling with Statecharts. Proceedings of the 18th International Conference on Software Engineering, 1997.

[MB01] Boger, M. et al.: Extreme Modeling. Proceedings of the 1st International Conference on Extreme Programming, Italy, 2000. Addison Wesly, 2001.

[GW01] Gentleware: http://www.gentleware.com, 2001.

Biography

Dr. Marko Boger is founder and CEO of Gentleware AG. As a researcher working on ideas for simulating models and visualizing running systems, he joined the open source project ArgoUML and became member of the core development team. Together with his research team he then started Gentleware to take the project to a production ready level and productize it under the name Poseidon for UML.

Thorsten Sturm is Chief Software Architect at Gentleware and leads the development of Poseidon for UML. He is an active member of the ArgoUML project and has been on the team of developers for Poseidon from the first hour.