Extreme Modeling

Marko Boger, Toby Baier, Frank Wienberg, Winfried Lamersdorf

Hamburg University, Distributed Systems Group,
Vogt-Kölln-Str. 32, 22527 Hamburg, Germany
{boger|4baier|wienberg|lamersd}@informatik.uni-hamburg.de

Abstract

Extreme Programming (XP) has been widely appreciated as a pragmatic software development process. But it has also been criticised to be centred too much on coding, leaving behind modeling and design. More traditional development processes stress the importance of modeling. There appears to be a contradiction between these two different approaches.

In this paper we discuss how the principles of XP can be applied also in the modeling phase. To achieve this it is necessary to be able to execute models, as well as to test them. A solution providing this for UML is presented. This allows a seamless integration of UML-modeling into XP. We point out the value added to both XP and UML-modeling and propose "Extreme Modeling".

Keywords: UML, Extreme Programming, Software Development Process, Model Execution, Extreme Modeling.

1 Introduction

In the area of object oriented software engineering two of the most important recent developments are UML on the one hand side and Extreme Programming (XP) on the other. 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, XP is a new development process, throwing overboard old traditions, saving only a few, composing them in a radical way and applying them to their extreme. It is centred around code. One of it's most important corner stones are tests, written before and during programming, rather than after, and validated at all times.

It seems as though the appearance of Extreme Programming has split the software developers community in two fractions. On the one hand side, the traditionalists, convinced of the necessity of an intensive modeling phase to find appropriate architectures, on the other hand side, the radical, disappointed by the failures of traditional ways and fascinated by the simplicity and the success of Extreme Programming. There seems to be no in between.

One of the things thrown overboard by XP and which is most desired by the traditionalists is modeling e.g. 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 radical argue that instead of the problem the design will become the driver, and as requirements change during development the flexibility to react is lost, if design phase and programming phase are divided.

However, where the traditionalists lack this flexibility to change, the radical lack a language to communicate and document their design. These disadvantages could be eliminated if the advantages could be unified synergetically. The obstacle is the devision of the design phase from the programming phase. In this paper the possibilities of uniting these two phases, of integrating UML into XP and of applying the principles of XP to the modeling phase are discussed. This integrated approach is presented as "Extreme Modeling".

2 Meeting the preliminaries

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, Extreme Modeling requires intensive support by an integrated tool that is able to execute UML models, test models, support the transition 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 XM.

2.1 Executing UML Diagrams

A UML model is specified by different diagrams that are – as the name indicates – drawings. Each dynamic UML diagram shows a part of a model which has an operational semantics, so it should be executable, as has been shown for state diagrams [2][3] and message sequence charts [5].

Execution of the dynamic UML diagram types can be achieved by transforming the diagrams directly to executable code or by translating them into an intermediary format with a precise operational semantics that can be interpreted by a machine. In our project we have chosen the latter: UML diagrams are translated into a special kind of Petri nets [6] by a newly developed compiler. The Petri nets in turn can be executed by a Petri net engine. The results can be used to animate the original diagram so that the execution or simulation of it can be visualised. This translation procedure can be compared to the compilation of Java source to byte code, which can be interpreted by a Java Virtual Machine. The diagrams correspond to the Java code while their Petri net representation corresponds to byte code. Accordingly we call this Petri net engine a UML Virtual Machine.

Different to code, a model does not have to be specified completely to be executable. Underspecified parts (like conditions of loops and branches) can be decided by the user at runtime or randomly. Our UML Virtual Machine also allows to use undeclared variables and assigns an according type at runtime; dynamic type checking is enforced. This gives more flexibility at modeling time without giving up static typing for refined models and the final code.

During execution, our tool allows three modes, single-step, interactive run, and automatic run or simulation. A transition (arc) of the statechart is translated to a sequence of sub-steps (in UML called a “run-to-completion step”). In single-step mode, each sub-step 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 and the execution proceeds automatically.

We now briefly introduce all dynamic UML diagrams and how they are executed. A state machine defines the states of an object, the events leading to state changes, and actions to be taken. During execution of such a diagram the actual state and its change can be visualised. The following example of a state diagram (taken from the book UML@work [4]) can successfully be translated by our compiler, instantiated and executed by the UML Virtual Machine.

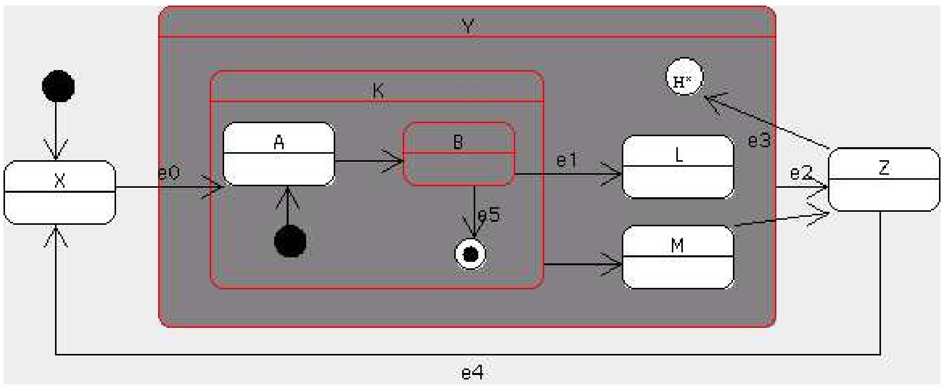

Figure 1: Example of a complex state diagram during execution.

Activity diagrams express processes. In terms of execution, activity diagrams are very similar to state machines. The difference is that the focus is less on states and events, but rather on processes, branches, forks and joins. During execution, the currently active activity is visualised and the transition to the next activity can be triggered. In case of forks or conditions, either the user can choose which option to take or one is chosen randomly, depending on the execution mode.

Interaction diagrams are used to express example runs and come in two different flavours, collaboration diagrams and sequence diagrams. These are interchangeable but stress different aspects. During execution, the modeller can see whether objects are active, waiting (suspended), or passive. Figure 2 shows an example of a collaboration diagram, which was compiled and executed with our tool.

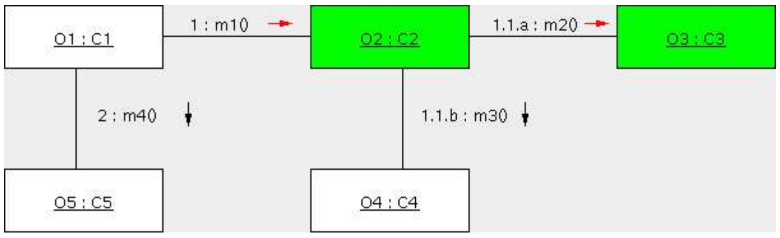

Figure 2: Example of a collaboration diagram during execution.

2.2 Interaction between UML diagrams

To be able to simulate and try out a single diagram is nice. But each diagram in UML has a certain role in the modeling process – they should be used collectively.

The class or static structure 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.

Together, state and activity diagrams specify the dynamic behaviour of a system. We assume that a state machine models the state of a single object. It can be seen as a model of the life-cycle of an object or as the protocol to use an object. Events are mapped to methods of the corresponding class. Activity diagrams are used for modeling the finegrained steps of a single method. Although they can be used on different levels of abstraction in UML, one can argue that higher level activity diagrams will eventually be implemented as a method of some controller or workflow object.

Typical collaboration between objects can be shown with sequence or collaboration diagrams. Sequence diagrams specify interactions between sets of given objects. An interaction consists of messages which are sent from one object to another object of this set. Collaboration diagrams additionally contain information about which associations are used to find the other objects of the interaction. Also a protocol of executing a UML model can either be visualised as a sequence or collaboration diagram.

All of the above diagrams can be translated into the same underlying representation (Petri nets) by our compiler. To actually run the simulation, diagram instances and the corresponding Petri net instances are created. While running simultaneously, they interact by method calls and events. As inscription language we use Java which is interpreted by the UML Virtual Machine. An example is shown in section 4.

2.3 Interaction between UML diagrams and code

Since all annotations in the diagrams are in Java, messages can also be sent to real Java objects. The UML Virtual Machine contains a Java parser, so that the inscriptions can be checked for syntactical correctness. Method calls to Java objects are invoked using the reflection mechanism. 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. Similarly, diagram instances are represented by Java wrapper objects that can be called from usual code.

The combination of these two mechanisms eliminates the difference between modelled and coded objects. An object can first be modelled 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.

2.4 Keeping model and code consistent

Usually, modeling and coding are done in completely separated phases and different tools are used. This strongly hampers the consistency of model and code and is often used as an argument why modeling should not be used in XP: Either the model becomes the driver and determines where the code is going, meaning that flexibility in the implementation phase is lost, or both model and code need to be changed, meaning high costs for change.

Code, just like the different UML diagrams, can be seen as one specific view on the model that is being constructed. If the meta model used to store this model is powerful enough to hold both diagrams and code in a consistent way, all diagrams as well as the code can be handled within one tool. If the model is changed from any one of these views, this is directly reflected in all other views.

3 The principles of XP applied to modeling

In the previous section an approach for executing UML models was presented. While this can have a strong impact on the traditional, modeling-oriented process of software development, the focus in the following discussion will be on how executing models applies to XP. This will lead to a process we call Extreme Modeling.

This discussion will be guided by the "basic principles" and "four values" of XP stated by Kent Beck in [1]. Good practices which are not touched by this, like pair programming, on-site customer, or listening, are left out. We will start though with the most important corner stone of XP, testing.

3.1 Testing UML-Models

In XP code is tested through other code. With Extreme Modeling there are additional possibilities:

- Testing models through models. Since diagrams can interact during execution, they can be used to express tests on other diagrams. Most typically, interactions (e.g. sequence diagrams) can test behavioural specifications (e.g. statechart diagrams).
- Testing models through code. Diagrams can be referenced and accessed from regular Java classes. This way usual tests, e.g. an XP testcase written in Java relying on JUnit [9], can be used to test a (set of) state machine(s).
- Testing code through models. The interaction between code and model works in both directions, so that a class can be tested for example by a sequence diagram.

Not only can these test flavours be executed individually, they can be mixed at will. What is required is a test framework, similar to JUnit, that supports this way of testing. In fact, JUnit can be reused to a large extend as is. Some additional features are required though. It should be possible for tests to ask statechart diagram instances whether they are in a certain state. Then assertions in the style of JUnit assertions can check the expected state of the target object after a sequence of method invocations from a sequence diagram. This can be achieved in several ways, we propose to use a generated boolean method isInState(state).

Sequence diagrams are most typically used to express test cases. They can contain object instances deduced of the class junit.framework.Test. Such objects inherit methods for expressing assertions that can be called in the sequence diagram. For testing code through diagrams, it is possible to use the JUnit framework directly without further changes.

From these test cases a test suite can be generated. This can be done by instantiating and calling each such sequence diagram individually from a special testsuite sequence diagram, or by adding each sequence diagram that contains an instance of junit.framework.Test automatically.

Two sequence diagrams are special: setUp and tearDown. These are used as in JUnit, the first is executed prior to any test sequence diagram, the latter afterwards. This ensures that each test is executed in the same environment.

With such extended testing abilities, all other XP-basics can be reconsidered for modeling.

3.2 Rapid Feedback

With standard modeling processes, you have no feedback from the model, since it could not be executed or tested. The first feedback you get is when you actually start coding. With XM you can sketch out an idea of an architecture in UML, run it interactively while it is still largely underspecified and see what you get. Instantly, you get a feeling of how your idea works. You can find problems and misunderstandings faster.

The programmers can always see whether the designed model is consistent, whether the behaviour intended is actually modeled, or if the model is valid at all. This is feedback to the programmer that has not been possible before, and it is much more rapid.

So the basic principle of rapid feedback is improved by Extreme Modeling.

3.3 Assume simplicity

Extreme programmers always want to have the simplest possible design. While conservative UML-modellers tend to design the whole system first, before starting to code, there is no need to do this anymore, because cost of change is really low with a system, in which model and code are always kept consistent. So you can start off with the simplest design that supplies the need. But the advantage of having a graphical view on the system still enables us to find even simpler designs. So Extreme Modeling can enhance XP also in this respect.

3.4 Incremental change

In XP short development cycles are realized by the code passing all tests. Tests are the means to ensure that the system under development is consistent. Since with our approach, models can also be tested and validated, this notion is now applicable on the modeling level. Like in XP, consistence results in a higher confidence in your work. This way you can “embrace change” already during modeling.

3.5 Communication and feedback

Having more and earlier feedback, communication between developers as well as with customers is highly improved. It is a lot easier to communicate about a model than about code, and easier yet about a running and visualised model compared to “dead” drawings.

Customers might not be able to comment on code. They might even have problems reading UML diagrams. But a running prototype appropriately visualised with UML might do. Customers will be able to get and give feedback much earlier.

Just like code tests, model tests improve documentation and communication of the system’s features. However, by using graphical notations, readability is enhanced.

4 An Example

The construction of software systems is a practical matter. Thus an example demonstrating XM will be helpful to explain it. We have chosen an example that has already been used for the traditional process of modeling as well as for XP. The chosen problem can be found in the book [7] and is referred to as Mark IV Coffee Machine problem. In [9] the development process for designing and implementing this example can be found, once using traditional modeling with UML and Java and once using XP.

The problem at hand is the design of a coffee machine, controlled by a Java chip. It should be able to brew coffee and keep it hot on a heater plate. If the pot is removed, the brewing (if still running) must be interrupted and the heater plate turned off.

It is not our intent to discuss the full development process from very early analysis to deployment. We will start out after the problem has been well understood and a simple first model has been established. At this stage the traditional approach derives the class diagram shown in figure 3.

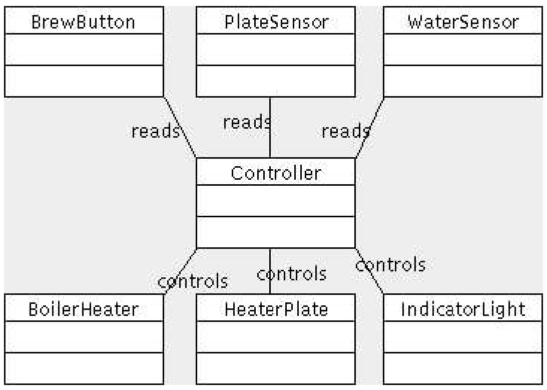

Figure 3: A first class diagram for a coffee machine.

Now let's continue in an Extreme Modeling fashion: before we define any behaviour, we write a test. For this we use a sequence diagram containing a Test object and the Controller object we want to test. The Test object sends the message brewButtonPushed() to the Controller to which we add an empty method with this name. Next the Test-object checks whether the Controller object is in state Brewing. If we run the test now, it fails, because the controller's behaviour is not modeled yet. We start with a very simple statechart diagram with only two states, Off and On. A transition from Off to On triggered by brewButtonPushed() is added, and our test succeeds.

When the water tank is empty, we want the controller to be in state Off. We add a method tankDry() to our controller, which is called by the Test object in the sequence diagram (see figure 4). After that, we add a check, whether Controller is in state Off. Because the test fails now, we add the appropriate transition to the statechart diagram. We run the test again, and now it succeeds.

Next class to test is the BoilerHeater. It has to be turned on and off by the controller. We write another test, which checks the state of BoilerHeater before and after brewButtonPushed() is sent to the controller. To get it running we add entry actions to the controller's states that turn the heater on and off. But wait, didn't the customer want the boiler to be turned off when someone removes the coffee pot? We need to refine the controller's statechart diagram, adding two substates to On (Brewing and Waiting), triggered by potRemoved() and potReturned(). Now we must change the boiler test so that we assert the boiler is in state Off when the pot is removed. Now the test fails. We have to adjust the controller state machine to turn off the boiler when entering Waiting and on

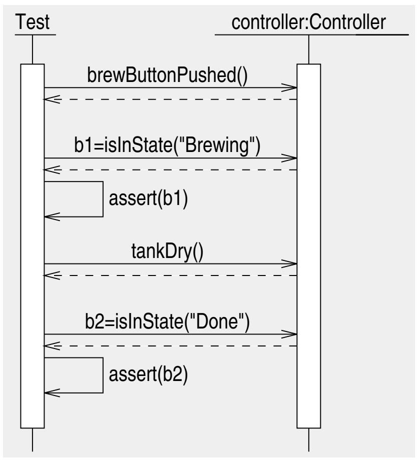

Figure 4: Sequence diagram for testing the boilers behaviour.

again when entering Brewing. The latter is done by moving the boiler.on() action from On to its substate Brewing. Now the test passes again.

We continue with the PlateHeater. It behaves similar to the BoilerHeater, but different to this we do not want it to be turned off when brewing is finished, it should keep the coffee hot until it's removed. Hmm, seems like we need another state for the controller (we'll call it Done), because now not everything is turned off at tankDry(). We create a new test to check that the PlateHeater is still on after tankDry(), but off after potRemoved(). We change the controller's statechart diagram until all tests succeed. The result is shown in figure 5.

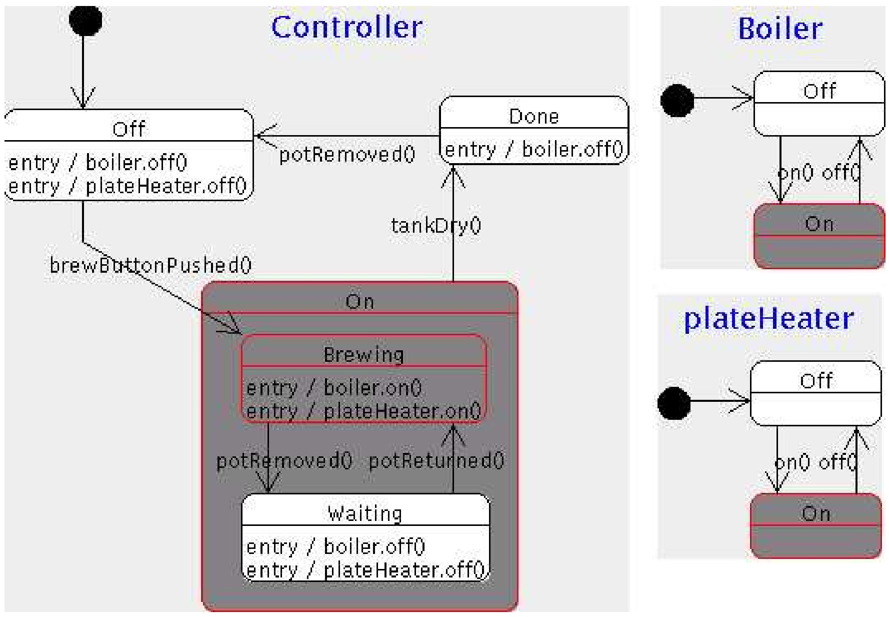

Figure 5: Statediagrams while the Controller is in state Brewing.

After writing a few sequence diagrams that all run to our satisfaction we now write the code for the Controller. To validate that the code fulfils its specification, we use the existing sequence diagrams as tests for this code. This can be done in both directions. So now we also transform the sequence diagrams to code. The code for the test shown in figure 4 looks as follows.

```
public abstract class ControllerTest extends TestCase {
  Controller controller;
  }
  public void testControllerOnOff() {
    controller.brewButtonPushed();
    boolean b1 = controller.isInState("On");
    assert(b1);
    controller.tankDry();
    boolean b2 = boiler.isInState("Done");
    assert(b2);
  }
  public Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new Test("testControllerOnOff"));
    return suite;
  }
}
```

To show that this code can be used to test both, the model and the code, we build two subclasses of BoilerTest, one instantiating the statechart diagram, one instantiating the coded class:

```
public class BoilerModelTest extends BoilerTest {
  public void setUp() {
    controller = new ControllerModel();
  }
}

public class BoilerImplTest extends BoilerTest {
  public void setUp() {
    controller = new ControllerImpl();
  }
}
```

Both tests can be run successfully. The integration of diagrams and code also works on a greater scale, for example the graphical user interface used in the solution of [9] can be used both for the graphical model and for the code implementation without changing the original code.

5 Current Status

The presented tool has been implemented, and a stable and usable prototype is available. The implementation is based on an open source UML tool called Argo/UML [8] and a Petri net tool developed at the University of Hamburg [6]. We are currently able to execute state, activity, collaboration and sequence diagrams. The translation of these to the according Petri net representation works for almost all complex diagram elements, including forks/joins, complex states, history states, transition guards and actions. In some cases where the UML specification is vague about the exact semantics of a notation we have chosen a particular. Details of the translation algorithm are beyond the scope of this paper.

The developed tools and the development process Extreme Modeling are currently evaluated in several practical applications. One of these is a technology study in cooperation with the software engineering company sd&m for a container logistic system for the new container harbour of Hamburg. The process of unloading container from a ship, transporting them to the container storage system and delivering it to a trolly is simulated. The systems behaviour can be executed as graphical model and also as actual code. The graphical user interface can be used both for the executing model as well as for the running code. Results will be presented in a further publication.

The next planned steps are to investigate the possibilities of automatic code generation not only from class diagrams. For example collaboration, activity and statechart diagrams contain important information about possible method bodies.

We have only started to integrate a code editor that directly manipulates the underlying model into Argo/UML, but the current status is quite promising. The UML meta model 1.3 needs to be extended to hold such implementation details.

One of our current problems for larger models is that the handling of many diagram instances becomes difficult. To better support navigation, we plan to develop an improved browser.

6 Conclusion

So far extreme programmers did not integrate modeling languages into their process of development. We pointed out that the reason for this was that models as built with UML were neither executable nor testable. We showed that by translating UML models to a special kind of Petri nets they are assigned a formal operational semantics. This allows the execution and visualisation of UML diagrams. Also, since these can interact, it is possible to use one diagram to test another. The executing engine presented allows bidirectional interaction of models and compiled code. A smooth transition from modeling phase to implementation is achieved. We showed how the appliance of the XP principles and values to modeling can bring benefit to both.

References

[1] Kent Beck. *Extreme Programming explained.* Addison-Wesley, 1999.

[2] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Prog.*, 8:231–247, 1987.

[3] David Harel. On modeling and analyzing system behavior: Myths, facts and challenges. In *ECBS Conference and Workshop of Computer Based Systems, Maale Hachamisha.* IEEE CS Press, 1998.

[4] Martin Hitz and Gerti Kappel. *UML@work.* dpunkt.verlag, 1999.

[5] ITU. Itu-t recommendation z.120 (11/99) - message sequence chart (msc). 1999.

[6] Olaf Kummer and Frank Wienberg. Renew, the reference net workshop, 2000. http://www.renard.de.

[7] Robert C Martin. *Designing Object Oriented C++ Applications using the Booch Methodology.* Prentice Hall, 1995.

[8] Jason Robins. Argouml, 2000. http://Argo/UML: Free Object-Oriented Design Tool w/ Cognitive Support.

[9] Donovan Wells. Extreme programming, a gentle introduction, 2000. http://www.extremeprogramming.org.