Model-Based Design of Embedded Systems

Tim Schattkowsky, Wolfgang Müller
C-LAB, Paderborn University
{tim|wolfgang}@c-lab.de

Abstract

The design of embedded systems is often based on the development of a detailed formal system specification. Considerable effort is spent to ensure the correctness of this specification. However, the actual implementation of the specification and later maintenance is usually done using traditional programming and tends to diverge from the specification. To avoid this, it is desirable to derive the implementation directly from the specification. We present an approach for model-based development of embedded systems applying a well-defined UML 2.0 subset with precise execution semantics. Our approach is fully object-oriented and accounts for important aspects like real-time behavior including timeouts and interrupts. Through the seamless integration of UML sequence diagrams with state transition diagrams, complete executable systems can be described. The direct execution of such models on a UML Virtual Machine (UVM) eliminates the separate implementation step and increases portability.

1. Introduction

The Unified Modeling Language (UML) [16] has become the de-facto standard for software systems modeling. However, its application to embedded systems design is still rather limited. One important reason is that UML application through the whole development process of embedded systems turned out to be complicated. The implementation of embedded systems based on UML models often means a conceptual break when the implementation takes place using traditional programming (e.g., using C). In such cases, substantial changes may occur to the system during its implementation and evolution. This causes specification and implementation to diverge. It is desirable to overcome this problem to preserve the value of the specification. Furthermore, it appears that substantial work is repeated during specification and implementation.

Roundtrip Engineering is one possible way to keep specification and implementation consistent. However, usually it is only supported for parts of the design model (i.e., class diagrams), but mostly fails to capture the complete application behavior. An elegant way to overcome this issue is the use of executable design models. Using executable UML models completely eliminates the separate implementation step. Such approaches do already exist, but mainly target towards platform-specific code generation, where a translator creates platform-dependent code in a programming language (e.g., C++ or Java). Afterwards, this code has to be compiled for the target platform.

A different approach to portable code comes from the idea of using a *Virtual Machine (VM)*. A VM is basically a virtual processor defining a runtime environment for certain software programs. The VM functions as a low-level abstraction layer determining the syntax and semantics of the executed software program. Such a program often consists of *bytecode* similar to the machine code executed by a microprocessor. One example for the use of bytecode is the Java VM (JVM) [13]. Java language programs are compiled to bytecode programs that can run on any Java VM implementation. Such implementations exist for different platforms and from different vendors. This indicates the key success of the VM approach, as the Java VM has enabled software to run on all of these platforms without any efforts from the application vendors.

The use of a VM eliminates the need to translate the models to different platforms by code generators. Instead, only the VM itself has to be ported to each platform. This avoids porting individual software applications to the new VM. Such software is supposed to run on any VM implementation. Thus, improvements to the runtime environment (VM) are immediately beneficial for existing software, which also significantly reduces costs for application development and testing.

The OMG Model Driven Architecture (MDA) [15] targets towards fully model-based software design. It is based on the mapping between a Platform Independent Model (PIM) and a Platform Specific Model (PSM). The PIM contains a platform-independent model of the application. The PSM describes a platform-specific implementation equivalent to the PIM. Both the PIM and

the PSM are UML models. Thus, Executable UML could have a major role here, as it enables the PSM itself to be executable. From the MDA point of view, executable UML models with its execution semantics define a new target platform where the PSM is already the implementation. Furthermore, a high level of abstraction simplifies the derivation of the PSM from the PIM, possibly using partially or fully automated methods.

The UML is intended to be a general-purpose modeling language. In order to be able to have executable models, it is necessary to tailor the UML to the execution platform to enable the creation of a VM for executing the resulting models. The execution semantics of a UML model represented by different UML diagrams have to be defined in this context.

In this article, we introduce a model-based, platform-independent design methodology based on a novel approach on modeling behavior using a combination of UML state and sequence diagrams that enables the creation of UML models suitable for direct execution on the UVM. The methodology gives means to separate concerns and supports timeouts as well as exception and interrupt modeling. In particular, the latter are key issues in embedded systems. Combined application of state and sequence diagrams allows overcoming the limitations of state diagrams when expressing complex algorithms and deeply nested control flows. We demonstrate that control-oriented sequence diagrams can be seamlessly integrated into state-oriented state diagrams. Finally, introduce the concept of a UML Virtual Machine (UVM) to enable the direct execution such models.

The article is structured as follows. The next section discusses related works. Section 3 introduces our model-based design methodology combining state and sequence diagrams to executable models. Section 4 gives the definition of the UVM. Section 5 outlines the implementation of the ZMODEM protocol using our methodology before Section 6 closes with conclusions.

2. Related Works

With respect to embedded systems and Systems-On-a-Chip (SoC) design methodologies, there are very few approaches, which are applied for real systems like the one from IMEC, COSYMA, OCTOPUS [6], or the SpecC methodology [1]. All except OCTOPUS are based on specific design environments and/or languages. OCTOPUS was one of the first methodologies, which followed a straight object-oriented approach including HW/SW partitioning already considering a hardware abstraction layer (hardware wrapper).

However, the Unified Modeling Language (UML) has become the de-facto standard for the design of object-oriented systems. The UML provides means to represent the different structural and behavioral aspects of a software system using different diagrams. Current UML design tools usually support a subset of these diagrams with mostly limited sets of elements. These diagrams can be usually applied with various semantics. If the tools support code generation from such diagrams, code generation is mostly incomplete and the user has to be aware of the tool-specific semantics used in the transformation that are determined by the built-in code generators. UML development environments, like Rose™, Together™, typically do not allow code generation for all diagrams of the full UML standard. Furthermore, behavioral diagrams like activity and state transition diagrams are not fully supported. Tools rather than support code generation just for their specific UML subset. There are a few tools, which support code generation for dedicated platforms like Rhapsody™ with MicroC (C for OSEK operating system) code generation for state transition diagrams.

In the context of the OMG Model Driven Architecture, the notion of executable UML becomes of wider interest [7]. Based on Starr's approach to executable UML [11], which is mainly based on class and state diagrams, Project Technology [8] developed $^{X}{}_{T}$UML as an executable and translatable UML subset for embedded real-time systems in the areas of flight-critical systems, performance-critical fault-tolerant telecom systems, and resource-constrained consumer electronics. The corresponding tool integrates a plugin-interface to adopt different code generators.

A second approach to executable UML is xUML [17], which also includes a complete development methodology. The concept of xUML is based on UML 1.x and the Action Specification Language (ASL) [14], which gives the behavioral semantics of active UML objects. ASL defines actions, which replace existing UML constructs dealing with actions and the definition of computational behavior (i.e., the action semantics). The action semantics defines an abstract intermediate level description by action primitives (e.g., conditions, loops), which carry out computational behavior or access to object memory. ASL is for the definition of both: of model transformations in an operational way and of a model for UML statement. Since the UML metamodel itself is a UML model, the action semantics can be used as a powerful mechanism for meta-programming. All UML features, such as constraints (pre- or post-conditions, invariants), refinements, or traces can be defined by ASL. The definition of the action semantics still enables the mapping to different so-called execution engines like single or multi-processor architectures, which still leaves several different interpretations open. Thus, the approach still requires the implementation of target language-specific code generators from the level of action

semantics. Currently, code generators for Ada, C, C++, Objective C, Java, FORTRAN and SQL are available.

The most prominent approach to portable code was introduced by Java and its VM [4]. Though Java was originally designed for embedded systems, due to the large footprint of its runtime environment and the garbage collection memory management, it is less suited for small micro controllers and real-time applications. There are already Java profile specifications for limited devices like the CLDC (Connected Limited Device Configuration) and the CDC (Connected Device Configuration). However, they mainly target for mobile phones and no reference implementation for small micro controllers is available. There exists a proposal for a UML VM [9]. However, this does not cover the use of a true VM for UML models. Instead, the authors present mainly a Java data model for representing the four-layer metamodel of the UML. To allow the execution of a model by an interpreter, they mainly combine state and class diagrams. Detailed operational behavior has still to be implemented manually using native Java.

Our approach presents a VM designed to directly execute UML models. The underlying design methodology resembles the structure of well-established concepts for embedded system specification by using an executable UML 2.0 subset. The subset covers the seamless integration between state and sequence diagrams. For the efficient execution of that subset for embedded real-time systems implementation, we have defined a dedicated UML Virtual Machine. In contrast to other approaches, our concepts implement a completely model-based approach with an adaptive and easily reconfigurable runtime kernel. To increase efficient execution and to increase expressiveness of the executable subset, we do not compile just to one common Bytecode rather than separate state-oriented from sequence-oriented code and process them on different interacting interpreters. In contrast to other UML-based approaches we additional focus on the specification of exception handling, interrupts, and timeout and its execution.

3. Modeling

The UML metamodel describes the abstract syntax of the UML that determines how the individual UML models are composed. UML diagrams can be used to represent different views of the same model to describe different structural and behavioral aspects. The UML model elements used in these diagrams are defined very broadly to cover different application domains. However, to successfully apply the UML in a certain domain, the set of model elements has to be limited and the semantics has to be tailored to the specific application domain.

Our modeling approach is based on a subset of the UML 2.0 to enable completely executable system models with focus on embedded systems. Thus, high-level concepts like components are out of scope here and the supported model elements need to have precise execution semantics.

Through the combined application of StateMachines and Interactions at the Operation level, we combine advantages from state- and control-oriented modeling. This allows a much more flexibility when creating complex and crosscutting control structures in the model also supporting the full (re)use of StateChart designs in OO systems while avoiding their limitations.

3.1 Structure Modeling

As our approach is based on the UML, it is fully object-oriented. The object model is derived from the UML 2.0 metamodel (see Figure 1). The *Class*[1] in our approach supports Operations and Properties. These can be static or non-static Features. Single inheritance is supported as well as the use of multiple interfaces.

An *Operation* may have all kinds of *Parameter* (in, out, inout, return) as defined by the UML. Additionally, Parameters may have a default *ValueSpecification*. Furthermore, an Operation may raise typed exceptions. The use of Constraints as precondition, postcondition and bodyCondition (as proposed by the UML specification) is not supported, as additional semantics have to be introduced to support these in an appropriate way. All Operation calls are synchronous.

Behaviour may be specified for an Operation to provide an actual specification of the Operation behavior through an UML Behaviour subclass like a StateMachine (represented by a StateMachine Diagram) or Interaction (represented by a Sequence Diagram). If no Behaviour is supplied, the Operation is implicitly abstract. A Class will be implicitly considered as abstract, if it contains an abstract Operation.

Property represents an attribute of Class. Not all features of Property as defined in the UML metamodel are supported. In addition to what is explicitly depicted, only the inherited features from StructuralFeature come into application. Thus, a Property is defined by its name, type, and visibility. No support for collections etc. is provided at this level. This is currently to be implemented by runtime classes. However, arrays are supported.

Attributes may have associated get- and set-operations. If no such operations are provided, it will be assumed that the Property value is held in an instance variable. That feature is provided for convenience. It allows stringent use of the attribute name in the models

[1] Please note that references to classes in the UML metamodel are capitalized.

instead of the usual mixture of get- and set-operation calls. Furthermore, it enforces encapsulation of the attribute implementation at model level allowing overriding direct attribute data access in subclasses with more sophisticated logic.

In our approach, class diagrams provide the necessary type information for a class. Generalizations implement subclassing and implemented Interfaces. Associations in the class diagrams must only reflect the attribute type information provided by the diagram.

Instance and class variables are declared by the Class Properties in the class diagram. These instance variables are the scope of all member operations of a class. Feature visibility (public/private/protected) is supported at the model level. Local variables are defined per State. Such variables will be in the scope of the state and its substates including the refining sequence diagrams. The Operation Parameters are defined in the scope of the State associated with the Operation.

3.2 Behavior Modeling

The UML 2.0 offers a rich set of possibilities for modeling the application behavior. Activities containing sequences of Actions are the foundation of behavioral modeling at the lowest level. Interactions (i.e., in sequence diagrams) and StateMachines (i.e., in state machine diagrams) are used to relate such Activities.

Most approaches employ StateMachines to describe the behavior of a Class at the state level as Transitions triggered Operation invocations. While this syntactically fits at first hand, the actual implementation of these Operations turns often out to be a mess. In contrast, the use of StateCharts [2] for behavior modeling in embedded systems design has been applied quite successfully. The application of StateCharts here is fundamentally different, as the StateCharts reacts to a set of heterogeneous signals without any OO semantics. This makes it hard to use StateMachines the same way as StateCharts in embedded design. Thus, some developers actually tend to implement such StateCharts as a single Operation. This appears to be the most appropriate design, as it avoids the creation of extremely complicated stateful classes with artificial private operations for simulating a StateChart.

The application of StateMachines to model the behavior of an Operation allows not only the integration of successful StateChart based models and approaches in OO systems design; it also provides enhancements to the modeling of the OO systems itself. The use of crosscutting aspects like exceptions and timeouts as well as the implementation of complex nonlinear control flows are simplified using StateMachines at the upper level of behavioral specification for an Operations.

Certain complex behavior with nested conditional control flows, loops, and complex data is usually hard to express in StateMachines and often leads to badly readable models. Such behavior is better expressed by control-oriented means like an Interaction (sequence diagram), where direct support for many of these features exists. The UML 2.0 contains several extensions to the Interaction concepts (e.g., for loops), which makes it an ideal compliment for the use of StateMachines. Thus, we can overcome the drawbacks of just using StateMachines through the seamless integration of these StateMachines with Interactions. Thus, we achieve more expressive and simple modeling combining the advantages of state-oriented and control-oriented design.

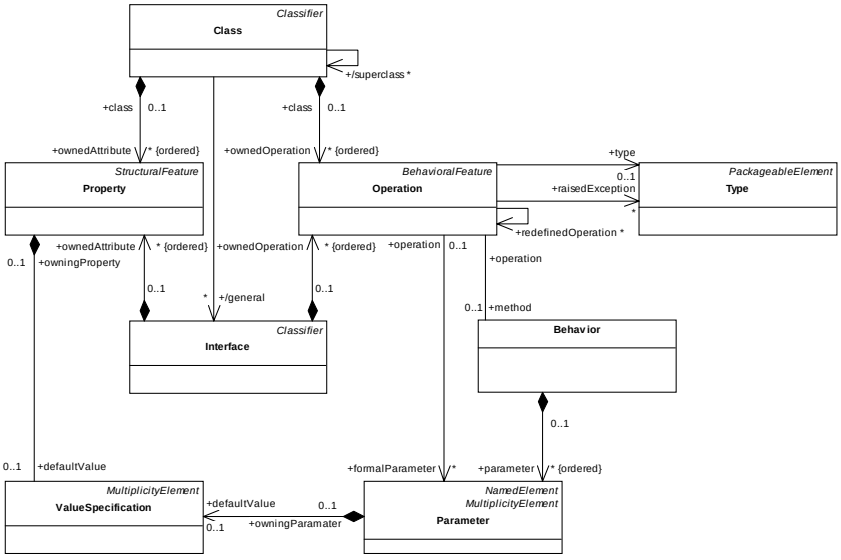

Figure 1: Object Model

Interfaces for passing data and events between Interactions represented as UML sequence diagrams and the StateMachines in our approach need to be defined. These interfaces allow the integration of both into composite models (i.e, a StateMachine containing sequential parts or vice versa).

Exceptions are raised from state machines by placing them as the action of a transition to a final state at the current state machine level. Generally, all exceptions thrown in a sub-StateMachine have to be caught in the containing StateMachine by providing Transitions from the containing state triggered by the exception. These Transitions may raise the same exception again to propagate it in the StateMachine hierarchy.

For timeouts, we use a TimeTrigger with an Integer value and a relative physical time unit as parameter. Absolute TimeTriggers are not directly supported, but can be resembled by computing the relative timeout accordingly. Interrupts and exceptions are introduced as stereotypes of such Triggers.

In our approach, Behavior specification starts at the Operation level with a *StateMachine*. A StateMachine consists of composite and simple States. For embedded systems, neither history states nor current states are supported. The use of concurrent states is intentionally not supported. Supporting concurrent states would add significantly to the complexity of the UVM execution logic for such StateMachines. Even more important, it would introduce concurrency semantics at the StateMachine level. This is not desirable in an object-oriented system, where concurrency should be defined on the object level. Thus, concurrent StateMachines have to be split into several active objects. This enforces a clear separation of orthogonal behavior, but means a significant changes in the execution semantics.

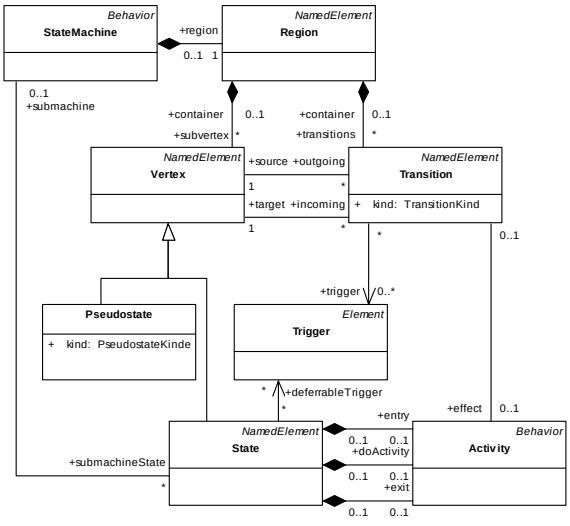

Figure 2: StateMachine Model

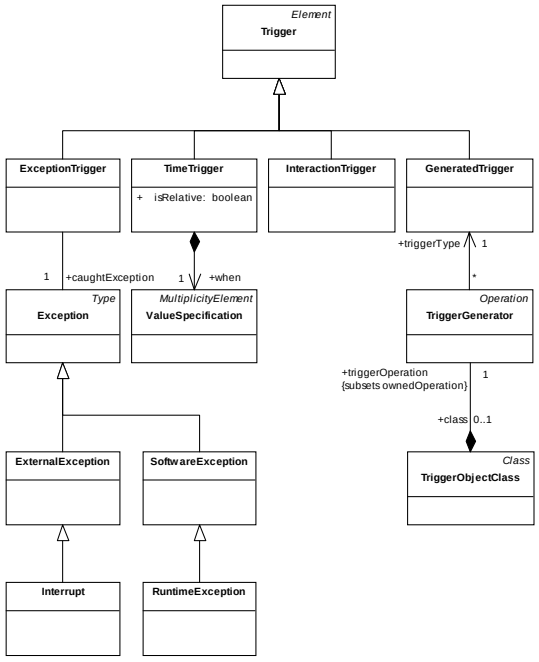

Figure 3: Trigger Types

Thus, a StateMachine (see Figure 2) in our approach consists, besides the start and final pseudo states, of simple and composite states where the latter contains another StateMachine. Both composite and simple States may have Activities describing their behavior on entry, exit and during execution as in the UML metamodel. Transitions between States can have different Triggers (see Figure 3). Such a Trigger can be the occurrence of a timeout or hardware Interrupt, a software Exception (i.e., division by zero) or an explicit Trigger from the current State's implementation (InteractionTrigger). Such Triggers are processed immediately and cause the current State to be exited. A State may have one unguarded completion transition without a Trigger. The whole Operation is completed when the associated StateMachine terminates by reaching a final state.

An integration mechanism is provided to feed additional Events from an Operation into a StateMachine. Each StateMachine may have an attached EventGenerator Operation, which produces additional EventObjects. The EventObject Class defines a member operation that can be used to retrieve the actual event identifier as an integer. Such a call will be made every time the state completes the doActivity and blocks until a Trigger becomes available or another Trigger (i.e., Timeout) fires. Additionally, an Operation can be assigned, that will be called for all compatible actions on transitions within the StateMachine. Together, this allows having individual action/event semantics per StateMachine and enables a much broader use of StateMachines than other

approaches. Finally, it enables the reuse of existing StateCharts. Here, care has to be taken as their execution semantics may differ.

However, to reflect the semantics of Interactions in our approach and to enable simple application for embedded systems, we have to limit the capabilities of Interactions compared to what UML 2.0 allows. In our approach, an Interaction describes an Activity of State or Transition in a StateMachine that describes the Behavior of an Operation on a Class. Thus, the Interaction includes one instance of this class as the only active class. A *Message* in the Interaction either triggers the surrounding StateMachine (causing immediate exit from the Interaction) or represents an Action as defined below. This is a considerable limitation compared to UML 2.0, but is required for a more efficient execution on embedded systems. All Messages have to be synchronous. Control structures are created using CombinedFragment as defined in UML 2.0. The use of CombinedFragment is limited to be an Alternative, Option, or, Loop in our approach. Thus, the CombinedFragment is used to completely resemble all sequential control structures.

When integrating an Interaction into a StateMachine, the Interaction has to expose state-like behavior. The sequential end of the Interaction is equivalent to a completion event. Furthermore, Transitions in the containing StateMachine can be triggered from inside the Interaction. In the model, this is depicted by a call to the transition() operation of the Runtime class. Finally, Exceptions raised by an Interaction have to consistently map to Transitions in the containing StateMachine as described. Interactions inherit the scope for variables from the containing state. Local variables that are implicitly defined within these diagrams are automatically added to the scope of that State.

Finally, in our approach, the language used to specify single Actions is syntactically derived from C++/Java. It allows value assignments based on nested expressions and Operation invocations using variables in the current scope. Computations involve basic math and bit operations as well as the logical operations, as defined by Java. The semantics of those operations are comparable to those in the Java language.

3.3 Data Types

The data types supported in our approach are classes and primitive types. We support the Integer, Boolean and String data types used in the UML specification itself. The UML UnlimitedNatural type used for representing infinity is unsupported due to technical reasons. The introduction of a type should be at the class level.

We have to extend the semantics of the primitive types to meet technical needs for practical application. As numbers inside real systems are of finite precision, we simplify the type system by assuming a signed integer of at least 32 bit for Integer. Furthermore, we introduce the Float type representing an at least double precision IEEE floating-point number. Furthermore, we introduce the Byte type for handling binary data. Bytes are – unlike in other languages – unsigned. These types can also be used to form multi-dimensional arrays.

We finally have to give additional semantics to the use of the String type. Each class may potentially have a constructor using an array of Byte. The use of string literals in the models is limited to assigning those literals to variables of a type that supports this constructor. In the case of such an assignment, the constructor is called with a byte array representing the string in a certain encoding that can be provided by the user. In the simplest case, this is used to have a String class generating a String instance from the array of Byte. However, it may also be used to implement object serialization.

4. The UML Virtual Machine

The UML Virtual Machine (UVM) is a virtual computer for executing UML models based on our approach using the previously introduced combination of state and sequence diagrams. To enable simple and efficient execution of such models, we use a transformation to equivalent executable state-oriented binary models (see Figure 4).

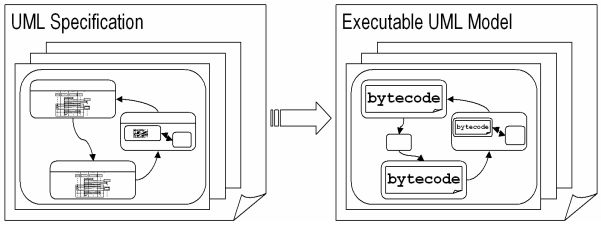

Figure 4: Derivation of the Executable Model

The previously defined UML 2.0 subset is directly mapped to a combination of executable finite state machines with embedded Bytecode where the complete binary representation of the design model can be directly executed on the UVM. A StateMachine is translated on equivalent binary encoding. A Sequence Diagram (Interaction) defining an Activity of a State or Transition is translated to equivalent executable UVM Bytecode.

5. Example

As an example, we outline the model-based implementation of the well-known ZMODEM protocol [5] for file transfers (i.e., used in Telnet sessions) to demonstrate the applicability of our approach.

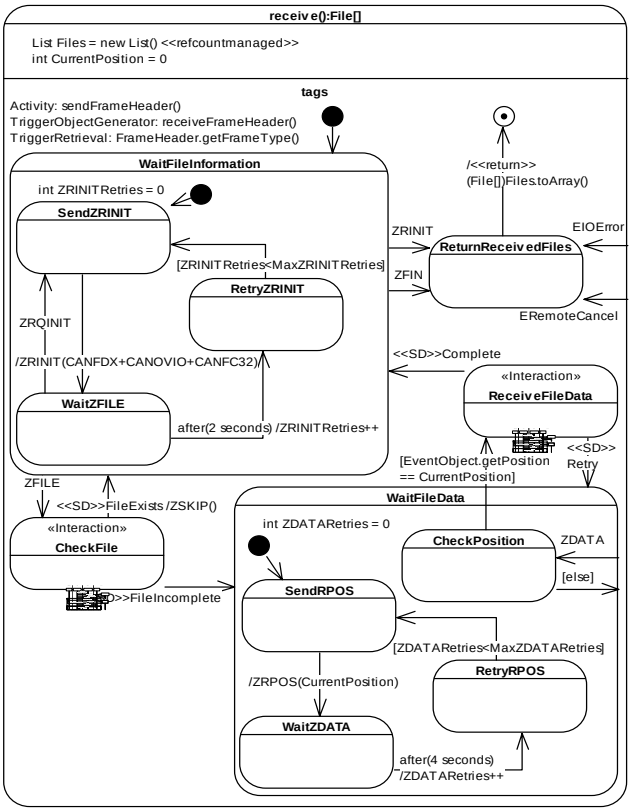

Figure 5: ZModem.Receive()

To outline the basic concepts of the approach, we focus our description of the implementation to the significant parts of the receiver implementation. We presume the implementation to be contained in a class ZModem with operations for sending and receiving files. Figure 5 shows the state diagram for the receive() operation. The operation is returning a set of File objects representing the received files. The implementation of the receive() operation closely resembles the ZMODEM state machine and thus demonstrates nicely the use of an existing StateMachine design in our approach. The provided tags define the event and action processing in this StateMachine as consuming ZMODEM Frames (data packets) using the frame type as the triggering event. The corresponding action is sending back response ZMODEM FrameHeader objects. This leads to the full implementation of the StateMachine based on the ZMODEM protocol frame types.

The StateMachine operates by starting at the WaitFileInformation state waiting for a ZFILE frame header, as a trailing subpacket would be containing the name and size of the file to be received. The state sends a ZRINIT header to initiate the file transfer. It retries this up to MaxZRININT times before giving up. That state illustrates the use of timeouts in our approach. A transition from WaitZFILE to WaitZRINIT takes place after 2 seconds to implement the timeout. It is worth noting that the action on this transition is used to specify simple sequential behavior. This is actually mapped to an additional state in between when compiling the executable model. This feature exists for convenience and allows increasing the readability of the model.

Once a ZFILE frame header has been successfully received, the transition to the CheckFile state takes place. This state is implemented by the sequence diagram (Interaction) shown in Figure 6. An Interaction is preferred here over a StateMachine as long sequential parts with conditional parts exist.

CheckFile reads the file information from the sender and decides whether or not to skip the file and (in case of a resume) from which offset to start the receive operation by setting CurrentPosition accordingly. If the file is skipped, the transition to WaitFileInformation is executed explicitly causing the end of this Interaction. Otherwise, the Interaction ends by making the completion transition to WaitFileData. WaitFileData transmits CurrentPosition using an RPOS frame header and waits for the file data transmission to begin by receiving a ZDATA frame header. The Timeout handling here is very similar to that in WaitFileInformation. ReceiveFileData is responsible for receiving the continuous file data until a transmission error or file end is detected. In case of a transmission error, attempts are made to recover by requesting the retransmission from the last valid position. This is done by executing a transition to WaitFileData. The actual implementation of this state is left out here, as it does not reveal additional insight.

Finally, some exception handling is undertaken in receive(). EIOError and ERemoteCancel exceptions indicate in/out errors and an explicit cancel from the remote side. Both are handled by terminating the session and returning the already received files. We left out additional error handling for the matter of readability.

As it may be inconvenient to enter an Interaction using a sequence diagram, we support the use of C++-style textual code fragments based on the available set of features. The corresponding code for the sequence diagram in Figure 6 is:

```
int Flags=EventObject.getPF();
byte[] FileInfoData=EventObject.readSubPacket();
FileInformation
FileInfo=FileInformation.create(FileInfoData);
File CurrentFile=FileInfo.getFile();
int RemoteFileSize=FileInfo.getSize();
boolean FileExists=File.exists();
if (FileExists&&(!(Flags&&ZMCLOB==0)))
{…}
else
{…};
```

While the StateMachine for receive() is executed directly by the UVM, the interactions are compiled to equivalent bytecode.

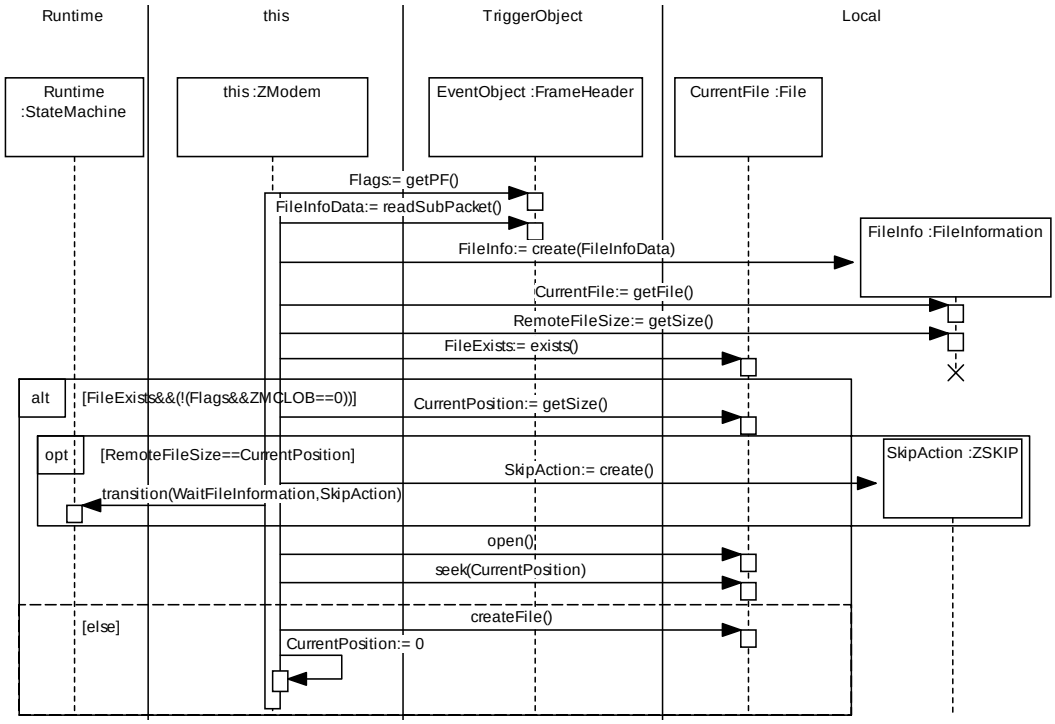

Figure 6: Sequence Diagram for State CheckFile

6. Conclusions and Outlook

We presented a general approach on the design of executable UML models from class, sequence and state diagrams and demonstrated its application by an example. Our approach is based on an UML 2.0 subset with minor extensions. We have defined a Virtual Machine for UML (UVM) to execute such UML models, which is only sketched in this article. The UVM can be implemented in software or hardware. Through the UVM, we achieve enhanced reuse and portability of the design models and eliminate the problem of synchronizations between model and implementation.

First investigations have shown that the UVM requires only a very small footprint. In future work, the software prototype has to be completed and evaluated through more different practical case studies. The final goal is to create a UVM design that contains no native parts except for the model execution logic. Other future work will investigate a UVM implementation on an FPGA.

7. References

[1] Gajski, D.D.; Peng, J.; Gerstlauer, A.; Yu, H.; Shin, D.: System Design Methodology and Tools, CECS Technical Report 03-02, Irvine, 2003.

[2] Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming, June 1987.

[3] Harel, D., Namaad, A.: The STATEMATE Semantics of Statecharts. ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 4, 1996.

[4] Lindholm, T., Yellin, F.: Java Virtual Machine Specification. Second Edition, Addison-Wesley. 1999.

[5] Omen Technology, Inc.: The ZMODEM Inter Application File Transfer Protocol. 1987.

[6] Marwedel, P.: Embedded Systems Design. Kluwer, 2003.

[7] Mellor, S.J., Balcer, M.J.: Executable UML - A Foundation for Model-Driven Architecture Addison-Wesley, 2002.

[8] Project Technology. Home Page, www.projtech.com, 2003.

[9] Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The Architecture of a UML Virtual Machine. In Proceedings OOPSLA 2001, ACM Press, 2001.

[10] Shlaer, S., Mellor, S.J.: Object Lifecycles – Modeling the World in States. Yourdon Press, Prentice Hall (1992).

[11] Starr, L.: Executable UML How to Build Class Models, Prentice Hall PTR, 2001.

[12] Starr, L.: Executable UML: The Models are the Code, A Case Study. Model Integration Llc, 2001.

[13] Sun Microsystems, Inc.: The Java Language Specification. Second Edition. 2000.

[14] The Object Management Group: Action Semantics for the UML. OMG ad/2000-08-04, 2000.

[15] The Object Management Group: Model Driven Architecture (MDA). OMG ormsc/2001-07-01, 2001.

[16] The Object Management Group: Unified Modeling Language: Superstructure. OMG ad/2003-04-01, 2003.

[17] Raistrick, C., Wilkie, I., Carter, C.: Executable UML (xUML). In Proc. 3rd International Conference on the Unified Modeling Language UML, 2000.