Executable UML

Stephen J. Mellor

Executable UML is here. While it is possible today to add code to UML diagrams and then execute them, in so doing you must make a series of decisions about implementation that may not be correct, appropriate, or even knowable. Executable UML models systems at a higher level of abstraction, thus avoiding the costs involved in a premature design.

Executable UML offers the benefits of early verification through simulation, the ability to translate the UML model directly into efficient code, and the ability to delay implementation decisions until the last minute.

This paper describes the components of executable UML and how they fit together. Special attention will be paid to translating an executable UML model into code and what decisions need to be made and what time during the process.

1. Unified Modeling Language

"The Unified Modeling Language is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system" or so says the UML Summary [1]. UML, until quite recently, has not attempted to be executable. In this paper, we examine what is being done to make the UML executable, how execution fits into the UML, various forms of execution that may be based on an executable UML, and what this all means for real-time and embedded systems engineers.

So what does the UML do, if it doesn't execute? It addresses the following development tasks:

- Requirement's gathering and analysis (i.e. the external usage of the system)
- System modeling (how a particular domain operates in terms of data, control and algorithm),
- System deployment (allocation of system functionality to processors, tasks and classes)

Of these, requirement's gathering is not an automatable or executable task—it is a human and creative one that cannot be automated. System modeling, on the other hand, can be made to have a semantics for execution. (System deployment describes allocations of system functionality.)

Because of the broad range of tasks taken on by the UML, the UML has many diagramming notations. Here are a few:

Use Case Diagram: system stimulus-response model (used in requirements gathering)
Static Structure Diagram: package, class, and object diagrams (used in system modeling)
State Diagram: control for dynamic behavior (used in system modeling)
Activity Diagram: workflow of activities (used in system modeling)
Sequence Diagram: dynamic interactions with time (used in system modeling)
Collaboration Diagram: dynamic interactions without time (used in system modeling)
Component Diagram: software components (used in system deployment)
Deployment Diagram: allocation of components to processing elements (used in system deployment)

Some of these diagrams are just views of other diagrams. For example, a collaboration diagram showing asynchronous communication can be built from the state charts and the signals they send and receive. Consequently, we may classify some diagrams as essential in that: they capture the complete scope and behavior of the system and support model translation to code; and some as derived: diagrams that show additional views of the essential models. Auxiliary diagrams can be used to augment the construction of the essential models.
The careful reader will have noted that these diagrams are just notation, not semantics. To be more precise, the UML defines the *static semantics* of the notation, but not the executable, or *dynamic semantics*. Moreover, we have used "model" for the underlying abstractions and "diagram" for a representation of those abstractions.

As a result, a UML *diagram* cannot be executed. It cannot be verified and 100% code generation cannot be achieved.

2. Execution

Model-based execution is the logical next step. In the 1960's we learned how to compile assembly language into machine language. Twenty years later, we learned how to do the same for high-level languages such as C. Now it is time to compile models so that developers work at a more productive level of abstraction, and automate tedious manual tasks. This allows the developer to focus on solving application and product problems, not on implementation details.

Model-based execution relies on the ability to compile or interpret the model. A model compiler can then focus on uniformly solving implementation problems.

Execution is a form of code generation. Bell, in [2], identifies three forms of code generation. The first is structural code generation, which uses the diagram to generate class headers and the like, but the user has to fill in the code for the functions. There are therefore two sources for the code, the code generated from the diagrams (usually just class headers/structs, include files and the like), and hand-written code. Therein lies the problem: the two sources must be kept in synch somehow. If you change the headers as a result of writing detailed code, it is impossible to regenerate the code from the diagram.

Behavioral code generation solves the "unsynchronized code" problem by adding code directly to the diagram. Tools that support behavioral code generation usually generate code from a state chart, rather than just allowing the developer to specify the code for the functions. For behavioral code generation we therefore need to have an agreed execution semantics for the state chart. Because behavioral code generation links the logic to the diagram, it enables early error detection through verification, and reuse of the model compiler. This is a Good Thing, and it is fair to call this form of code generation "executable." But there is a one-to-one correspondence between elements on the state chart and the generated code. Although it is possible to add code to UML diagrams and then execute them, in so doing you must make a series of decisions about implementation that may not be correct, appropriate, or even knowable. The code generation system is closed, and the performance is determined by the model compiler. If the performance is inadequate, the developer generally has to distort the application model.

In translative code generation the model compiler is open. This approach relies on subject matter separation—the complete separation of the application model from the *model compiler*. The model compiler comprises a set of reusable execution engine components, such as inter-task communication and list libraries, and a set of rules that direct the generation. The rules are

completely open, so they may be used to generate C, C++, Ada, Forth, even VHDL. We discuss the details of the model compiler and rules below in Section 6.

3. The Executable Model

The executable model is based on a limited subset of UML assembled in a particular way so that the diagrams have meaning. For example, in UML one may build a statechart for a class, an object, a method, a use case, a component, a package—just about anything. While the execution semantics of the state chart are defined in isolation, this leaves open the question of how the various elements relate to one another.

Executable UML comprises only three diagrams, the class diagram, which outlines the conceptual entities in the domain, a state chart for each class that models the lifecycle of each object, and An activity for each state—the set of actions that establish the state.

The classes must be abstracted based on both similar behavior and characteristics. This requirement allows a state chart to be built for each class that applies uniformly to each object, avoiding logic to determine what type of object each action is executing on.

The state chart captures the lifecycle of each object. There is one state chart per class because the behavior of each object is exactly the same. In some cases, there is no need to build a state chart because the only operations are synchronous data access under the control of other objects. For other classes that manage contention, we build a single state chart for the class as a whole. Conceptually, then, a class may have no state charts, or one chart that applies uniformly to each object, and an (optional) other that apples to the class as a whole.

An activity is a collection of actions that is executed on entry to a state. An activity is almost the same as an action, except that an activity allows for parameters, while actions rely on data flows. Specifically, when an event is received by a state chart, the event may carry data other than the target object, such as the phone numbers involved in a call. This "supplemental data" is made available to all actions in the activity on data flows. Activities on state charts do not have return parameters. All data must (logically) be stored as attributes of objects (a model compiler could optimize this), before the activity completes.

An executable model operates on data about objects. Each class defines attributes for objects, and each attribute may have an initial value for each object. When the model executes, each object has its own state, and each object executes concurrently and asynchronously with respect to all others. Each state chart instance, then, is in its own (just one) state, either executing an activity to completion, or not. The actions inside activities may read data through (logically) encapsulated functions of other classes at the same time as the target object is executing an activity. It is the modeler's responsibility to avoid data access conflict by synchronizing the behavior of the various objects, if required, using a state chart.

The state charts recognize events. Each event may be a signal sent from another object, from the outside, or it can be the result of a "deferred event" that signals expiry of a timer or of some absolute time.

Thus, an executable UML "program" comprises a collection of concurrently executing state machines communicating by sending signals. This highly concurrent model can re reorganized to

be fully synchronous or distributed without changing its behavior. Indeed, it is possible to reorganize the model so that some states are executed periodically in a separate task, and the attributes of a single object are stored in multiple processors. So long as the behavior is the same.

4. The Action Model

4.1 Requirements

Until recently, the UML had an under-developed model of actions. The model comprised seven actions, including create an instance, send a signal, destroy an instance, terminate an instance (don't ask what the difference is), and my personal favorite, "uninterpreted string."

In 1998, the Object Management Group developed a Request for Proposal for a precise Action Semantics [3] that laid out the requirements. One of these key requirements is that the semantics should require the definition only of required sequencing, and should not over-constrain sequence. Specifically, this means that the sequential nature of today's third-generation programming languages is often over-constrained. Consider for example two statements `a = b;` and `x = y;` In a third=generation program, there is a sequence for these two statements, so that `a = b;` must execute before `x = y;` logically, there is no reason why one should execute before the other.

On the other hand, in the fragment `x = 2 * a;` and `b = x * x,` its meaning is different depending on which statement executes first. If `x = 2 * a` should execute before `b = x * x,` the fragment will yield a value for `b` of $4a^2$. In this case, `x` is just a convenience, and could be replaced with its value `2a`. Alternatively, one could think of `x` as being a name for a data flow between the two statements. Of course, the meaning is completely different if executed in the other order. The goal of the requirement is to make the sequencing explicit so that some model compilers can optimize the execution of the statements, possibly even executing one of them on another processor.

Second, the action model should separate functional computation from data access. The intent of this requirement is to allow the statement of a function to be reusable regardless of where the data came from. Today we often express functions and data access code together. For example, if we track the time of each of the last ten calls on a cell phone, as well as the total of the ten, it is not uncommon to loop through the last ten calls, adding each one to the sum. This approach is efficient, but it relies on knowing how those call times are stored in some data structure. If we change the storage scheme, it has the unhappy effect of invalidating the algorithm. To avoid this problem, the action semantics requires that the data access is specified first, then the data values are presented as if they were just *n* elements to a function that sums. This allows a model compiler to optimize the solution based on the selection of the data structure. It also allows a model compiler to select a data structure to optimize the execution of the algorithms.

Third, the action model should manipulate only UML elements. This means two things. First the action model must know about classes, attributes, and other UML concepts. We can think of these elements as being 'pre-declared.' Second, only UML elements are permitted. No pointers, no arrays, no magic, except as explicitly modeled in UML. This has the effect of restricting the generality and so making a specification language.

The action model is therefore *software*-platform independent. It makes no assumptions about the organization of the software, only of the specification.

The action model does not require the specification of a notation. Any action language that conforms to the semantics is acceptable. This allows terse notations, verbose notations and even graphical ones.

4.2 The Result

In mid-2001, a consortium of companies made a proposal that satisfies the requirements of the RFP. This was accepted later that year, thus endowing UML with a semantics for actions. In the remainder of this section, we describe a few salient features.

In a distributed systems environment, time is relative due to transmission delay. The UML action model defines its semantics by assuming the only guarantee is the sequencing of signals between sender and receiver pairs. This is especially important for many real-time systems, even small, embedded systems that need to coordinate hardware and software. These systems cannot possibly assume a single synchronized sequential flow of control, for the simple reason that there isn't one.

Actions can execute as operations in classes, on entry to or exit from a state, and on a transition. Each such grouping is called an activity. Each activity runs to completion, which means that, from the point of view of other objects, each activity finishes before another can be processed. These rules are still quite loose (in the action semantics), but they allow for the concurrent execution of synchronous invocations on an object at the same time that a state chart executes. The state chart semantics specify execution of the activity on exit from a state, then the transition activity, then the activity on entry to a state *for flat state charts*. For hierarchical state charts the rules are more complex, and I personally remain unconvinced that they are fully defined in a standard manner. Certainly, I can't retain the rules long enough to work it out.

The foundation for actions is a data flow model close to that proposed in Shlaer and Mellor [4]. Shlaer-Mellor allows for data flows comprising multiple data elements, while the action semantics allow for results to be generated on pins connected by data/object flows. Consequently, there can be several flows between the same two actions.

A value on an output pin can only be written once, but read many times, which is just another way of saying that the action semantics rely on data flow. Re-consider for example the code fragment from above: `x = 2 * a; b = x * x`. As we indicated earlier, the assignment to `x` is not necessary. Were we to replace `x` with `x'`, the meaning of the fragment would be exactly the same. We can always (logically) replace a data flow with another variable, thus leading to the original statement that we can write once, but read many times. This property corresponds to the "Static Single Assignment" property in programming language theory.

Control flows impose sequence on actions. They can be combined with data flows.

Actions may be composed (recursively) for control, for conditionals, for loops, or for any other arbitrary reason.

All data access is modeled explicitly using actions. This facilitates re-organization of actions in an implementation. (See the requirement to separate data access from functional computation above.)

The submission does not define primitives, such as addition, multiplication, string concatenation, Bessel functions, and the like. There are actions to invoke these primitive actions and other actions that are not modeled further.

Some actions act on *collections*. A collection is a fancy UML term for either a *set*, in which each element is required to be distinct, or a *bag*, in which there is no such requirement. Four types of action are defined on collections. `Filter` takes a collection of elements and a test on some value to produce a subset of elements, only those for which the test is true. For example,

```
ReadAll( Switch.Circuit )| Filter( Circuit.Status != #Busy)
```

reads all the circuits connected to a switch, "pipes" the result to a filter that passes only those circuits for which the status is not busy. (Syntax is not specified in the standard.)

5. The Repository

To capture semantics, as distinct from notation, the meaning of the model, as distinct to the diagrams, must be captured in a repository of some sort. The logical structure of the repository mirrors the semantic rules described in the sections above, including the semantics of actions.

The executable UML metamodel is a model of executable UML using UML. It has classes such as Class, Attribute, Event, Signal, State, Action, CreateAction, ReadAction—all of the concepts we have discussed to define UML in English.

When we draw a class such as Batch in a developer model, this creates, presumably using some model building tool, an instance of the class Class, with data describing the class so created, such as a Name ("Batch"), a description, and the like. (It could also have graphical information that describes the location of the box and so on, but that is not of concern to us in the semantics.) Similarly, when we create an attribute amountOfBatch of the Batch class, this creates an instance of the class Attribute with name (amountOfBatch), the class it describes (a reference to Batch), a type (amount), and so on.

The precise structure of the repository is important to the developers of model building tools and if you, or a model compiler builder, need to traverse the repository to produce code.

6. Model Compilers

A model compiler comprises two main components, an execution engine and a set of rules. A model compiler relies on the existence of some repository and some language that can traverse an arbitrary repository and produce text. The rule language is processed by a "generator."

An execution engine is a specific set of reusable components that, when taken together, are capable of executing an arbitrary executable UML model. The execution engine will therefore contain ways of storing instances in some form, possibly as objects, but not necessarily; some way of sending signals, some way of recognizing event, some way of reading an attribute, and so forth. The selection of the elements in the execution engine determine the system properties, such as concurrency and sequentialization, multi-processing and multi-tasking, persistence, data organization and data structure choices. These choices, together with the pattern of usages in the application, determine the performance of the system.

The second part of the model compiler is a set of rules. As an example, the rule below generates code for private data members of a class by selecting all related attributes and iterating over them. All lines beginning with a period ('.') are commands to the generator, which traverses a repository containing the executable model and performs text substitutions.

```
.Function PrivateDataMember( class Class )
.select many PrivateDataMember from instances of Attribute related to Class
.for each PrivateDataMember
${PrivateDataMember.Type} ${ PrivateDataMember.Name};
.endfor
```

`${PrivateDataMember.Type}` recovers the type of the attribute, and substitutes it on the output stream. Similarly, the fragment `${ PrivateDataMember.Name}` substitutes the name of the attribute. Finally, the lone ; is just text, copied without change onto the output stream.

For a more complete example, consider:

```
.select many stateS related to instances of
      class->[R13]StateChart ->[R14]State
            where (isFinal == False);
public:
  enum states_e
    { NO_STATE = 0 ,
.for each state in stateS       .if ( not last stateS )
            ${state.Name } ,
       .else
            NUM_STATES = ${state.Name}
       .endif;
.endfor;
};
```

which produces:

```
public:
  enum states_e
    {  NO_STATE = 0 ,
       Ready ,
       Executing ,
       NUM_STATES = Complete
    };
```

In the rule, we used italics for instance references, which are like variables that refer to instances in the repository; underlining to refer to names of classes and attributes in the repository; and capitalization to distinguish between collections of instances vs. individual ones.

You may wonder what the code is that is produced is for. It is an enumeration states with a variable `num_states` automatically set to be the count for the number of elements in the enumeration. This enumeration is used to dimension an array that contains the pointers to the activity to be executed. You may not like this code. Cool: to change it all you have to do is modify the rule and regenerate. Every single place where this code was written will then be changed.

While the "result" is only about half the size of the rule, the rule can generate any number of these enumerations., all in the same way, all conforming to your coding standards, all equally right—or wrong.

7. Model Driven Architecture

Executable UML only carries us so far. We need also be able to interchange the models between tools. XML is a data interchange tool that relies on data type definitions (DTDs) to define the semantics of the interchanged data. This definition, for the UML in XML, is called XMI. However, XMI is still insufficient because it allows any syntactically correct UML model to be interchanged, and what we need is to be able to interchange standard semantics for execution.

Executable UML enables a market in model compilers, because a standard interchange mechanism for standard semantics means that any vendor who builds a model compiler will have access to all executable UML models.
Each model compiler can then target a specific platform. For example, one model compiler can target small embedded systems generating highly optimized C straight onto the silicon without an operating system, while another can generate multi-tasking C++ with persistence capabilities. Still others can generate VHDL from executable UML models.

8. What To Take Away

Executable UML models systems at a higher level of abstraction, thus avoiding the costs involved in making premature design decisions. This higher level of abstraction has to have rules, rules that make the models simple and regular, and thus amenable to compilation. There is no need to make executable UML have notations for every single packaging, every single, design construct, and every single way of describing behavior. Rather, executable UML contains only what is required for a specification language.

The basic structure of an executable UML model is that each class has a state chart, and activities containing actions hang off a limited number of places in a state chart. The action semantics are a part of UML (or should be by the time you read this!)

A model compiler is a executable UML compiler comprising an execution engine and a set of rules. The rules read the repository to produce code. Model compilers, like programming language compilers, can be bought as object code, or they may be bought as source, offering the modifiability of the rules to make the generated code more efficient.

To make a market in model compilers, there has to be interchangeability of models, then every model compiler will be able to compile every executable UML model. It also enables other tools: tolls that check for reachability or decideability, tools that can transform models by, for example, combining state machines or breaking them apart, and so on. This will, if all goes well, fundamentally change how we build software.

Finally, you have to see a complete example of an Executable UML model, Leon Starr's excellent book Executable UML [6]

9. References

[1] The UML Summary. See www.omg.org document ad/97-08-04 and friends

[2] Rodney Bell, *Code Generation from Object Models*, Embedded Systems Programming, March 1998.

[3] Action Semantics RFP, See www.omg.org, document ad/98-11-01

[4] Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice Hall, 1992

[5] Action Semantics Submission. See www.omg.org, document ad/2001-08-04[6] Leon Starr, *Executable UML*, Model Integration, LLC.; 2 CDs incl., ISBN: 0970804407