Executable and Translatable UML[1]

Stephen J. Mellor
Mentor Graphics Corp.

Executable and Translatable Unified Modeling Language (xtUML), which is a profile of UML, adds a standard execution model for a tractable subset of the language so developers can formally test models to reduce defect rates from early execution of a target-independent application model before decisions have been made about implementation technologies. xtUML separates application and software architecture design so that each can evolve, be maintained, modified, and reused separately and concurrently. This yields significant reductions in development cost and time to market. In xtUML, design is expressed as a set of transformation rules, providing automatic model translation. The transformation rules are applied either uniformly or to model elements marked to indicate which rule to apply. This allows optimized patterns to be propagated throughout the code, providing powerful performance tuning and resource optimization. It also allows for retargeting models to different implementation technologies as they change. This article describes these fundamental ideas behind xtUML, and how they work in practice.

As its key idea, Executable and Translatable Unified Modeling Language (xtUML) separates application and software architecture design and weaves them together only at deployment time via the following:

- Application models capture what the application does clearly and precisely. The models are executable, including details relevant to the application but independent of the software platform (i.e., design and implementation details).
- Software architecture designs – defined in terms of transformation rules and execution engine components – are incorporated by a generator that produces code for the target system. The software architecture designs are completely independent of the applications they support.
- A generator, which may be human, weaves the application models and the execution engine components resulting in 100 percent complete code for modeled components.

The complete separation of the software architecture design from the application models supports concurrent design of the application and the software architecture, compressing the development schedule and time to market.

These benefits accrue partly as a result of simplifying the tasks of *analysis* and *design* because each can be carried out separately. In particular, design is the definition of a set of transformations that can be applied to the various analysis elements. Each design transformation rule thus applies to all matching patterns in the application, significantly simplifying the design task. This also enables automation, which yields greater benefits. For that reason, the article assumes automation here.

xtUML Notation

xtUML defines a carefully selected, streamlined subset of UML to support the needs of execution- and translation-based development, which is enforced not by convention but by execution: Either a model compiles, or it does not.

The notational subset has an underlying execution model. All diagrams (e.g., class diagrams, state diagrams, activity specifications) are *projections* or *views* of this underlying model. Other UML models that do not support execution such as use-case diagrams may be used freely to help build the xtUML models.

> *"... just as programming languages conferred independence from the hardware platform, xtUML confers independence from the software platform, which makes xtUML models portable across multiple development environments."*

The essential components of xtUML are illustrated in Figure 1 (see page 20), which shows a set of classes and objects that use state machines to communicate. Each state machine has a set of actions triggered by the state changes in the state machines that cause synchronization, data access, and functional computations to be executed.

A complete set of actions makes UML a computationally complete specification language with a defined *abstract syntax* for creating objects, sending signals to them, accessing data about instances, and executing general computations. An action language *concrete syntax*[2] provides a notation for expressing these computations.

An xtUML model with actions is not a blueprint to be rewritten or filled out by programmers, but an executable specification. The difference between an ordinary programming language and a UML action language is analogous to the difference between assembly code and a programming language. They both completely specify the work to be done, but they do so at different levels of language abstraction. Programming languages abstract away details of the hardware platform so you can write what needs to be done without having to worry about things such as the number of registers on the target machine, the structure of the stack, or how parameters are passed to functions. The existence of standards also makes programs portable across multiple machines.

xtUML allows developers to model the underlying semantics of a subject matter without having to worry about such things as the number of processors, the data-structure organization, or the number of threads. In other words, just as programming languages conferred independence from the *hardware* platform, xtUML confers independence from the *software* platform, which makes xtUML models portable across multiple development environments.

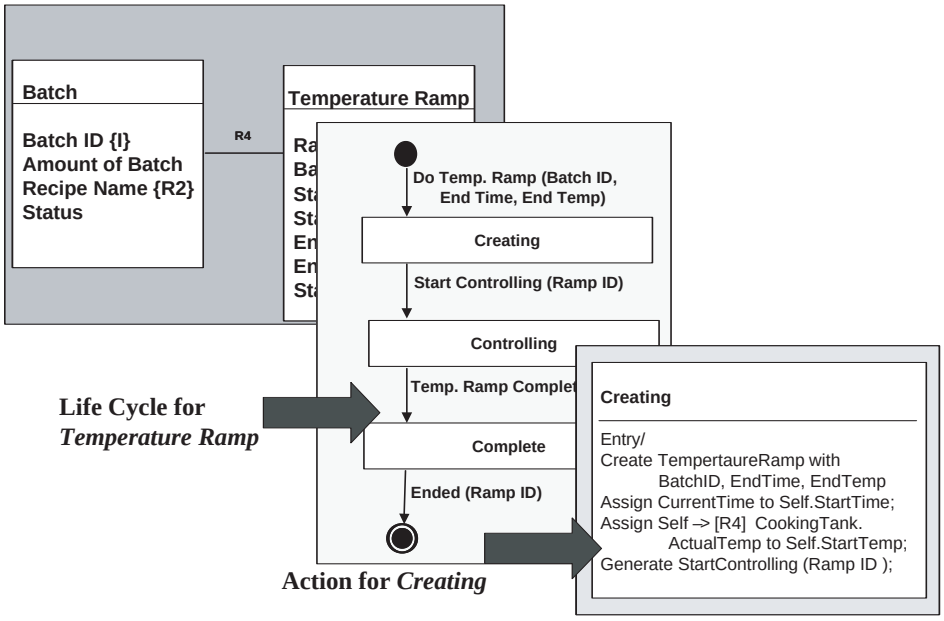


Figure 1: *The Structure of an xtUML Model*

xtUML Dynamics

Figure 1 shows the static structure of xtUML, but a language is not meaningful unless there is a definition of the dynamics, or how the language executes at run time. To execute and translate, xtUML must have well-defined execution semantics – and it does. xtUML has specific, unambiguous rules regarding dynamic behavior, stated in terms of a set of communicating state machines, the only active elements in an xtUML program.

Each object and class (potentially) has a state machine that captures the behavior over time of each object and class. Every state machine is in exactly one state at a time, and all state machines execute concurrently with respect to one another. A state machine synchronizes its behavior with another by sending a signal that is interpreted by the receiver's state machine as an event. On receipt of a signal, a state machine fires a transition and executes an activity, a set of actions that must run to completion before the next event is processed.

State machines communicate only by signals, and signal order is preserved between sender and receiver instance pairs. The rule simply enforces the desired sequence of activities. When the event causes a transition in the receiver, the activity in the destination state of the receiver executes *after the action that sent the signal*. This captures desired cause and effect in the system's behavior. It is a wholly separate problem to guarantee that signals do not get out of order, links fail, etc., just as it is a separate problem to ensure sequential execution of instructions in a parallel machine.

Each activity comprises a set of actions such as a computation, a signal send, or a data access. The semantics of these actions are defined so that data structures can be changed at translation time without affecting the definition of computation – a critical requirement for translatability. The actions in each activity execute concurrently unless otherwise constrained by data or control flow, and these actions may access data of other objects. It is the proper task of the modeler to specify the correct sequencing and to ensure object data consistency.

The application model, therefore, contains the details necessary to support application model execution verification and validation, independent of design and implementation. No design details or code needs to be developed or added for model execution: Formal test cases can be executed against the model to verify that application requirements have been properly addressed.

Those are the rules, but what is really going on is that xtUML is a concurrent specification language. Rules about synchronization and object data consistency are simply rules for that language, just as in C++ we execute one statement after another and data is accessed one statement at a time. We specify in such a concurrent language so that we may *translate it* onto concurrent, distributed platforms, as well as fully synchronous, single-tasking environments.

At system construction time, the conceptual objects are mapped to threads and processors. The generator's job is to maintain the desired sequencing specified in the application models, but it also may choose to distribute objects, sequentialize them, even duplicate them redundantly, or split them apart *so long as the defined behavior is preserved.*

Model Compilers

A model compiler automatically generates target-optimized, 100 percent complete code from models. A model compiler comprises two main components. First, there is an execution engine that supplies the execution infrastructure such as storage schemes, action invocation, and signal sending. 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 invoking an action; some way of sending signals; some way of reading an attribute; and so forth. The selection of the elements in the execution engine determines the system properties such as concurrency and sequentialization, multiprocessing and multitasking, persistence, data organization, and data structure choices. These choices, together with the pattern of usages in the application, determine the performance of the system.

Second, a set of *archetypes* specifies how to translate an application model into code. Archetypes are a formalization of the design patterns and translation rules. The archetype describes when it should be used, the set of patterns to be applied in code generation, and how model components will be populated or utilized to build code. (Archetype examples are provided in the next section.) The combination of translated application code, legacy code that is linked, and the execution engine constitutes the runtime system.

The archetypes use a generator to traverse an arbitrary repository and produce text. The repository contains the meaning of application model, distinct from the diagrams. (The repository will maintain graphical information too, but that is not of concern for generation.) The logical structure of the repository (the *metamodel*) mirrors the semantic rules described in the previous sections, including the semantics of actions – which means that the repository contains the entire, detailed application model.

The metamodel is a model of xtUML using UML. It has classes such as Class,

Attribute, Event, State, Action, CreateAction, and ReadAction – all the concepts we have discussed. When we draw a class such as Batch in a developer model using an xtUML-aware model building tool, it creates an instance of Class, with data describing the class so created such as a Name (Batch), a description, and the like. Similarly, when we create an attribute amount of Batch, this creates an instance of Attribute with name (amount), the class it describes (a reference to Batch), and a type (quantity).

The generator is a translation engine that extracts application model information, interprets the archetypes, and performs the mapping of model components to generate complete code. (Recall that the repository contains the actions too.) The partitioning of model compilers into these pieces streamlines their customization, construction, and maintenance. Changes and additions can be made to the archetypes or run-time library without having to contend with the details of generator or repository management.

Generator Operation

The generator and the archetypes constitute a compilation environment. When generating code, the generator extracts information from the application model. The generator then selects the appropriate archetype for the *to-be-translated* model element. The information extracted from this model is then used to *fill in the blanks* of the selected archetype. The result is a fully coded model component.

The archetypes are applied either uniformly to certain kinds of model elements (all classes, say), or to model elements that have been marked to indicate which rule to apply. For example, a class could be marked to indicate the processor in which it resides, or a state chart could be marked to show which storage scheme (a list or a table) to use, and so on. Using marks provides complete control over the output and enables performance optimization at any level.

Population of an archetype commonly requires invocation of other archetypes. These newly invoked archetypes, in turn, often invoke other archetypes. The creation of code, for what appears to be one model element, can ultimately involve several nested layers of archetypes for multiple model elements. This is fully automated by the generator. This simple approach is incredibly powerful for real-life applications.

The simple archetype in Table 1 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 pdmS from instances of Attribute related to class;
.for each pdm in pdmS
${pdm.Type} ${ pdm.Name};
.endfor
```

Table 1: *Simple Archetype*

Archetype	Generated Code
.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; };	public: enum states_e {NO_STATE = 0 , Filling, Cooking, NUM_STATES = Emptying };

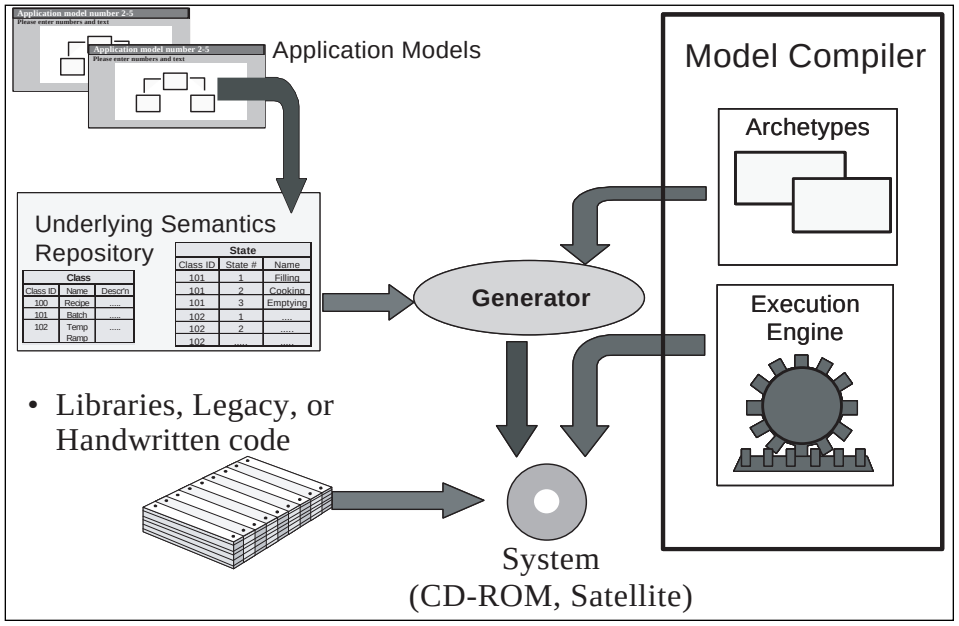
Table 2: *Example Archetype*

${pdm.Type}, as shown in Table 1, recovers the type of the attribute and substitutes it on the output stream. Similarly, the fragment ${pdm.Name} substitutes the name of the attribute. The space that separates them and the lone semicolon (;) is just text, copied without change, onto the output stream.

In the more complete example in Table 2, the archetype uses italics for references to instances in the repository, underlining to refer to names of classes and attributes in the repository, and noticeably different capitalization to distinguish between collections of instances versus individual ones.

You may wonder what the produced code is for. It is an enumeration of states with a variable num_states automatically set to be the count for the number of elements in the enumeration. (There is a similar archetype that produces an enumeration of signals.) The enumerations are used to declare a two-dimensional array containing the pointers to the activity to be executed. You may not like this code, or you may have a better way to do it. Cool. All you have to do is modify the archetype and regenerate. Every single place where this code appears will then be

Figure 2: *Components of an Automated Solution*



changed. Propagating changes this way enables rapid performance optimization.

While the generated code is less than half the size of the archetype, the archetype can generate any number of these enumerations, all in the same way, all equally right or wrong.

Because the archetype is a data access and text manipulation language, it can be used in conjunction with the generator to generate code in any language: C, C++, Java, or Ada, and, if you know the syntax, Klingon. We have used xtUML to generate Very High Speed Integrated Circuit Hardware Description Language.

System Construction

Figure 2 shows how the pieces fit together. To build a system, developers build xtUML models that specify the desired behavior of the application, and buy a model compiler comprising a set of archetypes and an execution engine. The compilation process proceeds in two phases. First, the archetypes traverse the model as stored in the repository to produce source code. Second, the generated code is compiled with the execution engine library and any handwritten, legacy, or library code. The result is the system.

Examples

xtUML has been used on over 1,400 real-time and technical projects, including life-critical implanted medical devices, Department of Defense flight-critical systems, 24x7 performance-critical fault-tolerant telecom systems, highly resource-constrained consumer electronics, and large-scale discrete-event simulation systems.

One telecommunications switch project with an in-house model compiler generated in excess of 4 million lines of C++. One hundred percent of the modeled code was generated, comprising over 80 percent of the total system code. The system was extremely real-time, fault-tolerant, and used over 1,000 distributed processors [1].

A consumer electronic project compared handwritten code with a model compiler. The model compiler generated faster code than the handwritten version, though it was slightly bigger. This difference was traced to different choices made for caching variables. Both the handwritten code and the generated code met performance and size constraints [2].

A joint forces wargaming system built xtUML models of the maritime portion of the battlespace and translated them into C++ that runs on an high level architecture-based distributed discrete event-simulation engine. The target platforms were UNIX workstations and Windows boxes. They used a customized version of MC-2020 as the base for the model compiler. One portion of the simulation uses a special purpose simulation language generated by archetypes. The model compiler was derived from a C++ model compiler with similar system characteristics.

Another organization that has built its own model compiler, for sale, with sophisticated transaction safety and rollback features reports between seven and 10 lines of generated C++ for each line of action language. More importantly, all that delicate code is known to be correct; it is not handcrafted by fallible, or worse, *creative* coders.

For a completely worked out, publicly available example model, see "Executable UML: The Case Study" [3].

xtUML Capabilities

xtUML provides a unique opportunity to accelerate development and improve the quality, performance, and resource utilization of real-time, embedded, simulation, and technical systems. The approach provides for the following:

- Fully customizable translation generating 100 percent complete, target-optimized code.
- Reduced defect rates from early execution of target-independent application models by an average of 10 times (*not* 10 percent).
- Accelerated development of products with multiple releases, growing or changing requirements, and families of products.
- Concurrent design and application analysis modeling to compress project schedules.
- Powerful performance tuning and resource optimization.
- Effective, practical reuse of target-independent application models.
- Effective, practical reuse of application-independent designs.
- Reduced maintenance costs and extended product lifetimes.

This article described the fundamental ideas behind xtUML and how it works in practice. These ideas are more fully described in [4].◆

References

1. Case, J., and John R. Wolfe. "Modeling Accelerates Optical Networking System Development." Project Technology Inc. <www.projtech.com/pdfs/success/tellabs.pdf>.
2. Yamaguchi, Minoru. "Recursive Design for Real-Time Embedded Systems." Sony Corp. 2001 <www.projtech.com/pubs/confs/2001.html>, then access yamaguchi.zip.
3. Starr, Leon. Executable UML: The Case Study. 2nd ed. Model Integration LLC, 21 Feb. 2001.
4. Mellor, Stephen J., and Marc J. Balcer. Executable UML: Foundation for Model-Driven Architecture. 1st ed. Addison-Wesley Professional, 15 May 2002.

Note

1. Parts of this article were derived from the draft for "MDA Distilled: Principles of Model-Driven Architecture" by S.J. Mellor, K. Scott, A. Uhl, and D. Weise, Addison-Wesley, 2004.
2. BridgePoint provides an Object Action Language that is compliant with the abstract syntax standard, but there is presently no action language (notation) standard.

About the Author

Stephen J. Mellor is chief scientist of the Embedded Systems Division at Mentor Graphics. He is an internationally recognized pioneer in creating effective engineering approaches to software development. Mellor is active in the Object Management Group, chairing the consortium that added executable actions to the Unified Modeling Language (UML), and is now active in specifying model-driven architecture (MDA). He is chairman of the "IEEE Software" industrial advisory board and is a signatory to the "Agile Manifesto." He is author of "Executable UML: A Foundation for Model-Driven Architecture," "MDA Distilled," and publisher of the 1985 Ward-Mellor trilogy "Structured Development for Real-Time Systems," and the first books defining object-oriented analysis in 1988. Mellor co-founded a company focused on tools to execute and translate xtUML models that is now a part of the Embedded Systems Division of Mentor Graphics where he is chief scientist.

E-mail: stephen_mellor @mentorg.com