Agile MDA

Stephen J. Mellor
steve@projtech.com

Project Technology, Inc.
www.projtech.com

MDA is a broad church covering a number of different approaches to model-driven development. Most commonly, people think of models as blueprints that are filled in with code, and model-driven development supporting automation of transformations between these several models. That is, MDA is commonly viewed as supporting "heavyweight" process-heavy modeling techniques; but MDA can do better than this.

Agile MDA is based on the notion that code and executable models are operationally the same. Hence, the principles of the Agile Alliance—testing first, immediate execution, racing down the chain from analysis to implementation in short cycles, for example—can be applied equally to models. An executable model, because it is executable, can be constructed, run, tested and modified in short incremental, iterative cycles.

To reach this happy state, models must be complete enough that they can be executed standing alone. There are no "analysis" or "design" models, because all models are equal. Models are linked together, rather than transformed, and they are then all mapped to a single combined model that is then translated into code according to a single system architecture. This approach to MDA is called **Agile MDA**.

What is Agile MDA?

To some, the notion of putting "agile" and "modeling" in the same sentence makes no sense. The modelers worry that "agile" is a synonym for "hacker" in its most pejorative sense, while the agilists see lumbering heavyweight processes (and quite possibly lumbering heavyweight methodologists) that deliver the wrong system late at great expense.

One reason for the disconnect is the recognition of the *verification gap*. This gap comes about when we write documents that cannot execute. Certainly, we can review them and draw conclusions about their correctness, but until we have something that runs we cannot know for a fact that they really do exactly what is needed. In addition, in the time it takes to deliver a solution, the market and the technology have moved on, making the delivered system, even if it is correct, irrelevant. Worse, some systems are "wicked," in that the existence of a solution changes the (perception of the) problem, which makes a complete and detailed specification document somewhat futile.

Agile methods propose to address these problems by delivering small slices of working code as soon as possible. This working functionality is immediately useful to the customer, and it can be interacted with, possibly improving understanding of the system that needs to be built. As these

delivery cycles can be short (a week or two), the systems' development process is able to adapt to changing conditions and deliver just what the customer wants.

To increase customer involvement, agile processes encourage customer participation even while programming, but no one suggests they help write assembly code for it is at too low a level of abstraction. Of course, customers aren't stupid: they certainly could learn assembly code, but they wouldn't (and shouldn't) want to, because this language is far from their concerns, involving register allocation and heap management, and all sorts of interesting tricks far removed from banking, telephony, a copier, or whatever the customer's application is.

Java, Smalltalk, and C++ are higher level, but they still call for consideration of concepts of no interest to a customer: list structure, distribution strategies, the niceties of remote procedure calls and so on. To eliminate the verification gap and enable immediate delivery of fragments of the system, what we need is a highly abstract modeling language that focuses on a single subject matter—the subject matter of interest to the customer—and yet is specific and concrete enough to be executed, an *executable model*. This executable model is at a higher level of language abstraction than Java and Smalltalk, just as they in turn are at a higher level of abstraction than C or assembly code.

Executable models are neither sketches nor blueprints; as their name suggests, models run. This fact eliminates the verification gap, and allows us to deliver a running system in small increments in direct communication with customers. In this sense, *executable models act just like code*, though they provide the ability to interact better with the customer's domain. This executable model is platform independent. In the parlance of MDA, it's a platform-independent model (PIM)

Building an Executable UML Model

Executable UML is a profile of UML that defines an execution semantics for a carefully selected streamlined subset of UML. The subset is computationally complete, so an executable UML model can be directly executed. Modeling rules are enforced not by convention but by execution: Either a model compiles and runs, or it doesn't.

All diagrams (*e.g.* class diagrams, state diagrams, procedure specifications) are "projections" or "views" of an underlying model. UML models that do not support execution, such as use case diagrams, may be used freely to help build the executable UML models.

The essential components of executable UML are illustrated in Figure 1, 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" provides the notation for expressing these computations. Because executable UML is computationally complete, it can be used as a PIM to specify any subject matter in the system.

The difference between an ordinary boring programming language and a UML action language is analogous to the difference between assembly code and a programming language. They both specify completely 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 the number of registers on the target machine, the structure of the stack or how parameters are passed to functions, etc. Action languages abstract away details of the software platform so you can write what needs to be done without worrying about distribution strategies, list structure, remote procedure calls, and the like. For example,

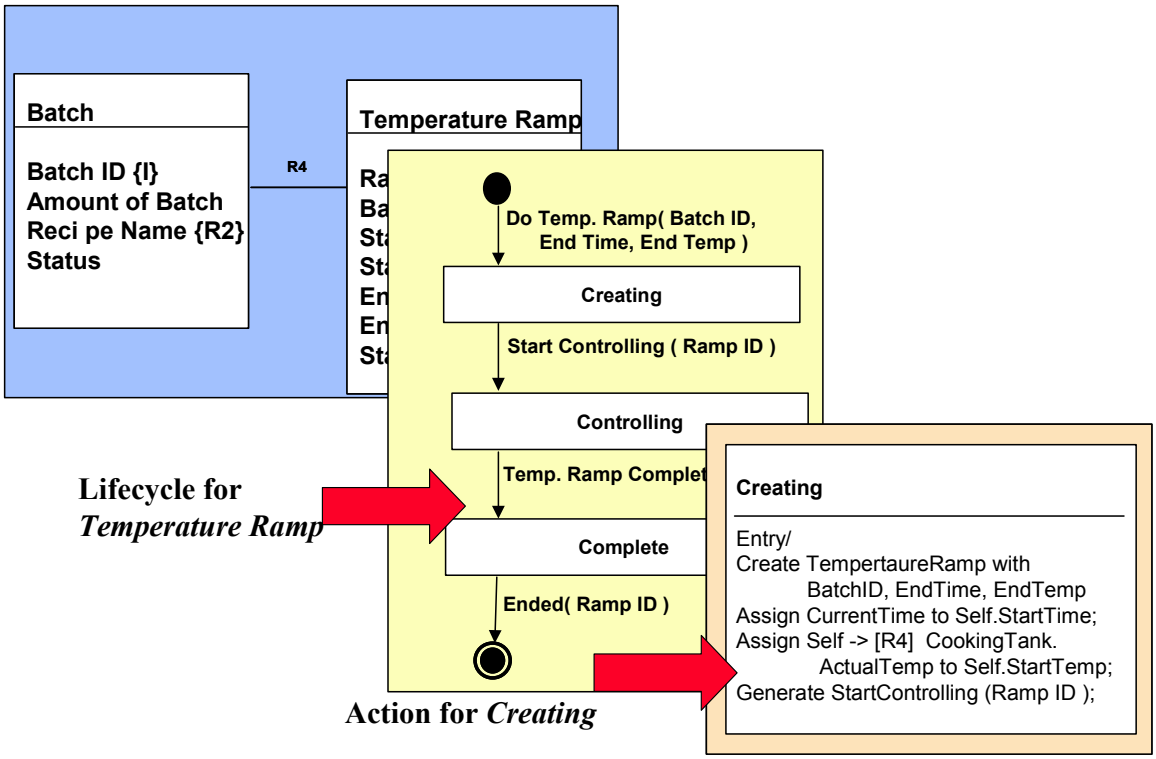

Figure 1: The primary projections of Executable UML

action languages do not concern themselves with the cardinality of the data structures, treating everything as a set. It is only when it comes time to transform these structures into implementation do we select an appropriate physical data structure. Similarly, with the patterns of access to data, we generate only what we need after examining whether a type of access is actually used.

As the existence of standards made programs portable across multiple hardware platforms, so too does the existence of an executable UML standard make models portable across multiple software platforms.

Executing an Executable Model

Figure 1 showed the static structure of an executable UML, but a language is not meaningful unless there is a well-defined execution semantics. Executable UML has specific unambiguous rules regarding dynamic behavior—how the language executes at run time—stated in terms of a set of communicating state machines, the only active elements in an executable UML 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.

The actions in each procedure 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.

Coding vs. Actions

Why use an action model rather than just write code?

The semantics of actions are defined so that data structures can be changed at translation time without affecting the definition of the computation—a critical requirement for translatability. Therefore, the action semantics allows you to specify behavior without relying on knowledge of the implementation. For example, a common approach to finding the total amount of the last ten transactions is to loop through the data structure creating a sum as we go. This inextricably links the computation to the data structure, but what if it changes? The action semantics casts this example problem instead as: Get the last ten transaction amounts, then sum them. The consequence of this small change in view is that it is possible to change the storage scheme without affecting the algorithm that merely sums values.

And the result of that is an executable UML model can be translated to any target.

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 a procedure, 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 states the desired sequence of activities. When the event causes a transition in the receiver, the procedure 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 separate problem to ensure sequential execution of instructions in a parallel machine.

An arbitrary model therefore contains the details necessary to support its execution, verification, and validation, independent of implementation. No design details or code need be developed or added for model execution, so formal test cases can be executed against the model to verify that requirements have been properly addressed. This form of verification can be carried out on each executable model, independently of the other models.

Them's the rules, but what is really going on is that executable UML 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.

Translating a Model

Executable UML defines groupings of data and behavior ("classes"), the behavior over time of instances ("state charts"), and for precise computational behavior ("actions"). The reason for the quotation marks is that executable UML does *not* prescribe implementation. Rather, a "class" in executable UML represents a conceptual grouping of data and behavior that *may* be implemented as a class, or it may be implemented as a C struct and a set of associated functions, or as a VHDL entity. That is, a "class" doesn't have to be implemented as a class. Consequently executable UML is a software-platform-independent language that can be translated into *any* target. For this reason, we also use the word "translatable" as well as "executable."

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

Models, Models, Models

There are at least three meanings of "model," and each one denotes diverse usages and connotes different processes. One denotation for the word "model" is a sketch. We sketch out the shape of a wing on the back of a beer mat, show a few lines indicating air flow, and write an equation or two describing how the two interact. The sketch is not complete, nor is it intended to be. The purpose of the sketch is to try out an idea. The sketch is neither maintained nor delivered.

Agile exponents are willing to sketch out their classes and use cases, sometimes called "stories," and perhaps even use UML to do it. There's no fight here: even the most extreme use sketches to outline their plans for the code.

A second denotation for the word is the model as blueprint. The physical model of the airplane in a wind tunnel is one example; more commonly, we think of a blueprint as a document describing key properties needed to build the real thing: the blueprint is the embodiment of a plan for construction.

The connotations of a model-as-blueprint cause conflict. The very idea of a "blueprint" evokes images of factories and manufacturing, together with uncreative drones. In an environment that is 80% construction and 20% design, like manufacturing, it makes sense to view the blueprint as the plan that is predictive of the construction work to be done. "Heavyweight" processes have encouraged the idea of model as blueprint; the manufacturing analogy is drawn repeatedly in the Software Engineering Institute's Capability Maturity Model (CMM), for example. But we know software is a creative new-economy thang, not at all like old-fashioned manufacturing. To the contrary, software is known for its creative aspects, more like 80% design and 20% construction. In this case, developers need to be adaptive rather than predictive in their relationship to any model, effectively putting the kibosh on the use of models as blueprints.

The third denotation for the word is the model as an executable. The model of the airplane can be transformed into the real, physical airplane. The transformation requires other inputs, in this case the metal plates, bolts, and screws that make up the body, yet the model is complete in every detail in the one aspect of the problem related to the shape-that-flies. When we build an executable UML model, we have described the behavior of our system just as surely as we had when we wrote a program in Java.

Under this interpretation of "model" as an executable, a program in a high-level language — Java, say — is a model too. The Java program can be transformed into the real thing (byte code). The builder of the model, the programmer in this case, need not know how a Java compiler works, nor what the compiler does to make a program run. Of course, the byte code produced by the compiler is itself a model that can be replaced by ones and zeroes, one layer of abstraction removed, and those ones and zeroes in turn define the desired behavior of the underlying hardware, yet one more layer of abstraction away.

Many of the principles of Extreme Programming (XP) and the Agile Alliance involve process and customer relationships and their management, not code. As such, the agile process principles for the construction of code apply just as well for the construction of executable models. For those principles that do specifically mention "code" or "software," executable UML, under this definition, is code.

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 may to choose to distribute objects, sequentialize them, even duplicate them redundantly, or split them apart, *so long as the defined behavior is preserved*.

A program in C++ is complete and executable but that doesn't do us much good until it has been transformed into some language that can be directly interpreted. We therefore run the program through a series of transformations that preserve the semantic content of the program (otherwise there's an error in the compiler) but express it in a language more oriented to implementation.

The same happens when we build a complete executable model. When we transform a model, tools populate the metamodel for the modeling language at hand. To carry our language example above to extremes, we could use the text of a C++ program to populate the instances of a model of C++ with classes Class, ProtectedMember, StaticMemberFunction and so on. As a result of the next transformation, we would have a model of C, in which instances of StaticMemberFunction and ProtectedMember would both be cast as ordinary functions, although with different signatures.

Such transformations may be continued indefinitely until the final, lowest, most grungy metamodel of them all. The classes in an (assembly-language) metamodel could be Instruction, Registers, MemoryLocation, and so on. The instances in this metamodel contain all the information of all the "higher level" models, but at a low level of subject-matter abstraction. The lowest level metamodel selected could generate C++, Java, C (sharp or natural, we don't care), assembly code or even microcode. This final level is the level that is compiled in text form to a language that can be executed by a model compiler, about which more below.

Merging Models Together

In addition to successive transformation of complete models, models need to be woven together with other models to produce a system. In an elevator system, the elevator model could be expressed as an executable model, but it would not solve the building's problem until it was connected to another complete model of the Transport subject matter. When linked together, and translated into code, the executable models become systems.

To effect this combination, we can define a mapping function. (This mapping is generally a merging or representing mapping, rather then the refining mappings that transform a model from one form to another.) Specifically, we need to establish that one kind of thing in one model "corresponds" to another kind of thing in another model. An example is that the class Teller and the class Customer in the Bank model each correspond to an instance of a Role in the Security model. Another example is that each Account instance in the Bank corresponds to an instance of ProtectedResource in Security. Or a subset of Train instances in a control application corresponds to instances of Icon in a User Interface domain. (These last two are called "counterparts" or "counterpart instances.")

These mappings can be between any two kinds of identifiable entity in an executable UML model. For example, the state Stopped in the Train Control domain corresponds to the enumerated value Red of the attribute Icon.color. This extends to dynamics. A signal is one domain, say, Button(3) pushed in the UserInterface, corresponds to a signal in the Train Control domain, TimeToLeaveStation to Train 47. Signals may map to function calls and vice versa, and functions may map to changes in attribute values. Anything.

Mappings are often directed by **marks** that correlate elements in source and target models. In Mellor and Balcer [1] and earlier work, these correlations (mapping functions and marks) are called *bridges*.

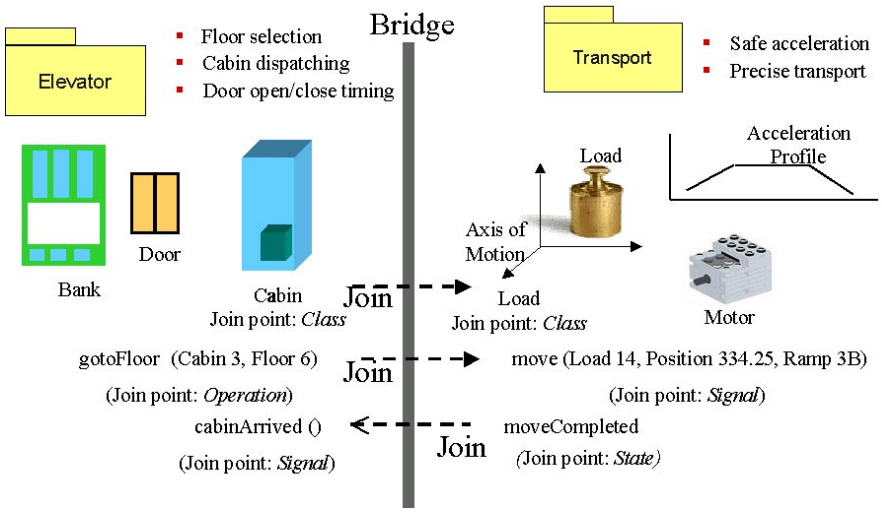

Figure 2: A sketch of a bridge

Once these mappings are defined, we are ready to combine all the models into a single populated metamodel from which code can be generated. (These steps can take place at once.) The mechanism responsible is a model compiler, which weaves together the several models according to a single set of architectural rules.

Model Compilers

A model compiler takes a set of executable UML models and weaves them together according to a *consistent set of rules*. This task involves executing the mapping functions between the various source and target models to produce a single all-encompassing metamodel (aka "the grungiest") that includes all the structure, behavior and logic—everything—in the system.

The final mapping from this metamodel can be done in several ways. One approach to defining mapping functions is to use an *archetype*, which are especially suited to manipulating text.

Weaving the models together at once addresses the problem of architectural mismatch, a term coined by David Garlan to refer to components that do not fit together without the addition of tubes and tubes of glue code, the very problem MDA is intended to avoid! A model compiler imposes a single architectural structure on the system as a whole.

Adding Code Bodies

Executable models are not exactly the same as code (though they are similar to code in an aspect-oriented system) because they need to be woven together with other models to produce a system. However, because each model is complete in itself, once woven, the system is complete.

An alternative approach to building a system is to copy the structure of a PIM into a PSM and then add code bodies. (Obviously, these code bodies need to be "protected" so that a change in the model or the regeneration of one does not cause the disappearance of the hard-won code.) The PSM can also be expressed as a graphical model that can then be manipulated by adding further code details or otherwise reorganizing its structure. There may be several transformations, and code may be added at each transformation. This has the advantage—it is said—that code is added only at the appropriate time, at the appropriate level of abstraction.

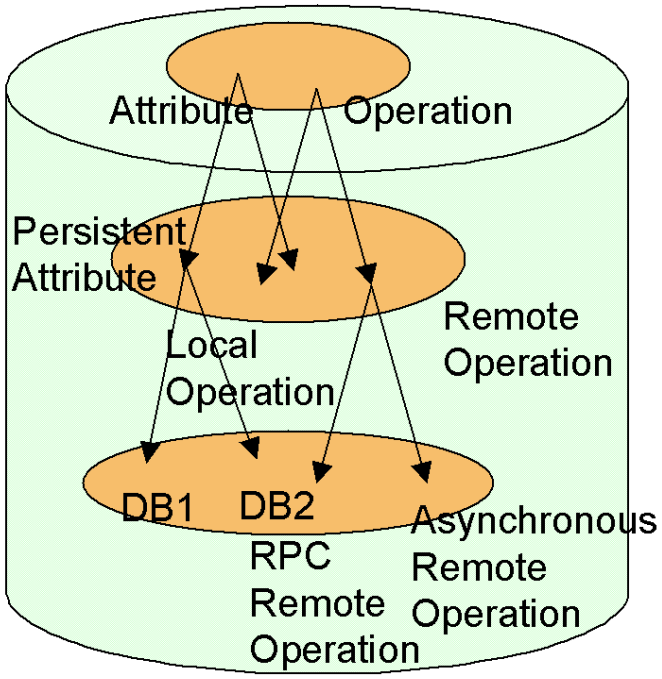

Figure 3: A silo of successive transformations

We may also distinguish this type of MDA by its reliance on *successive transformation*. The typical scenario is to define a profile for a language that has a lower degree of abstraction, as represented by the ovals in Figure 3. For example, a subset of UML may serve as the "analysis" model that has no knowledge of remote or local objects, while another, larger, subset serves as the "design" model that does have knowledge of accessibility. This model is then transformed again into a still less abstract language, perhaps against a profile of a platform such as CORBA.

Yet such code bodies are inherently dependent on the structure of the model for which the code is intended. For example, any code written for the target metamodel that assumes remote procedure calls can only be used in an environment that uses remote procedure calls, even if the subject matter captured by the model is banking. The result of this approach is that models are not universal, but instead form silos that mix the modeling language (the metamodel) and the subject matter at hand. This is another form of architectural mismatch. Agile MDA explicitly addresses this problem by treating all models as equal and merging them together at once.

Obviously, models of this sort cannot be executed. In fact, to be blunt about it, these models are really an automation of a heavyweight process. This is not to deprecate that usage—some people have to do it that way—but it ain't agile!

The Agile MDA approach is to build executable translatable models, which are complete models linked together with others without further intervention or elaboration, by a process of translation.

In Agile MDA, because each executable model is a PIM and the model compiler compiles the models to make code, you might ask what happened to the PSM. The PSM is there alright—it's the code. Code is a weaving together of the elements of the PIM and of the required platforms, and it executes too. In Agile MDA, there is no need to manipulate the PSM or to visualize it as a model. We go straight to the ultimate PSM: the code.

Summary

The conflict between the modelers and agile programmers1 is perceived to be fundamental and large, partly because of differing technical focus—"extreme" is an explicit reaction to deliberate processes—and partly because of hype. Yet in reality the gap is quite small. Many ideas of the Agile Alliance and XP (such as Sustainable Development, Customer on Site, or Estimate to Improve) hold equally for models, and if we simply replace the word "code" with "executable model." Enough, anyway, that your author became a signatory to the Agile Manifesto in good conscience.

To build a system using Agile MDA, we follow an agile process. We build test cases, write executable models, compile the models using a model compiler, run the test cases, and deliver fragments of the system to the customer incrementally. This is just the same as "We build test cases, write code, compile the code using a language compiler, run the test cases, and deliver fragments of the system to the customer incrementally" except that we have replaced one language (code), with another at a higher level of abstraction: an executable and translatable model.

Agile MDA requires the construction of various PIMs using the UML profile called executable UML, and the compilation of those models using a model compiler that executes the mapping functions to produce the most interesting and useful PSM of them all: the code.

References

[1] For more information on this usage for bridges, see *Executable UML: A Foundation for Model-Driven Architecture,* Mellor and Balcer, Addison-Wesley, 2003, Chapter 17: Multiple Domains, and Chapter 18: Model Compilers.

[2] For a description of MDA in general, see *MDA Distilled: Principles of Model-Driven Development*, Mellor, Scott, Uhl and Weise, March 2004. This paper was derived from Chapter 10 and from [3].

[3] Agile MDA, Mellor and Wolfe, 2005

Figure 2 was conceived by Leon Starr, Model Integration, LLC. San Francisco, CA.

[1]For information on the Agile Alliance, see their website www.aanpo.org. The best-known agile method is XP, for which see *Extreme Programming Explained* by Kent Beck.