A Framework for Aspect-Oriented Modeling

Stephen J. Mellor
Project Technology, Inc.
steve@projtech.com
www.projtech.com

This paper proposes a framework for aspect-oriented modeling, and places it in the context of the Object Management Group's Model-Driven Architecture initiative. We assume little in the way of background on aspects, UML, or model-driven architecture; rather, we place several issues into a framework that brings models and aspect orientation together.

There are, of course, many ways of thinking about and applying aspect-oriented concepts in terms of models. This paper focuses on complete, stand-alone, executable and translatable models as the fundamental starting point.

Subject Matters

A key aspect-oriented concept is the "cross-cutting concern." Developers can define a cross-cutting concern in any way, but one key approach is to base them on independent subject matters, or problem domains. Examples are: a Bank, Security, Device I/O, Elevator, User Interface, Train Control, Logging.

> *A* problem domain *is an autonomous, real, hypothetical, or abstract world inhabited by a set of conceptual entities that behave according to characteristic rules and policies.* [1]

Domains are semantically autonomous and they stand alone. For example, an Elevator model should be built without a button class because a Button is not a fundamental part of moving an elevator cabin from one floor to another. Button technology could be replaced by voice recognition or a swipe card that identifies a secure floor. Note that virtually every published Elevator model has a Button class, though an aspect-oriented version certainly should not. Similarly, an Automated Teller Machine and its control, should be completely separated from the bank.

Though domains are autonomous, they also rely on the existence of other domains. For example, the elevator relies on some scheme to tell it which floor to go to. That mechanism could be a user interface that supports buttons, or it could be a voice recognition domain that recognizes the word "Engineering!"

Figure 1 shows that the Elevator domain relies on the existence of domains Transport, to move the elevator from one floor to another, and on a domain user interface, in this case one that uses Button, Panels, and Fields. The dotted lines between the domains indicate the existence of the dependency: a *bridge*.

> bridge *(at specification time) is a layering dependency between domains. One domain makes assumptions, and other domains take those assumptions as requirements.*[1]

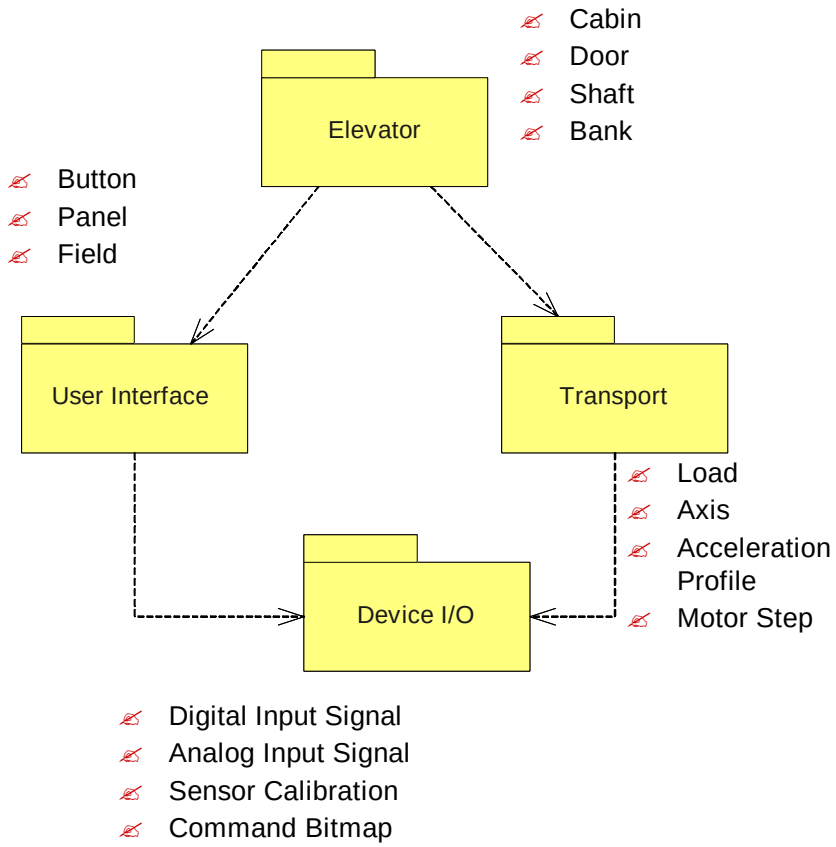

Figure 1: Example Domain Chart with typical classes

The implication is that the elevator domain can be constructed without specific reference to a button or to the mechanisms involved in transporting cabins. In turn, these models have no explicit knowledge of their clients.

The UML package diagram could be used to show packaged and stereotyped dependencies, as shown in Figure 1, though “package” and “dependency” in the way they are normally used in a package diagram is exactly what we do *not* mean. There should be a way to show this form of dependency in UML.

Complete, Executable Models

Aspect-oriented programmers expect their aspects to be executable and complete, even if not yet woven with other aspects to make a working system. The same must hold true for modeling languages, otherwise, obviously, the models would not run. The introduction of the Action Semantics to UML enables execution, but it does not provide a definition for a complete executable modeling language. To achieve that, a profile of UML must be defined that states how model elements fit together.

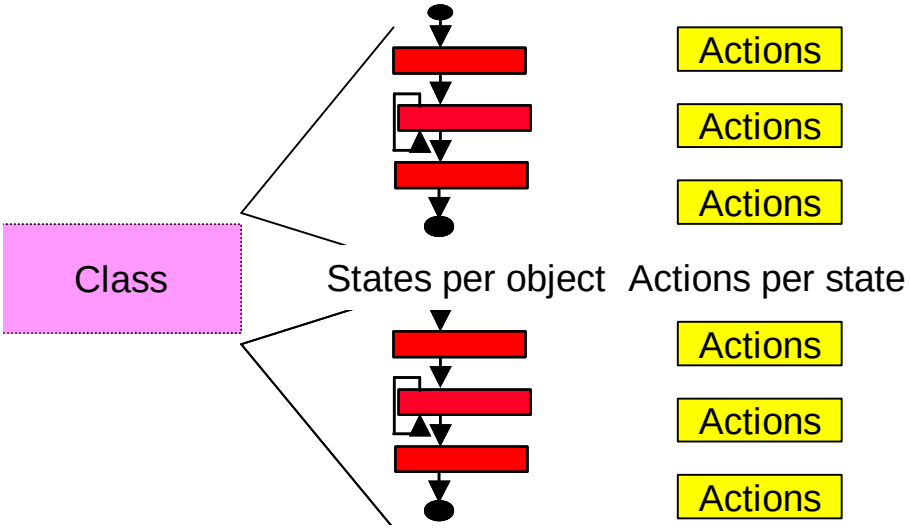

Figure 2: The essential views of Executable UML

An executable UML profile must state what elements form a part of a legal model and how those elements fit together into a coherent whole. In this profile, all diagrams (*e.g.* class diagrams, state diagrams, procedure specs.) are "projections" or "views" of an underlying semantic model. Other UML models that do not support execution may be used freely to help in the construction of the executable UML models.

One such profile is Executable UML [1], which defines an execution semantics for a carefully selected streamlined subset of UML. The essential projections are illustrated in Figure 2, 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.

Figure 2 shows 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." (This paper is not the place to describe exactly what these rules are.)

We build executable models of a single subject matter at a time. In other words, each "package" on the domain chart of Figure 1 is a separate executable model. One domain captures the behavior of the elevator, another of the user interface, another of transport, and so on.

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.

In the parlance of MDA, these *platform-independent models* are called PIMs.

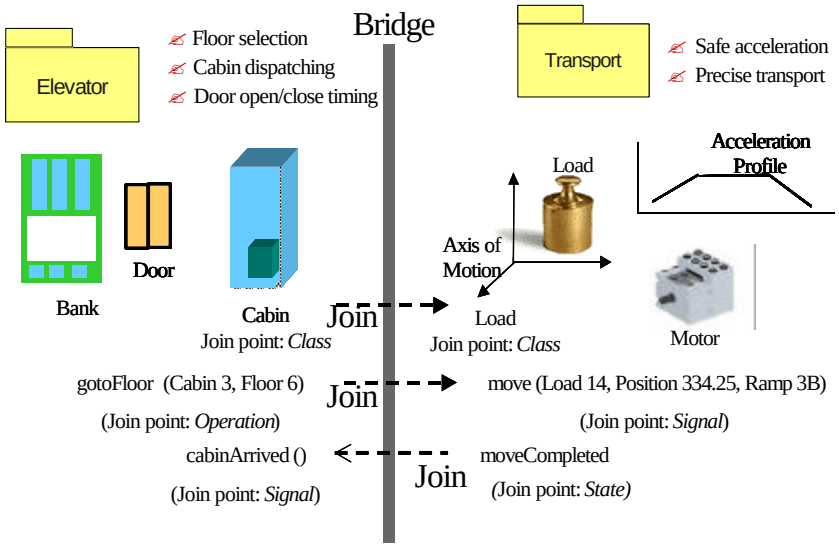

Figure 3: A bridge, join points and joins

Join Points

Domain executable models are not exactly the same as (ordinary, boring, non-aspect-oriented) code because they need to be woven together with other models to produce a system. At compilation time, the models of each domain are woven together by a system of *join points* between the models.

> *A* join point *is some identifiable element in some domain model that can be woven together with a join point in another domain model.*

More colloquially, a join point is some place where you can connect two models together. Some examples are the *class* Cabin that corresponds to the *class* Load; the *object* Cabin 3 that corresponds to *object* Load 14; the *operation* gotoFloor in the Elevator domain that corresponds to a *signal* move in the Transport domain; *arrival in state* moveCompleted in the Transport domain that corresponds to a *signal* cabinArrived in the Elevator domain. The italicized words (class, object, signal, operation, arrival in state) are the join points. These are illustrated in Figure 3.

The join points can refer to elements in the model (Cabin) and the model-at-run-time (Cabin 3). Join points can also be passive (Class) as well as active (Signal). In addition, join points can be attributes or even attribute values (either a range or an enumeration). In short, a join point can be any identifiable element of the model profiles involved: a class, an object, a data access, an attribute, and attribute value, a signal, an operation call, and so on.

Because all domain models are constructed using the same executable UML profile, the size and complexity of describing possible joins is significantly reduced compared to arbitrary-model-to-arbitrary-model joins. In Executable UML, a subset of UML, for example, there are only eighteen identified join points. One shudders to think how many might be needed for the entire UML.

Elevator		Transport	
Join Point	**Instance**	**Join Point**	**Instance**
Class	Cabin	Class	Load
Object	Cabin 3	Object	Load 14
Operation	CabinDispatch::GotoFloor	Signal	AccelProfile::move(Ramp_3B)
Signal	CabinDispatch::CabinArrived	State	AccelProfile::moveCompleted
Object	Floor 6	Attribute Value	Track.(3)::Position = 334.25

Figure 4:A bridge: a set of joins

Bridges and Joins

Bridges exist between domains, linking them, as represented by the dotted lines in Figure 1. At construction time, each bridge comprises a set of joins between join points, as illustrated in the table, Figure 4.

The thick line between columns represents the bridge, just as in Figure 3. On each side of the bridge (it doesn't matter which—bridges are not directional) there is a "half-table" that specifies the kind of join point and the instance. For example, the Cabin in the elevator is an instance of a *class*, and CabinArrived directed to CabinDispatch is a join point of type *signal.*

A join is represented by a row in the table, though as we shall see, there is no requirement that a join be specified in a tabular fashion.

> *A* join *is a specific connection between instances of join points in two domains.*

The first join asserts that a Cabin class in the Elevator domain corresponds to a Load class in transport; that is, each cabin is a load.

The second join asserts that object Cabin 3 is a counterpart to object Load 14. Nothing is said about whether this relationship is established statically or at run time, though from context (cabins are not created on the fly!) we would expect this to be static. Should there be a requirement to create this correlation at run time, we would expect another join from a join point CreateOperation for class Cabin to a join point CreateOperation for class Load.

The third join captures the idea that a functional operation (data accessors *etc.* are considered separate join points) gotoFloor corresponds to a signal to a specific instance (Ramp_3B) of the class AccelerationProfile. Note that the operation applies to each instance of CabinDispatch and the signal specifies acceleration profiles. Each instance of CabinDispatch will correspond to a separate instance of AccelerationProfile (conforming to specification 3B), and that join also needs to be established.

The fourth join maps (arrival in) the state moveCompleted to a signal in the Elevator. Note that the flow of control is *from* the transport domain, which does not affect the mapping in any way. moveCompleted is an abstraction, just like a FloorRequest that is mapped to a Button push.

Where is Floor 6? At some position (334.25) along some track (we chose 3). We will also need to make some joins between Shafts and Tracks at both the class and instance levels.

These examples show that joins can be static (class to class *etc.*), dynamic (signal to operation), and from something static to something dynamic (state to signal). Another example is an attribute that reaches a certain value (range) causing a signal to be generated. These mappings also show that joins can be made between synchronous elements (an operation) to asynchronous ones (a signal). This is critical for enabling models to be built without assuming an underlying implementation software architecture.

The table is intended to be illustrative. For example, the double colon is an informal notation used to define the "owning element." (In Executable UML, signals are always directed to classes and their instances.)

We would be unlikely to specify joins by enumerating every row in a table, for example to correlate each all call with the phone from which it is being made. Rather, we need a compact way of stating joins.

Pointcuts and QVT

The examples illustrated that joins, like mathematical mappings, can be enumerated *(e.g.* Cabin to Load, and Cabin 3 to Load 14), but joins can also be stated by rule. For example, we know each and every cabin will correspond to a load. Instead of expressing this by enumerating each one (Cabin 3 to Load 14, Cabin 4 to Load 15, Cabin 5 to Load 16 etc.), but we could instead state a rule: Cabin i maps to Load i + 11. The rule asserts a set of related joins.

In the aspect-oriented programming community, this kind of rule is a *pointcut*.

> *A* pointcut *is a set of join points and what to do ("advice") when one is encountered.*

In AOP, the pointcut also refers to "advice," which is code that is executed before, after, or around an occurrence of the join point. Put another way, a pointcut defines a rule for a set of joins (conforming to the same join points).

In the modeling context, advice is merely the definition of another join point.Hence, we can say "Every Cabin object corresponds to a Load object, offset by eleven instances."

The modeling version of the pointcut (with join point specifications on both sides) can be expressed as a *query on one model* that maps to a *transformation onto the other*.

This is where MDA comes in. The Object Management Group's Model-Driven Architecture initiative comprises a set of evolving standards that provide for mappings between models. QVT (Query, View, Transformation) is an OMG standard for specifying these mapping presently undergoing the submission process.

There are many proposals extant and it not clear which one will "win" and the extent to which the final standard will support defining joins. Nonetheless, QVT will certainly support much of what is needed, and mapping functions expressed in some variation of QVT are the natural home for defining the mapping between models that constitute a join.

Marks

MDA also specifies the concept of a *mark*, which is a tag that may be appended to any model element. Marks describe models but they are not a part of them, rather like sticky notes.

> *A* mark *is a lightweight, non-intrusive extension to models that captures information required for model transformation and joins without polluting those models.*

An example is the mark Persistent, which may be associated with attributes (say). The pointcut is then defined to refer to all Attribute join points that are also marked Persistent. Now, only those attributes that have this marked will participate in the specification of the pointcut, subsetting the query to only those attributes that must be mapped into persistent storage.

The simplest form of a mark is a discriminator such as Persistent, which is either present or not. Other forms of mark include inputs, such as a prefix to be appended to all operations, or quantities that can be used to determine whether to make one join or another.

At this stage, marks can be implemented using UML profiles, and there are no "marking" standards. This is far too weak for industrial usage. Not only do we need standards, but we also need a "theory" of marks that would enable us to propagate marks (if the class is persistent, so are all it's attributes); handle intersections (if that attribute is not persistent and the accessor is local to this class, then map the accessor to a data member access); and so on.

We suspect that AOP's experience in defining pointcuts can help in this area.

Translatable UML[1]

Once these mappings are defined, we are ready to combine all the models into a single all-encompassing model that includes all the structure, behavior and logic—everything—in the system.from which code can be generated. The mechanism responsible is a model compiler, which weaves together the several models according to a *single consistent set of architectural rules.*

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.

The key issue here is to recognize that the structure of an executable model does not determine the software structure. For example, 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. Similarly, state charts can be flattened, split, duplicated—anything—so long as the implemented behavior is the same as the original model.

1. Some vendors have taken to using "Executable UML" to mean a model with code added in directly, which rather misses the point! For this reason, we use the word "translatable" as well as "executable."

And the same applies for actions. The Action Semantics were deliberately defined so that the specifications of computational functions stand alone, completely separated from control and data structures. For example, a Java program might loop over a set of accounts to produce the total balance for a customer, but in the Action Semantics, this *must* be expressed as: first, find all the account balances for this customer, then sum the values. This small change in perspective means that the data structures can be changed at will without changing the specification of the computation.

Consequently executable UML is a software-platform-independent language that can be translated into *any* target. 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*.

The final mapping from this model is different from the others because it is not a mapping between models, but a mapping from models to text. One approach to defining these mapping functions is to use an *archetype*, which reads the model and produces text. An example is:

Archetype

```
.select many stateS related to instances of
   class->[R13]StateChart ->[R14]State where
      (isFinal == False); // Traverse the metamodel
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;
};
```

The first line (ending with a semicolon) traverses a model to find all non-final states related to the class in question. The next three lines are clear text, which are copied to the output stream. The *for* statement loops over the selected states and substitutes (as shown by ${ ... }) the state's name. The underlined word refer to names in the metamodel of UML (e.g. State, State.name)

The result is shown below:

Generated code

```
public:
  enum states_e
   { NO_STATE = 0 ,
Door_Closed,
Door_Opening,
Door_Open,
NUM_STATES = Door_Closing
    };
```

It remains to be seen whether the adopted QVT standard will be sufficiently general to cover this special case of metamodel-to-text mappings as well as metamodel-to-metamodel. If not, a new MDA standard will have to be proposed.

Conclusion

This paper has proposed a framework for the incorporation of aspect-oriented concepts into modeling and model-driven architecture. MDA's focus on mappings and mapping functions is a natural home for defining join points, but the models we build must be complete and executable to avoid merely messing about with models. To that end, action semantics were defined to ensure easy translation.

There is work to do in the context of UML and MDA (modeling aspects, ensuring mapping functions support joins, defining standards for expressing marks, constructing a theory of marking models, mapping to text and code, *etc.*) to make aspect-orientation a fundamental part of modeling technology, but a framework for so doing has been described here.

Acknowledgements

The original expression of an implementation of a bridge is due to the late Sally Shlaer in her internal Project Technology, Inc. note *Building Bridges*, September 1994.

The elevator examples are drawn from the Leon Starr and his elevator case study *Executable UML: The Elevator Case Study*.[2]

A couple of paragraphs were drawn from *MDA Distilled*, Mellor, Scott, Uhl and Weise, to be published by Addison Wesley Spring 2004.[3]

None of this would have been possible without the practical work of John Wolfe and Campbell McCausland of Project Technology, Inc, and the team at SPAWAR.

References

1. Executable UML: A Foundation for Model-Driven Archicture. Addision-Wesley, 2002
2. *Executable UML: The Elevator Case Study*. Leon Starr
3. *MDA Distilled*, Mellor, Scott, Uhl and Weise, to be published by Addison Wesley, Spring 2004