

Advanced Methods and Tools for a Precise UML

Andy Evans¹ (Moderator),
Steve Cook², Steve Mellor³, Jos Warmer⁴, and Allan Wills⁵

¹ Department of Computer Science, University of York, UK.

² IBM EMEA Object Technology Practice, UK.

³ Project Technology, Inc. US.

⁴ Klasse Objecten, Netherlands.

⁵ Treme International, UK.

1 Introduction

Imagine for a moment you are a software ‘architect’ in the year 2003. You’re working at home as usual, and decide to use your quantum computer to do some system modelling. Imagine also that the UML is ‘still’ the de-facto language for software engineering. As a language it has made some big advances of the last few years. The last three versions (3.0-5.0) have all had a precise semantics - even the mysteries of aggregation have been resolved - and its applicability has been widened to every kind of system imaginable. Standardisation has also been good for the software profession. CASE vendors and methodologists, no longer able to invent new notations, have devoted their energies to building increasingly sophisticated tools and processes. Thus, the tool and methods you are about to use incorporates a maturity of software technology that has never been realised before...

The question for the panel is, what would you expect of the tool you imagine using, and the methods it supports? How will a precise UML (assuming it *is* or *can* be made precise) benefit software development tools and processes over the next five years? What features will be desirable in future tools and methods, and what type of semantics will they require to support these features? How might tool vendors go about building such tools? How might methodologists go about using the semantics to compliment or improve their work?

To answer these questions, four leading methodologists and tool developers will give their vision of the future. A snapshot of their thoughts, and views on the way forward, are given below.

2 Steve Cook

Let’s look at the question “Is rigorous proof achievable in UML” This is an interesting question from a theoretician’s point of view. But from the perspective of today’s typical software developer, such a question would be met with blank incomprehension. Just to understand and formulate the need for a proof is outside the experience of most developers, let alone actually to do one. Twelve years

on from the publication of Eiffel, we have just about reached the point where a reasonable proportion of developers have heard of the idea of a pre-condition, although the number who can actually formulate one correctly is a small subset of these.

So I want to distinguish strongly between precision in the definition of UML and precision in its use. Long experience tells us that normal software developers will not, in the foreseeable future, be willing to use abstract formal languages and notations to design software, regardless of how theoretically desirable it might be to do so. Even such simple semi-formal languages as OCL meet great resistance from the typical developer or student of programming. We might surmise that the reason for this is that the mental interpretation of declarative, logical statements is intrinsically rather hard and requires a skill in abstract thinking which is uncommon. Whether the fault is in the education system or in our genes, programmers prefer to write code, because they can test it to find out whether it is correct; and analysts prefer to remain vague, because they know that the programmers will have to sort it out. The only way to get end users to be precise is to provide tools that make it very simple and obvious to do so.

On the other hand, the designers of UML itself have a strict obligation to ensure that the structure and semantics are consistent and well defined as UML evolves. Without this, the language will fail to provide the promised interchangeability of designs, UML skills will be a matter of local interpretation, and the evident advantages of standardization will be only very partially achieved. Furthermore, the ability to construct tools that can help modelers be more precise will be fatally compromised.

It's critically important to realize that UML is a family of languages, each with its own semantics. For example, one can easily envision two versions of UML identical except for some detailed specifics about the amount of concurrency that is permitted within statecharts. It would be quite wrong, in my view, to decide which of these specific semantics applies at the level of the kernel UML specification. Instead there needs to be a set of mechanisms which allow these semantics to be applied in the precise definition of standardized UML family members. It follows from this that the kernel UML should have rather sparse semantics, consisting of simple definitions of well-formedness and simple axioms about dynamic semantics. The profile mechanism must be composable and provide rich and robust mechanisms for extending and refining the structural and dynamic semantics.

Assuming that UML becomes well-defined in this way, the main added value for the next generation of tools would be model interchangeability and the ability to recognize and interpret profiles. Building on this, tools could be constructed to assist the end user to produce consistent and expressive models, including tools for automated consistency checking.

Biography

Steve Cook is an IBM Distinguished Engineer. He is currently Chief Architect of IBM's Object Technology Practice. He was the lead author of IBM's submission

(with ObjecTime Limited) to the OMG's Object-Oriented Analysis and Design Facility standard, which introduced several elements including OCL (Object Constraint Language) into the UML definition. He was the co-author (with John Daniels) of the Syntropy method, a formalized variant of OMT which was the main precursor to OCL. He has been working with object-oriented methods for 20 years.

3 Stephen Mellor

A View of the Present

Assumptions: I assume that a precisely defined, verifiable, executable, and translatable UML is a Good Thing and leave it to others to make that case.

Where We Are. In the summer of 1999, the UML has definitions for the semantics of its components. These definitions address the static structure of UML, but they do not define an execution semantics. They also address (none too precisely) the meaning of each component, but there are "semantic variation points" which allow a component to have several different meanings. Multiple views are defined, but there is no definition of how the views fit together to form a complete model. When alternate views conflict, there is no definition of how to resolve them. There are no defined semantics for actions.

Work In Progress. In November 1998, the Object Management Group (OMG) made a request for proposal (RFP) for a definition of the semantics of actions. The schedule calls for submissions in the autumn of 1999 and an accepted standard by 2000. The RFP encourages proposals to address the question of how the semantics of actions fit in with the (undefined) execution semantics of the remainder of UML.

What Are The Issues? Even with defined semantics for actions and a model of execution, model interchange—the goal, surely, of the standard—is still not possible because views conflict and they do not fit together to form a complete model. Further, the plethora of semantic variation points make semantic model interchange impractical, even if it is possible to import and export diagrams.

The bottom line is that models today are expenses. They need to be assets.

A Path to the Future

Primitives and Composition. A precise UML should be constructed from as small a set as possible of primitive, well-defined, building blocks. This approach will lead to a smaller and simpler UML with no loss of power. The rationale for this assertion is that a compositional approach permits higher-order constructs to be built from smaller, more primitive, pieces. Consequently, if only one component can be defined in terms of primitives, the UML has a smaller, simpler, definition than at present.

This still leaves open the issue of whether some constructs should be removed from the UML to make it easier to learn, use, and communicate with. This argument certainly deserves to be made, but not here.

Multiple Views. To determine what requires formalization, the UML must distinguish clearly between essential, derived, auxiliary, and deployment views. An essential view models precisely and completely some portion of the behavior of a subject matter, while a derived view shows some projection of an essential view. For example, if the state chart is an essential view, then a collaboration diagram that shows communications between state charts, but not the details, is a projection. On the other hand, were a collaboration diagram defined to be essential, then all the communications on a state chart would perforce be derived. Because the state chart would then comprise both essential and derived elements, the former view, in which the state chart is essential, is to be preferred.

The distinction between essential and derived views does not establish a sequence for the construction of the various views. A developer may begin with a collaboration diagram, using it to come up with the state charts, then revise—preferably automatically—the collaboration diagrams to conform to the essential view.

An auxiliary view, such as a use case diagram, supports the essential and derived views. An auxiliary view models informal aspects of the system and does not require formalization.

A deployment view models the packaging (or repackaging) of existing components. The existing deployment diagrams certainly model deployment views. One can also make the argument that any model which shows processors or tasks also shows a deployment view. Taking this one more step, if a writer intends a class diagram to be interpreted as a decision about which classes should exist, then a class diagram also represents a deployment view. This writer intends a 'class' to be a logical grouping of the definition of data and behavior, but not necessarily the structure for the implementation.

In summary, formalization of the execution semantics is required for the essential views only. The derived models are, well, derived, and auxiliary views don't require formalization. We take deployment again later.

A Coherent Set of Essential Views. As an example, consider the following coherent set of essential views. First, a class diagram has essential compartments for the class name and attributes only. The operation compartment shows derived operations only. Second, exactly one state chart describes the behavior for each class. No other (essential) state charts exist. The primitives for the state chart comprise only transitions, events, and the data that accompany them, (non-hierarchical) states and an action set associated with each state (i.e. a Moore state model). Finally, each action set comprises synchronous actions of only four types: data access, transformations of input data into output data, transformations of input data into mutually exclusive control outputs, and asynchronous signal generators. All four types may operate on single values or collections. In

this formulation, data access from one class to another is a derived operation on the destination class, which could be shown-preferably automatically-on the class diagram.

The execution model defines run-to-completion semantics for the action set; preserves order of signal events only between sender and receiver object instance pairs; requires the developer foreswear data access conflict (or be oblivious to it), and treats all events as having the same priority.

Pace state chart fans. We can now define the elements of the UML state chart in terms of the set of primitives that comprise the basic execution model. We may formulate concurrent states in terms of two state charts that act on two sets of instances that mirror each other; action sets on transitions (a Mealy machine) as action sets on destination sets, adding states if actions conflict; deferred events as an additional (hidden) class to queue the deferred event and a state model for the associated deferred behavior, and so on.

Should this prove impossible in any particular case, we require the invention of a new semantic unit, which may not appear on any diagram. Alternatively, perhaps we have a contradiction. Good luck finding that contradiction in today's 808-page specification!

Semantic Variation. A semantic variation point is a Bad Thing. For one, it reduces the interchangeability and understandability of UML models because the same element has multiple meanings. To the extent that semantic variation points interact, so the complexity of the UML as a whole increases dramatically.

The perspicacious reader will have observed that the execution model above de-emphasizes operations. This perspective conflicts with a synchronous, transaction-oriented, perspective assumed, inter alia, by the Meta-Object Facility (MOF). One could define a completely separate execution model that emphasizes operations and transactions to the detriment of states and events. Some components may have the same definition in both execution models. This, however, is a rather different matter than the indiscriminate use of semantic variations.

The perspective above constitutes a coherent set of essential views defined by composition. There are no "semantic variation points" One can make the argument that this perspective is too sparse to be useful. What about, say, actions on transitions, priorities on events, and arbitrary methods defined for classes?

Deployment. The UML Summary describes the UML as "a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system". As such, the UML incorporates many elements intended for implementation and deployment. For example, consider a deadline-driven real-time system that controls a manufacturing plant, say. At first blush, this system requires priorities on events and tasks. However, the underlying behavior can be stated without priorities. Events merely synchronize the actions of various state models. And the very concept of a task is a deployment issue. A core UML, therefore, is a very sparse beast-all the better to define its semantics.

How then may we define the organization of the software target? The target may comprise tasks, processors, classes of certain kinds, operations on those classes that may do more than access data. We could redraw the application model to show this deployment, and the various deployment diagrams would necessarily use the elements as currently defined in the UML. While this set of elements is extremely large, it is nonetheless limited.

As an alternative, we may define the elements of the target using the core UML. The resulting model is a model of the components that comprise the target. Hence, we may define classes Periodic Task, Event-Driven Task, Function, Parameter, Application Class and so on. The target model is just another UML model. It models the organization of the software components and it is independent of the application. In fact, it is a model of an application-independent software architecture.

The object instances of this target model can be primitive, so the decision to have three periodic tasks and seven event-driven will manifest itself as three instances of the class Periodic Task and seven instances of the class Event-Driven Task. Other object instances in the target are elements—even down to the individual actions—copied from the original application model. The populated target model has exactly the same semantics as the original application model. It's just organized differently.

It is then a simple matter to spit out the content of this model in code. With supporting tools, the target model—the application-independent software architecture—acts as a model compiler for the application.

A Vision for the Future

With the components identified above in hand, the nature of methods and tools change significantly. First, let's summarize what we (can soon) have:

- a small set of primitives that can be composed
- a clear distinction between essential and other views of a model
- a coherent (small) set of components that comprise the essential view (with syntactic sugar)
- an execution model for the components that comprise the essential view without semantic variations

There are three main features to consider: separation of subject matter, execution, and translation.

Separation of Subject Matter. The separation of the application model from the application-independent software architecture makes each model an asset. There may be multiple implementations of that same application model, each of which has different performance properties depending on the pattern of usage for the system. An application model can be redeployed using a different model compiler, so the application model does not change repeatedly as the business

undergoes technology change. Similarly, the model compilers are themselves assets that can be reused across multiple applications that share the pattern of usage.

The model compiler is general. Any application that has the same pattern of usage for the system can make use of that same model to produce code. Consequently, this target model can be sold to a broad variety of customers. There is a need for only a relatively limited number of these target models.

The models-both the application models and the target models-are now assets. The application model can be re-deployed by acquiring a new target model. And vice-versa.

Execution, Simulation, and Verification. Because the essential view has a well-defined execution model, we can build a model of a problem and simulate it. This will shorten the analysis cycle because a model can be executed without making detailed decisions about the design of the system, only the content of the underlying problem.

Translation. In reality, systems comprise multiple layers. Each layer can be constructed as a UML model. The object instances derive from the layers above. The bottom-most layer is the one that, when all is said and done, the entire system resides. We may then generate code from that final layer by serializing the model into text that can be compiled.

Methods and Tools: What Do We Need?

The fundamental issue in model building, and therefore the methods required to build such models, is abstraction. How does the developer come to a set of abstractions that properly captures the domain?

This task is eased by the simplification of the UML to contain only essential elements. Now we can obviate all packaging elements, there is no need to choose to package a model in a particular way for the sake of efficiency. The next step, then, is to define a “normal form” for model structure. This normal form is implicit in the coherent set of models defined above. The abstractions must represent the invariant in a problem domain, and all variants must be expressed in data. Methods will focus on how to find those invariants. Believe me, it won't be use cases.

As for tools, the fundamental issue is the logical structure of the repository.

- First, model builder tools will act as front ends for that repository. Separate files per model will not work. Goodbye, Rational Rose
- Second, tools that execute the model in the repository are required. Specifically, there is a need for offline model verification tools, online interpretive simulation tools, and online animation tools. The last is frankly not very useful, but it demonstrates well. The other tools will be used for regression testing and model debugging respectively.

- Third, we need a wide variety of different model compilers, each targeting different architectures.
- Fourth, a set of bridging tools that populate one model from the content of another is needed. The issue here is generality so that the bridging tools can populate any arbitrary model.
- Fifth, a standard “language” for turning a populated model into code is required.

This set of tools enables a market for complete models of domains, expressed in UML, that can be deployed onto other models, including model compilers.

Methods and Tools: What Do We Have?

We have:

- a method
- repository-based tools
- verification and simulation tools
- the beginnings of a market in model compilers
- little, so far, in the way of general purpose bridging tools
- languages (but not a standard) for generating code
- the stirrings of a market for domain models

All we need now is to make the market aware that all this is possible, build tools around the standards defined by the core, executable UML, and make it so, Mr. Crusher.

3.1 Biography

Stephen J. Mellor is best known in the real-time community for his contribution to the development of Object-Oriented Analysis, Recursive Design, and the Shlaer-Mellor Method. He is Vice President of Project Technology, Inc., where consults, teaches, and researches applications of the method. Mr. Mellor is currently working with the Object Management Group (OMG) to define semantics for actions. Mr. Mellor is also a member of the editorial board for IEEE Software.

4 Jos Warmer

With the emergence of the UML as a standard for modeling new opportunities arrive and they will change the future of tools. UML has several characteristics that allow for this development. UML includes an explicit meta-model and an explicit interchange format: XMI. Any tool that supports UML will be able to read and write models using XMI. If the tools implement this correctly, UML models can readily be interchanged between different tools from different vendors. When this becomes reality, the tool market can change fundamentally.

How tools cooperate today

Until today, object oriented case tools from different vendors are mostly incompatible. Once you start using a specific tool to develop your models, you are locked into this tool. Changing to another case tool is a very costly decision, because almost all information needs to be entered into the new case tool by hand. Also, the choice of add-on tools is restricted to those available for the chosen case tool. For vendors that develop add-ons or plug-ins for case tools, this situation is bad. Let us assume that we are a company developing a code generator for e.g. the Java language. We need to decide beforehand to which case tool this will become an add-on. Let us assume that we choose case tool C1 to be our target. The market potential of our code generator is now limited to the market share of the targeted case tool C1. Being good software engineers, we design a good architecture of the code generator: we develop it as two components. One component will interface to the case tool C1; the other component generates the actual code. When we want to target a second case tool C2, only the interface component needs to be rewritten. At first sight this looks like a solution, but the task is more complex than expected. To start with, we need to use different methods to extract information from C2. The real surprise follows shortly: the structure and content of the information we get from C2 is completely different from what we got from C1. Both C1 and C2 use a different meta-model (sometimes explicit, most often implicit) and there is no well-defined way to transform the meta-model of C2 into that of C1. Our code generator, however, was based on the information we got from C1, and therefore based on the meta-model of C1. In the end we need to rewrite a large part of our code generation component specifically for C2.

How tools cooperate tomorrow

With UML the scenario above will change drastically. First of all, the XMI interchange format allows us to write the interchange component of our code generator once. Because all UML tools export the UML model in XMI format, our new interchange component can read the output of all case tools. Secondly, the structure and contents of the information that we get will be identical for each case tool. It is based on the UML meta-model. The fundamental change that we see is that we do not need to make a choice to support one or more specific case tools. The only thing we decide is to develop a code generator for UML. We will be able to combine our code generator with any existing UML tool. Even new tools, which didn't exist at the time we started, will be able to be combined with our code generator. Consequences for tools For tool vendors UML will open up a much broader market, because they can target their tools to the whole UML market without restrictions. This will generate stronger competition for widely used tools. A specific group of more specialized tools, like advanced model validators or proofing systems, are barely available on the marketplace today, because their market share is too small. With UML, these tools will now have a wider market, and they might become economically feasible.

Limitations of tool cooperation

Although UML will enable much more tool support, there are several areas where UML won't help and the old situation won't change much. The UML meta-model and interchange format only defines the contents of a UML model, not the visual diagrams that a modeler draws. The visual information cannot be interchanged yet. This means that the choice of the visual modeling tool will still result in a vendor lock-in. This limitation might even affect the possibility to combine different non-visual tools. It is therefore important that the OMG will start work on a standard for interchanging visual UML information.

New types of tools

UML defines an explicit meta-model and includes a complete description of the semantics of all modeling constructs. Having this semantic description of the meaning of all constructs, might allow tools do a more intelligent job. For example, until now information found in sequence diagrams (messages sent and received) and state charts (events being accepted, states being defined) has been used mostly as analysis and design documentation, but it doesn't necessarily have a well-defined and checkable relationship with the actual code. Using the semantics defined in the UML tools might be able to validate the actual code against the sequence diagrams and state charts. It remains unclear for now whether the UML semantics is defined rigorously enough to allow for these kinds of validations. If not, the UML semantics specification should be enhanced and made more precise.

Conclusion

Although the subject is about tools for UML, it is important to realize that it is not UML alone that makes a difference. Without the XMI interchange standard none of this would ever be possible. The inclusion of XMI is the enabling factor for this process. It is therefore understandable that none of the above has taken place with the UML 1.1 and 1.2, because XMI was still lacking. With UML 1.3, the industry can finally start the real work on generic UML tools.

Concluding, I expect that we will see a much broader variety of specialized tools that use the UML meta-model and XMI as their basis. More tools will become widely available. It will probably take several years before we will see a real change, because most vendors are still struggling to get their tools up to date with the new UML 1.3 and XMI standard. Also a new type of tools might emerge, although this will certainly take much longer.

Bibliography

Jos Warmer is working as a trainer and consultant object technology with Klasse Objecten. Before starting to work with Klasse Objecten Jos worked within IBM's Object Technology Practice. During the development of the UML 1.1 until now

he has been IBM's representative as a member of the UML core team. He was responsible for the development of the Object Constraint Language (OCL). He is still a member of the UML Revision Task Force, which prepares new versions of the UML, under the responsibility of the OMG. Jos Warmer is one of the authors of 'The Object Constraint Language: Precise Modeling with UML' and of Dutch books on OMT and UML. You can reach Jos by email: J.Warmer@klasse.nl.

5 Alan Wills

Tools support methods. Here is a selection of methods that my clients find very useful. To make them work, we have to assume certain clear relationships between the different parts of the UML notation, and between the UML and the program code. Currently, no such relationships are generally agreed for UML; and the techniques are under-supported by tools.

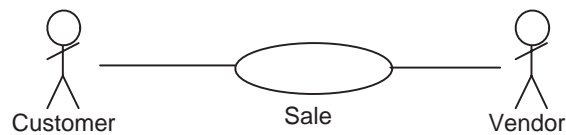
I'll explain each technique, show how precise UML semantics are necessary, and show what a supporting tool could do. The ideas are part of Catalysis, developed by Desmond D'Souza and me.

Business modeling with consistency and completeness checks

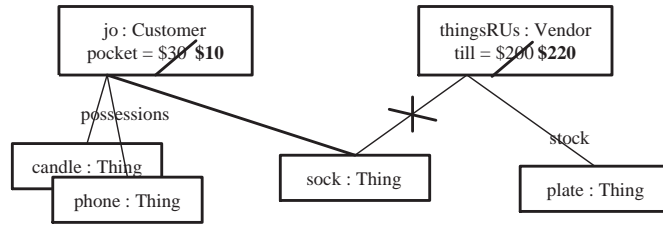
We start most projects from a business domain model, separate from any particular piece of software, defining the vocabulary common to all components. Scenarios, stories about typical sequences and flows in the business, form a good start; but we need to identify exactly what types of 'actions' (use-cases, tasks, interactions, operations) and objects are involved: they will ultimately be reflected in the software.

We go round this cycle, asking questions of the domain experts & source documents. You can enter at any step:

- Method – Identify actions by asking “what happens?”, looking for verbs, etc. Document each as an action: draw it and its participants using the UML use-case ellipse:



Write a postcondition (and precondition) defining its effects on the participants. (We prefer to start with pre/post instead of giving a use-case sequence, because there may be several different ways in which the same effect could be achieved: we want to say what's common to all of them, deferring detail.)



The postconditions are expressed both informally and in terms of changes in the associations and attributes of the participants. We can illustrate this with a before/after instance diagram ('snapshot') – the bolder lines represent 'after'. The postcondition can be written:

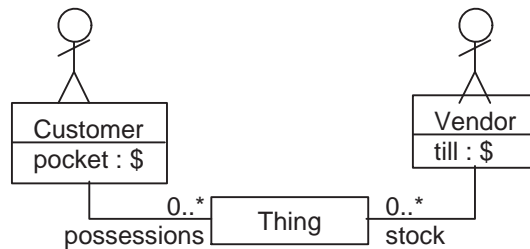
```

action sale (cust: Customer, vendor: Vendor, thing: Thing,
price:Price)
  
```

```

post      cust.possessions == cust.possessions@pre + thing
          and vendor.stock == vendor.stock@pre - thing
- - thing is transferred from vendor's stock to customer's
possessions
          and cust.pocket == cust.pocket@pre - price
          and vendor.till == vendor.till + price
- - price is transferred from customer's pocket to vendor's till
  
```

This gives us a type diagram of the vocabulary we've used in the action spec:



So by being precise about the specification of the action (use-case) we've made a clear relationship between the static or logical part of the model, and the dynamic part. As far as our modeling cycle goes, we have a way of discovering new objects, attributes and associations: they come up when you try to describe the postcondition of an action.

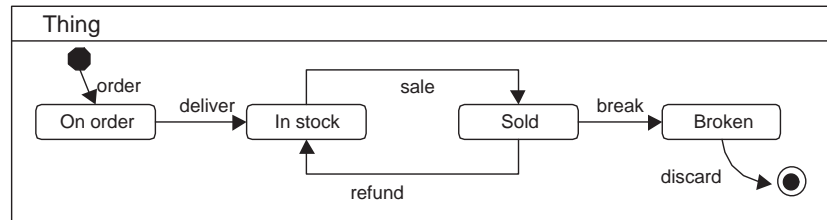
More informally, we can ask about any action, 'who or what is affected, or affects the outcome?'

Precision – Notice that we can't really do this unless we assign a semantics to UML in which the ellipses we draw in a dynamic diagram can be described in

terms of the changes of state in a static diagram. In my work, we assume this; but it is not part of the UML standard, and there are no tools that support it.

Tools – A good support tool would draw before/after snapshots; and would check the more formal part of the postcondition against the type diagram. ?

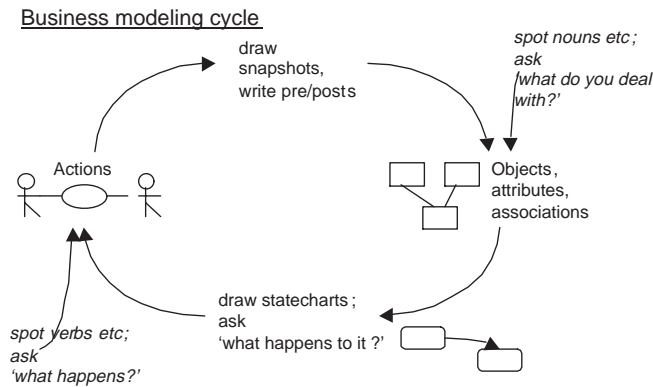
Method – Identify types of object – either as a result of action specification, or by spotting nouns etc informally. For each one, ask “what changes can it undergo?” One way is to draw a statechart for each type:



Notice that ‘sale’ is one of the actions we can see here. In the Catalysis interpretation, the transitions on statecharts are actions: this gives us a useful way to discover actions: we can now write postconditions of ‘refund’, ‘deliver’, etc, and possibly discover new types of object, associations and attributes in the process of describing them.

Precision – This technique relies on a clear relationship between the transitions on statecharts and the actions in the use-case model. We also use the idea that states are boolean attributes, usually derived from other attributes or associations: for example, ‘sold’ means that the Thing is part of the ‘possessions’ of a Customer.

Tools – A tool should be able to generate the actions and attributes, and highlight those that still need to be defined.



We continue around this cycle, finding actions and objects, until there are no new ones, and we don’t have to modify any definitions. We stick to a chosen

level of abstraction, avoiding going into very detailed actions – the beauty of a postcondition is that it can state the relationship between the states before and after a transaction that might take a long time and involve all kinds of smaller tasks that we don't need to concern ourselves with initially.

Tools – A good support tool for domain modeling should help us to create new actions and associations, and highlight those that need further definition.

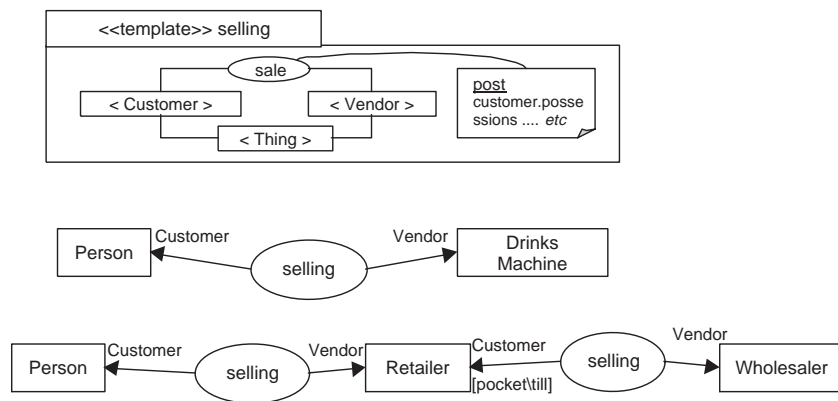
Imprecision

Although I've described the modeling cycle in terms of precisely-defined postconditions etc, it is also possible apply the technique less formally. Postconditions written in informal language still use a vocabulary of terms, about the relationships between things: those words can be spotted and should turn up as associations and attributes.

Tools – an informal noun-spotter and verb-spotter would help to take us round the modeling cycle.

Model construction from views and frameworks

The same modeling patterns tend to crop up repeatedly. This is often about the relationship between objects; not just the static relationships, but the dynamic protocols describing how they interact. The Observer pattern is a good example. To use the pattern, you don't usually create a class called Observer and another called Subject: you take some existing classes of your own and add in the associations and operations required by the pattern. Nor do you necessarily copy code from the patterns book: what's important is not how each operation is coded, but what it achieves – its postcondition. What we need is a way of defining 'macros' that can be combined with other chunks of model in a well defined way. So for example, if our Customer- Vendor definitions are a framework (in its own package), we could apply them to specific types:



The application of the template establishes a relationship by adding the appropriate attributes and associations and actions to the target type definitions. A tool should be able to ‘unfold’ the resulting model on demand, but preferably normally show the model as drawn.

Precision – UML lacks a semantics of model template composition. There is one described in the Catalysis book ([1] chapter 9). Particular issues include how to combine postconditions – not the same as subtyping.

Tools – While some of the tools on the market have a way of applying a pattern, it’s a one-off operation: you lose any trace of the original intention of the designer. A tool should preserve and present the original framework application, showing the ‘unfolded’ resulting model only if the designer asks for it.

The same mechanism is useful when we have several views of a domain or a set of requirements: different interested parties have their own ideas about what it should do, and each applies their own set of constraints on the final design. Combining views together is therefore a process of adding constraints.

Refinement and Conformance

Object and component design are all about encapsulation. We therefore like to separate the specification of an interface from the definition of the class that implements it – Java, IDL, and Eiffel provide explicitly for this. Then there may be many implementors of an interface.

Interface definitions as compilers understand them aren’t sufficient for us as designers: a list of operation signatures doesn’t say anything about what the operations are supposed to achieve. We can use postconditions to define what’s expected of an operation; if we don’t like formal language, we can write them in English or whatever, or draw illustrative snapshots as above.

The postconditions use a vocabulary of attributes and associations, which are not all directly visible in the implementation. For example, if I want to say of a Queue, that the put operation increases the length (more formally: put post: length==length@pre + 1 and .etc), then I have to have an attribute ‘length’. This makes sense, because everyone knows that every queue has a length:

Queue <<interface>>
length: int <i>etc</i>
put (x: Thing) <u>post</u> length=length@pre + 1 and get () : Thing <u>post</u> length==length@pre -- 1 and

Notice that I’m not providing any operations that read the length: clients just have to remember for themselves how much they’ve put in, or look out for exceptions.

Now suppose someone comes to me with a class that implements Queue as a linked list. I want to check to see if their put increases the length as specified. The first obstacle is that they don't have a variable called 'length': it's just a chain of nodes. But they can write a function that 'retrieves' my length by counting up the nodes. Then we can test to see what happens to the length. Such a function is just for the purposes of verification against my spec, and it doesn't need to perform well, and might be omitted in production versions.

Precision – UML doesn't currently have a clear notion of what an attribute means. In this case, we take it to mean something that doesn't have to appear in the code, but can be written as a read-only function for QA purposes. (More in [1], chapter 213.)

The retrievals are part of the refinement relation – again, this is not clear in UML as it stands.

Tools – Refinement is about traceability. Any change to a postcondition mentioning 'length' affects the parts of an implementation that implement it – which are traceable through being mentioned in the retrieve function. I want tools to help me find out how changes propagate through my design, and refinements are part of that.

Testing

In case postconditions seem a little too abstract for everyday use, let me point out that they are equivalent to test harnesses: if your design meets the postcondition, you have done it properly. The advantages of pre/postconditions and invariants are: that they can be much more succinct than the program code; and they are general across all the different implementations. The programming language Eiffel includes executable invariants, pre and postconditions used for testing. All modern RAD approaches advocate writing the test material before completing the design.

Pre/postconditions are generally written in OCL within a UML model; but they can just as well be written in Java. The benefit of OCL is that it is more succinct: for example, you can write conditions across all members of a set, instead of having to code a loop.

Precision – we want a clear definition of what program code does and does not conform to a given UML specification.

Model refinement and systems integration

The model written for a high-level specification is usually simpler than the class structure in the program code: the former is intended to explain the externally-visible behaviour of a system or component; the latter is supposed to make it efficient, flexible, and built from reusable parts. While we would like to keep the two models similar for traceability, we also need to be able to map from one to the other.

One reason for the model/implementation difference is that the system has been constructed from disparate parts. While our sales business has quite a simple model, the sales support system may be made up of an accounts subsystem, a deliveries system, and so on, each of which has its own partial model.

Precision – UML needs a clear definition of what it means for a model to be implemented by several partial ones, especially where they overlap. (See [1] ch.10.11).

Biography

Alan Cameron Wills is a consultant in object and component based development, working with a wide variety of clients in Europe and America since 1990, in fields as diverse as finance, telecoms, and manufacturing control. He gained a PhD in the application of formal methods to object oriented programming in 1991. With Desmond D'Souza, he is author of the Catalysis method, detailed in the book “Objects, Components and Frameworks in UML” (Addison-Wesley, 1999). Alan is Principal Consultant with TriReme International Ltd, based in the UK. Contact details: alan@trireme.com. Tel: +44 161 225 3240. <http://www.trireme.com/>

References

1. Desmond D'Souza and Alan Cameron Wills. Objects, Components and Frameworks in UML: the Catalysis approach , Addison Wesley, 1998.