Why there are difficulties in using UML

Jim Cooling - Lindentree Associates

The main issues being:

- The size and complexity of the standard.
- Problems in understanding the standard.
- Impreciseness of the semantics.
- Problems in choosing and using diagrams.
- Confusion between notation and process.
- Inadequate modelling of run-time software.

These are not trivial aspects but can cause developers real grief and angst (it might be a good idea to include a 'health warning' with UML documents, figure 3.4!).

Figure 3.4 Software health warning for the unwary

(a) Size and complexity of the standard.

As a real-time developer, your source of information will be the Superstructure and MARTE documents. Unfortunately, the size of the standard beggars belief. Version 2.4 of the Superstructure document, for example, is 748 pages long; version 1.0 of MARTE occupies 738 pages. In all a total of 1486 pages. This is an enormous inflation of the initial specification which, as noted earlier, took up 171 pages.

Just how has this come about? Does it help rather than hinder? Can the bloat really be justified? The answers, in reverse order, are 'no', 'hinder' and 'indiscipline'. Experience has shown that good standards are produced by small groups of highly experienced people. The group also needs to concentrate on the core aims of the standard; everything must contribute to these aims. Anything of the 'nice to have' variety must be ruthlessly opposed (unfortunately we all have our own hobby-horses). And something really important; standards shouldn't be *driven* by commercial concerns.

In my view, the history of UML shows that it fails to meet these criteria. In effect it went from something well-focussed to a 'do-all' thing. This can be seen in the Conformance section of the Superstructure specification:

UML is a language with a very broad scope that covers a large and diverse set of application domains. Not all of its modeling capabilities are necessarily useful in all domains or applications. This suggests that the language should be structured modularly, with the ability to select only those parts of the language that are of direct interest.

(b) Problems in understanding the standard.

This may be an unkind remark, but it seems that many of the contributing authors had only a passing knowledge of English language and grammar (much more unkindly, it has been referred to on Wikipedia as 'geekspeak'). The resulting document was in general tortuous to read, difficult to understand and poorly constructed. Many semantic aspects are common to various topics, but these were scattered through the document. This made it really quite difficult to tie things together. When the UML 2.0 specification was first released it took us the best part of two months to 'deconstruct' the document (that is, to work out what it really meant). In many cases we just couldn't establish the rationale behind the specifications. Thus it was doubly difficult to understand exactly what the writers were getting at. The poor (almost opaque) writing style resulted in numerous examples of ambiguity and ambivalence. There were examples of contradiction and circular definitions that, because of the sheer size of the work, were difficult to spot.

(c) Impreciseness of the semantics.

Consider the following scenario. We've gathered together a group of developers who claim to have a good working knowledge of UML. We present them with a set of UML-based diagrams and ask them to explain what these diagrams are all about. Let us also assume that the diagrams conform to the UML standard, i.e. legal UML. Well, if you believe some of the literature concerning UML, we would expect that they would:

- Recognize all the symbols used in the diagrams, that is, know their syntax.
- Know the semantics (the meaning) of these symbols.
- Understand, from the diagram, the nature of the relationships between the various constructs.
- Agree on the general structures of the code models produced from these design models.

Regrettably this may not be the outcome of the test; there could well be confusion and disagreement. But, you ask, if UML is so precise, why the problem? The reasons are twofold. First, certain parts of the standard are *not* precise but can be interpreted in different ways. Second, and more important, UML deliberately gives implementers ways to 'customize' aspects of the specifications. These are called 'semantic variation points'. According to the Superstructure specification they *explicitly identifies (sic) the areas where the semantics are intentionally under specified to provide leeway for domain-specific refinements of the general UML semantics*. Two mechanisms support such refinements, stereotypes and profiles. In other words you can adapt UML to produce specialized diagrams having their own particular syntax and semantics. Of course other developers can do exactly the same. So we end up with sets of diagrams that are specialized, different but, in UML terms, perfectly legal. Now this is not a trivial issue. It runs right through UML, the examples below coming from the Superstructure specification:

- *Precise semantics of shared aggregation varies by application area and modeler.*
- *The order and way in which part instances in a composite are created is not defined.*
- *In accord with this semantic variation point, inheritance of values for static features is permitted but not required by UML.*
- *It is a semantic variation whether one or more behaviors are triggered when an event satisfies multiple outstanding triggers.*

You should now see why our experienced developers may have problems.

(d) Problems in choosing and using diagrams.
Over time the number of diagrams specified by the UML standard has increased. And in many cases there has been a corresponding increase in diagram complexity. The root cause of this is the desire, as noted earlier, to produce the 'do-all' specification:

The modeling concepts of UML are grouped into language units. A language unit consists of a collection of tightly-coupled modeling concepts that provide users with the power to represent aspects of the system under study according to a particular paradigm or formalism. For example, the State Machines language unit enables modelers to specify discrete event-driven behavior using a variant of the well-known statecharts formalism, while the Activities language unit provides for modeling behavior based on a workflow-like paradigm. From the user's perspective, this partitioning of UML means that they need only be concerned with those parts of the language that they consider necessary for their models

In effect it's like having a big bag of tools to cover any and all events.

As a result, not only do we have numerous diagrams, we also have overlap and redundancy of diagrams and constructs. This makes it really quite difficult for those new to UML to make sense of things (and the specification is less than helpful). CASE tools usually provide a subset of the full set of UML diagrams, which simplifies the choices for the developer. However, many users of CASE tools don't truly understand the meaning of the diagrams and their symbols.

(e) Confusion between notation and process.
UML started life as the 'Unified Method', with the idea that it would cover both a development process *and* a related set of diagrams (although the first document limited itself to the diagramming aspects only). OMG decided to restrict UML to being a diagram modelling technique only; the process aspect was dropped. However, Rational pursued the process idea, developing the Rational Unified Process, RUP. Their CASE tool, Rational ROSE, incorporated the RUP and the UML diagram set. Other vendors went down the same path, so when you got their UML tool you also got their process. It's not surprising then that many software developers mistakenly assume that UML specifies a development process. Newcomers usually turn to the specification for process guidance and end up being very confused indeed. Moreover, advice and information from tool vendors can further confuse the situation owing to differences in the various processes.

(f) Inadequate modelling of run-time software.
UML places great emphasis on the structural modelling aspects of software; relatively little attention is paid to run-time aspects. The profile for MARTE is intended to remedy this, as defined in by its scope:

This specification of a UML™ profile adds capabilities to UML for model-driven development of Real Time and Embedded Systems (RTES). This extension, called the UML profile for MARTE (in short MARTE for Modeling and Analysis of Real-Time and Embedded systems), provides support for specification, design, and verification/validation stages.

A review of the profile shows that features are provided to model the:

- Temporal features of real-time systems
- Hardware and software resources and

- Allocation of functional aspects onto the computing platform.

Unfortunately it doesn't explain these things particularly clearly. It's sheer size, verbosity and abstract slant makes for lengthy, difficult and unrewarding reading. Moreover, it seems to assume that detailed modelling of run-time aspects (the run-time architecture) is not its responsibility. And yet the run-time model is key to understanding, developing, implementing, testing and maintaining embedded systems. In UML terms this model defines the features and behaviour of the Platform Specific Model (PSM), dealing with:

- Tasks, threads and processes.
- Resource sharing and protection in concurrent designs.
- Periodic and aperiodic tasking models, including timing aspects.
- Event-driven designs.
- Interactions between concurrent units (e.g. data transfer, synchronization).
- Mainstream scheduling methods (e.g. priority pre-emptive, co-operative and deadline monotonic).
- Memory partitioning, allocation and protection.
- Isolation of resources via time (temporal) and memory usage (spatial) partitioning.
- Robustness and fault tolerance.

The situation at present is that MARTE doesn't deliver on these items. ß