UML is great for Embedded Systems – Isn't it?

Theodor Tempelmeier[1]
Fachhochschule Rosenheim

Preamble. Object-Oriented Design is definitely the right way of building systems, and this also applies to embedded systems in the view of the author. Encapsulation, abstract data types, etc., and other concepts such as generics or templates, are a must when developing complex embedded systems. This is in fact also the way the more progressive companies do already develop embedded systems.[2] The author is reluctant, though, to accept inheritance as a dominating design principle per se (see appendix). But at least an object-based approach is the right choice in the author's opinion, without any doubt. However, this contribution does not deal with object-orientation in itself, but only with the application of the Unified Modeling Language within this context.

1. Introduction

Thinking about the application of the Unified Modeling Language (UML) to embedded systems development actually involves questions on different issues:

1. Can something be modelled in UML in principle?
2. Can something be modelled in base UML (i.e. UML without extensions)?
3. Can something be modelled in UML in a "better" way, seen from a pragmatic point of view, i.e. concerning tool availability, etc.?
4. Can something be modelled in UML in a better way with respect to the concepts in UML?

The answer to the first three questions is probably a plain "yes". If something can be modelled with certain concepts (say in the framework of ROOM) then it can be modelled in UML by defining extensions (stereotypes) which exactly exhibit the same behaviour as the original concepts. UML just serves as an implementation vehicle, in this case. Concerning the second question, one may well assume that anything can be modelled with UML, given the plethora (and vagueness) of concepts in UML. And, of course, availability of tools will profit from the unification process. This contribution only deals with the fourth question, both from a practical and theoretic point of view.

2. Software Requirements Specification

From an actual project in the aerospace industry, three sample issues are used to evaluate the Unified Modeling Language for the requirements specification phase (see appendix). The conclusion from this is as follows.

Appropriate models for specification have been used in the engineering domain for quite some time (perhaps in contrast to the domain of business and administrative applications). It can not easily be seen how the Unified Modeling Language would provide any fundamental improvement as compared to the current modelling approaches.

---

[1] Prof. Dr. Theodor Tempelmeier, Fachbereich Informatik, Laboratory of Real-Time Systems, Fachhochschule, Marienberger Str. 26, 83024 Rosenheim, Germany, phone +49-8031-805-510, fax -105, e-mail tt@extern.lrz-muenchen.de, world wide web http://www.fh-rosenheim.de.

[2] By the way, the call for papers for this particular workshop seems to be wrong in claiming that "software for embedded systems is mostly *designed* according to the concepts of Structured Analysis." It is true that Structured Analysis is very often used for describing and analysing(!) the software requirements, but using Structured Analysis for *design* is rather rare (only in Ellis: Objectifying Real-Time Systems, for instance).

3. Software Design

The role of a modelling language in the design phase may be manifold. The two most important roles are

- the modelling language is used as design language down to coding, i.e. the modelling language is also used for "programming"
- the modelling language is used for design and the programming language is used for coding

The problem with the first approach is that most modelling languages do not have semantics as precise as it is necessary for programming. The problem with the second approach is that there may be a semantic discrepancy between the modelling language and the programming language (the programming language is in ultimate authority).

From the experience of applying object-based designs to embedded systems and from experience with various modelling languages and tools (e.g. HOOD), the author takes the following position.

- "Programming" in the modelling language is neither practicable nor reasonable nor desirable.
- The modelling language can be and should be used for a "visualisation" of the design. *This also implies that the modelling language resembles the design concepts within the programming language on a one-to-one, or "isomorphic" basis.* (Naturally, this is only valid if the programming language includes sound design concepts, e.g. programming languages such as assembler are ruled out.)

Obviously, the Unified Modeling Language fits the concepts of mainstream languages such as C++ or Java nicely. The author could accept UML for representing designs targeted to these languages, though not all concepts of embedded systems development (outside C++ and Java) can perhaps be described adequately.

If other target languages are used, e.g. Ada for safety-critical systems, the situation is different. Ada has sound design concepts (which are sometimes superior to C++ or Java concepts) which seem to be without counterpart in UML. This gives rise to the weird situation that the language for modelling a design is less powerful (in terms of design concepts) than the programming language. The following three examples are given.

- *Protected objects.* Protected objects are not supported by UML. The keyword "protected" has a totally different meaning to the C++ or to Ada95 programmer. What would be the meaning of a keyword "protected" in UML?
- *Hierarchical libraries.* Hierarchical libraries can be realised by nesting, which causes some problems concerning recompilation, etc. Ada95 has a much smarter scheme for hierarchical libraries which avoids such problems. How are hierarchical libraries handled in UML? Even if semantics in UML were simply tied to the library model of Java, one would still face the question of how to handle the generic hierarchical library units of Ada95, as there are no generics in Java. Further, it would still be open how to distinguish nested hierarchies from "smart" hierarchies.
- *Abstract data types (in the meaning of a class without inheritance) as distinct from classes (with inheritance).* Ada95 offers both classes without and with inheritance (keyword "tagged"). There are good reasons for this distinction, for instance that one sometimes wants to avoid certain aspects of inheritance in safety-critical systems (see ISO/IEC), while there is no reason to refrain from using abstract data types. Suggestions to use the Java concept of "final" classes for the purpose of this distinction may be acceptable, but this seems not to be a concept within base UML, version 1.3alpha.

It is possible, of course, to enrich UML with special notes and comments to reflect some additional concepts. Additionally, tools may employ a variety of switches to steer code generation in the desired way. But, this is exactly a repetition of the well-known situation with earlier notations and tools: The designer has in mind an exact idea of his design (in terms of the concepts of the target language, say, Ada). He then has to understand the design notation (UML), its semantics (maybe defined in terms of another language, say, C++ or Java), and the effects of the code generation switches of the tool (for instance about 50 switches for generating C++ classes in the Rational Rose tool). And then, maybe, the designer will get the design (tailored to his target language) he had exactly in mind from the very beginning. Such a procedure his hardly acceptable in practice.

As a conclusion, a one-to-one (or "isomorphic") correspondence of design concepts in the modelling language and in the programming language is required (if the programming language itself includes a set of adequate design concepts as it is the case with Ada). This requirement does not seem to be fulfilled by the combination "Ada and UML".

Appendix:
Excerpt with modifications[3] from a presentation at the 24th IFAC/IFIP Workshop on Real-Time Programming, Schloß Dagstuhl, Germany, May 30 - June 2, 1999.

Aspects of Flight Control Software - A Software Engineering Point of View

Alfred Roßkopf
DaimlerChrysler Aerospace AG

Theodor Tempelmeier
Fachhochschule Rosenheim

...

Software Requirements Specification

Requirements for flight control law software are usually defined using control block diagrams, which essentially describe blocks and data flows between them. Each block represents a transformation of input data to output data. The detailed control law computations are described in an algorithmic language, e.g. in FORTRAN. It is essential that this specification is executable, in order to validate the control law design.

In a small case study the Unified Modeling Language UML (Version 1.0) has been investigated with respect to specifying requirements for typical flight control software. The results are presented here in the form of three examples:

- definition of control surfaces
- control law block diagrams
- definition of external interfaces

Definition of Control Surfaces.

The main outputs of a flight control system are commands to the control surface actuators. Therefore, a requirements specification for flight control software would typically include a definition of these control surfaces. A possible UML definition for this is given in figure 1. A control surface "is a" primary or a secondary surface (inheritance relation) and so on. And a delta-canard aircraft "has" one rudder, four flaperons, and so on (aggregation or composition relation).

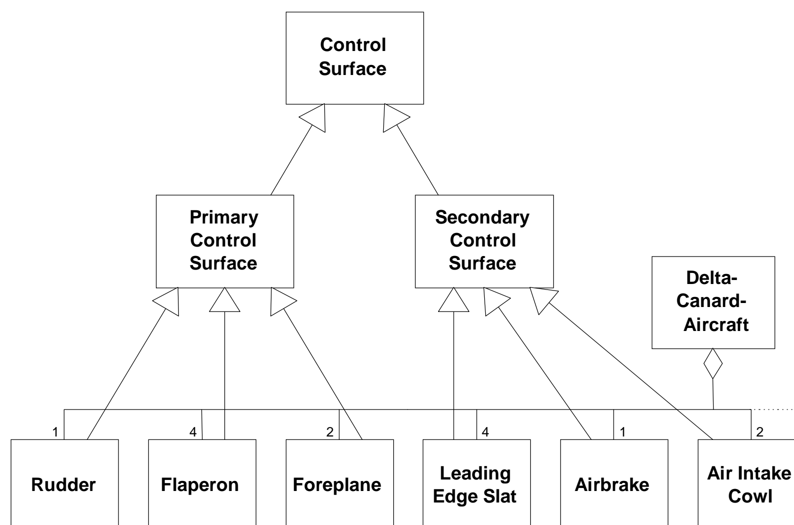

Fig. 1: Control Surfaces of an Aircraft in Delta-Canard-Configuration (UML Diagram)

[3] Modifications are shown as footnotes and are in the sole responsibility of the second author (T. Tempelmeier).

Figure 2 shows as an alternative a sketch of an aircraft with the primary control surfaces emphasised in white and the secondary ones in dark. It can be seen that a simple figure is sufficient to convey at least the same information as the UML diagram. The authors would not consider it very helpful to rephrase such parts of the requirements in UML.

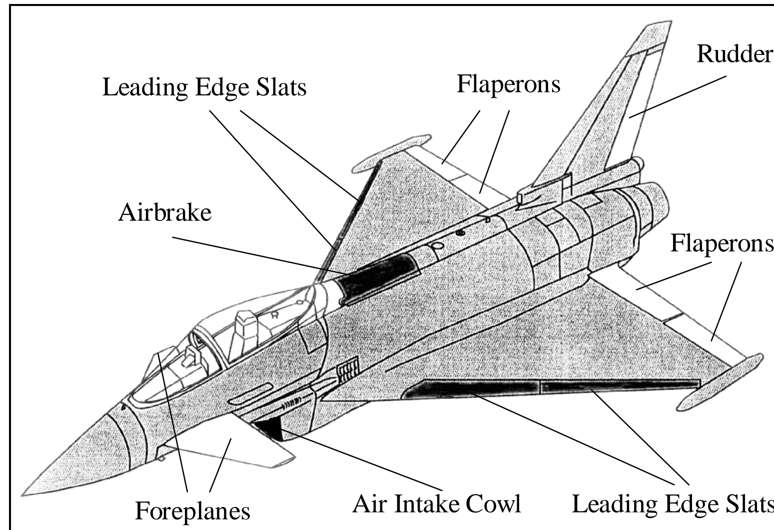

Fig. 2: Control Surfaces of an Aircraft in Delta-Canard-Configuration

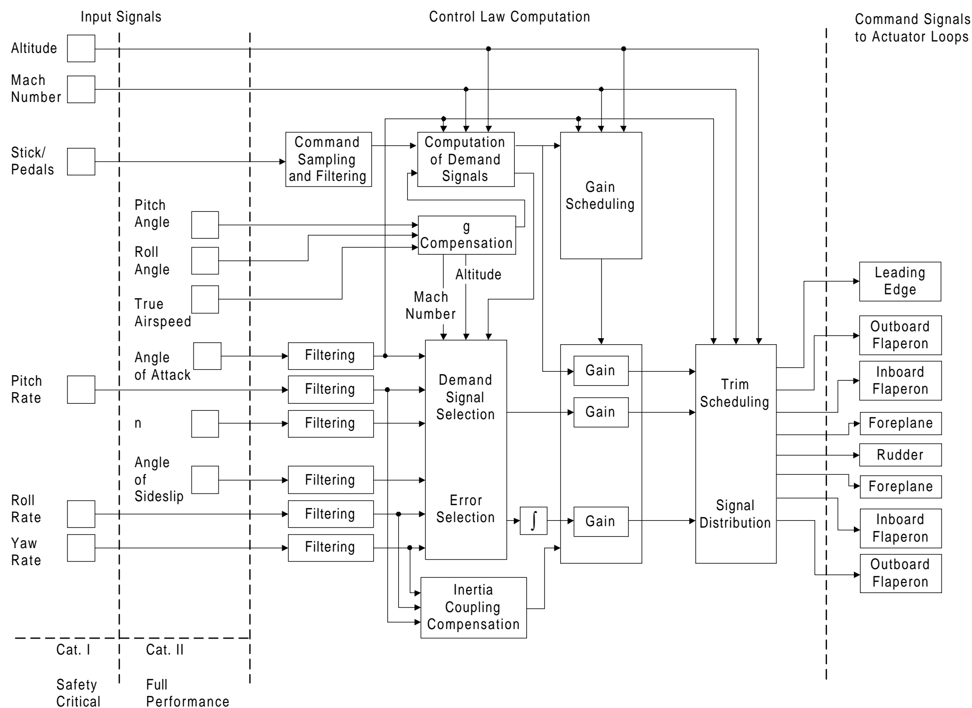

Figure 3: A block diagram of flight control laws according to (Kaul, 1992)

Control Law Block Diagrams.

As said above, control block diagrams contain fundamental parts of the requirements for control law software. Figure 3 shows an example. If one would try to rephrase such diagrams in UML, the following difficulties arose.

- Obviously, the control block diagram is not a class diagram. Instead, the blocks constitute

multiple instances of classes, e.g. the “Filtering” objects. The control block diagram thus must rather be seen as an “object diagram”. While it is possible to draw object diagrams in UML, only the association relation can meaningfully be used in this case. However, the association relation is only a line drawn between rectangles with almost no semantic information. UML’s object diagrams are thus not a suitable alternative for control block diagrams.

- Class diagrams do not help very much in this case, either. They would reveal surprisingly little information, for instance that a notch filter “is a” filter, etc. The complexity of the control block diagram lies in the functionality, i.e. in the contents of the block diagram elements and in their interdependencies. In contrast, UML seems to be more suitable for applications where the complexity lies in the class relationships, like in database applications, for instance.
- One could try to (mis)use other diagrams of UML, e.g. collaboration diagrams or activity diagrams, but the authors do not see any advantage in doing so, as compared to using well established control block diagrams.
- [4]

Definition of External Interfaces.

Requirements for flight control software must include a definition of the external interface of the software, including the formats of the input and output values. The authors consider Ada a good choice for specifying these formats and advocate its use starting with the requirements definition. The following examples repeat some of these probably well-known features of Ada.

```
type switch_values is (neutral, on, off);
  for switch_values use (neutral => 1,
                         on => 2, off => 4);

  C_Small_180 : constant := 180.0 * 2**(-15) ;
type T_Fixed_180 is delta C_Small_180
                         range -180.0 .. 180.0 ;
  for T_Fixed_180 'small use C_Small_180 ;
  for T_Fixed_180 'size  use 16 ;
```

Such Ada definitions, firstly, have precise semantics according to the language definition. Secondly, the use of the representation clauses (“for ...”) allows a precise format definition down to the bit level. For instance, the first definition ties the switch values to their bit representations in say a digital input or output register, the second definition associates fixed point values with the format of an analog digital converter, for instance. Note the use of a delta which is not a power of two in the latter case. This represents a very common situation in practice.

Most importantly, using such definitions in the software requirements specification automatically guarantees consistency between (this part of) the requirements with the final code. The authors feel this method to be superior to rephrasing such requirements in some formal notation, and then perform proofs of consistency with the final Ada code. On the other hand, using UML would almost certainly be inferior, because there is no semantics and not even a syntax for describing types in UML[5].

Software Design and Implementation

In a research and technology project of DaimlerChrysler Aerospace, an integrated process for the development of control laws for complex aircraft configurations has been investigated. As part of this project control laws for a typical fighter aircraft have been implemented in Ada. The resulting software is called COLAda (Control Law Software in Ada). In a companion publication, various architecture and design aspects of COLAda are reported (Roßkopf, 1999). Here, the focus is on the following three general issues:

- using object-oriented design techniques
- design for multi-processor targets
- using finite state machines

Using Object-Oriented Design Techniques.

In the design of COLADA object-oriented techniques have been used, where appropriate. Certain control law elements, e.g. filters, have been identified as candidates for objects, and corresponding abstract data types. (e.g. filter types) have been defined. Several objects of these types may be created as required by the control law application–the objects are just instances of abstract data types. The external behaviour of these objects is defined by the operations associated with the abstract data type. Of course all internal data are encapsulated in the objects.

According to a certain terminology, the design might be termed to be object-based, as no type exten-

[4] Finally, use case diagrams could be used. It is still unclear to the author, whether use cases are just a recurrence of the once condemned functional decomposition models such as Structured Analysis, or whether they also contain some fundamental new ideas (in UML version 1.3, eventually).

[5] A rudimentary type system and some other formalisations have been added to UML in version 1.1 with the help of the so-called Object Constraint Language OCL, though.

sion (“inheritance”) is used. This is for the following reasons:

- Full use of inheritance, in particular polymorphism and dynamic binding (i.e. the use of class-wide types in Ada terminology), may cause certain problems in safety-critical systems (cf. ISO/IEC, 1998).
- There is hardly a need for using inheritance in the context of this project. The cases where variants of certain object types occur (e.g. first order and second order filters) can easily be covered without inheritance.

Though certain aspects of flight control software can be designed and implemented in an object-oriented way, and though the authors very strongly favour such an approach, there must be a warning against hypocrisy with respect to object-orientation: The approach “everything is an object” does not seem very helpful. Over-simplistic approaches like in Coad et. al., where a case study of an object-oriented auto-pilot system is reported on, seem to be impractical for the implementation of real flight control software. And, of course, normal arithmetic operations and static typing are to be used in control law software. This is in contrast to the philosophy of radical object-oriented languages such as SmallTalk and Lisp, for instance, which are considered unsuitable to real-time safety-critical systems.

... ...

References

Coad, P., North, D., Mayfield, M.: Object Models. Strategies, Patterns, and Applications. Yourdon Press, Englewood Cliffs, 1995.

Ellis, J.R.: Objectifying Real-Time Systems. SIGS Books, New York 1994.

ISO/IEC: Working Draft 3.8 - Programming Languages - Guide for the Use of the Ada Programming Language in High Integrity Systems. ISO/IEC PDTR 15942, ISO/IEC JTC 1/SC22/WG9, October-29, 1998.

Kaul, H.J.: Flugsteuerungssystem Jäger 90 (In German). In: G. Bürgener (Schriftleitung). Jahrbuch 1992 III der Deutschen Gesellschaft für Luft- und Raumfahrttechnik e.V. (DGLR). Deutscher Luft- und Raumfahrtkongreß 1992. DGLR Jahrestagung, Bremen, 29. September – 02. Oktober 1992. Deutsche Gesellschaft für Luft- und Raumfahrt e.V. (DGLR), Bonn 1992.

Rosskopf, A.: Development of Flight Control Software in Ada - Architecture and Design Issues and Approaches. Submitted to: Ada-Europe'99. International Conference on Reliable Software Technologies. June 7-11, 1999. Santander, Spain.

Schwald, A.: UML und Ada95 – Über das Zusammenwirken zweier Stars. (UML and Ada95 – About the Cooperation of two Stars. In German.) Vortrag auf dem Workshop „Objektorientierung und sichere Software mit Ada“ der GI-FG 2.1.5 Ada „Ada-Deutschland“, 21. und 22. April 1999, Karlsruhe, Germany. To appear as a report of the Forschungszentrum Karlsruhe.

Tempelmeier, T.: An Overview of the HOOD Software Design Method. In: Real Time Computing. NATO ASI Series F, Vol. 127. Halang, W.A. and Stoyenko, A.D. (eds.). Proceedings of the NATO Advanced Study Institute on Real Time Systems, Sint Maarten, Dutch Antilles, October 5-17, 1992. Pages 726-734. Springer Verlag, Berlin, Heidelberg, New York 1994.

Tempelmeier, T.: Formal Methods – An Informal Assessment. Technischer Report Dasa MT36 SR-1775-a. 41 pages. Daimler-Benz Aerospace, Ottobrunn August 1997.

Tempelmeier, T.: Hierarchical Object-Oriented Design (HOOD) – Die Software-Entwurfsmethode der europäischen Raumfahrtbehörde ESA. (Hierarchical Object-Oriented Design (HOOD) – The Software Design Method of the European Space Agency ESA. In German.) Kolloquiumsvortrag an der Universität Oldenburg, 29.6.98.