

Defining an Architecture Description Language for Dassault Systèmes

Yves Ledru¹, Rémy Sanlaville², Jacky Estublier¹

¹ Laboratoire Logiciels, Systèmes Réseaux - IMAG

B. P. 72 - F-38402 - Saint Martin d'Hères Cedex - France

{Yves. Ledru, Jacky. Estublier}@imag.fr

² Dassault Systèmes

9, quai Marcel Dassault, B. P. 310 - F-92156 - Suresnes Cedex - France

remy_sanlaville@ds-fr.com

1. Introduction

Architecture Description Languages (ADL) are a major research theme in the software architecture community. Although many interesting proposals have been made, they don't seem to influence the daily industrial practice. In this paper, we report on an attempt to define an ADL that fits the needs of a specific company: Dassault Systèmes. This company maintains and develops a large body of software with very short cycle times. Their relationship with partners and OEMs has led them to develop the Object Modeler: a component technology for open architecture (Sect. 2). We then give the major requirements on the ADL (Sect. 3), and briefly explain why existing ADLs are not satisfactory in this context (Sect. 4). We finally discuss our approach which takes advantage of the Object Modeler technology to ground the definition of our ADL (Sect. 5).

We believe that this strategy for the definition of an ADL is not specific to Dassault Systèmes and the Object Modeler technology, and could be adapted to other component technologies like the Java beans or DCOM.

2. Context

2.1 Software development at Dassault Systèmes

Dassault Systèmes is one of the world leaders in Computer Aided Design products. Its software development activity is characterized by the large size of its existing software products (4 Mloc). Moreover, Dassault Systèmes customers, partners and OEMs have build an even large body of software on top of the CATIA products.

There are several reasons why a software company like Dassault Systèmes is interested in mastering the architecture of its products. A first reason is that a CAD company is obviously aware of the interest of using CAD techniques for its own products. A more practical reason is that the size of a software product like CATIA implies architectural reasoning in order to allow concurrent developments by 700 software engineers and to master its complexity.

The next section shows how these architectural concerns, and limitations of C++, have led the company to define the «Object Modeler» layer on top of C++. This definition corresponds to the integration of several architectural constructs at the level of the programming language. It also corresponds to state of the art technology in the field of open architectures.

2.2 The Object Modeler

In the recent years Dassault Systèmes has moved its products from Fortran to C++, resulting in a huge redevelopment effort which succeeded at the beginning of 1999 with the release of CATIA Version 5. The developers at Dassault Systèmes rapidly discovered that C++ as such was not suited to their needs. Their concerns were mainly of three natures:

- *incremental compilation concerns*. Compiling and building a system as large as CATIA leads to unacceptable delays; therefore very precise incremental compilation mechanisms, combined with a programming discipline, were required to limit the impact of evolutions of the software.
- *open architecture*. Concurrent engineering inside the company as well as working with third parties requires to provide as much openness of the system as possible without releasing too much information on its implementation. In particular, it is unacceptable for Dassault Systèmes to give its source files to its partners. The solution to this problem is to provide *open architecture* mechanisms where components can be extended without requiring access to source files for compilation (Figure 1). This also allows the company itself to add custom extensions for a specific customer without impacting the main commercial release or requiring the customer to re-install the whole software.
- *multiple inheritance* is difficult to combine with incremental compilation ; Dassault Systèmes adopted a multiple interfaces technology instead, like in Java or COM. ¹

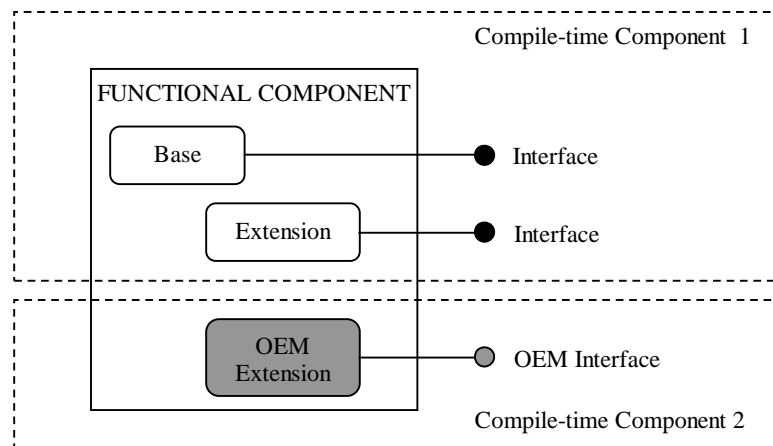


Figure 1: Extended component

Therefore, the Dassault Systèmes engineers have defined an additional layer on top of C++, named the Object Modeler. Fig. 1 shows several features of this layer: the notion of multiple interfaces of a class (close to the Microsoft COM mechanism), the possibility to «extend» a functional component by another object which adds additional interfaces to the component (some kind of aggregation mechanism). This extension feature, combined with modular mechanisms, also allows third parties to add their own extensions to an existing component without access to its source code. Moreover the Object Modeler includes an extended notion of inheritance (taking into account the notion of interfaces), and mechanisms aimed at easing the build activities. The design of this Object Modeler layer took also into account performance requirements resulting from the highly interactive nature of CATIA. The Object Modeler information is expressed in the C++ code as macros, and in dictionaries associated to this code.

¹ Another reason for not using multiple inheritance is that CATIA software is aimed at running on multiple platforms; it appears that C++ compilers did not always exhibit the same behaviour on all platforms with respect to multiple inheritance.

2.3 The Dassault Systèmes/LSR collaboration

Recently, Dassault Systèmes and the LSR laboratory have started several projects related to software architecture. The major theme of this research collaboration is the design of a support environment for architecture development. Such an environment should provide services to help reasoning on a given architecture, and to control its evolution. Defining the right services is one of the goals of this research project. In this context, the definition of an architecture description language (ADL) which fits Dassault Systèmes needs appears as a key element for this support environment.

3. Requirements for an ADL

Finding out the actual requirements for the Architecture Description Language appeared not to be an easy task, mainly because different stakeholders in the company have different concerns related to software architecture. For example, at the developers level, structural aspects of the software related to its functionalities do not necessarily match the modular structure designed for incremental compilations. These constitute two different views of the architecture of a system. Other views of the software architecture emerge in discussions with managers who have a more global but less technical perception of the architecture. We finally decided to target our effort towards the functional aspects of the architecture.

Another aspect is the distinction between a global view of the architecture which shows the overall picture of the software, and a local view which shows a restricted set of components with their relations. It appeared to us that this local view can be particularly useful for software developers by showing them the environment in which the component they are working on will be integrated, and highlighting the functional structure of this component. In an open architecture context where different partners are working on the same component, this view gives the developers an abstract view of their direct environment.

So our ADL needs to be well-suited for the expression of the functional aspects of the architecture and should address the local view in priority.

Tools are another aspect of requirements. They are expected to fulfill the following objectives:

- *Tutorial*: help new developers or third parties to rapidly learn how to properly define and integrate components in the open architecture of CATIA;
- *Decision support*: a developer will visualize the impact of architectural decisions like the adhesion of his component to a new interface or its extension by an existing component;
- *Code synthesis*: generate code for the structure of a designed component.

Other aspects must be taken into account in the context of such a company.

- The cycle time between two releases of the software is 4 months. The actual architecture of CATIA may evolve rapidly. Architectural descriptions and code evolution must keep synchronized, otherwise architectural descriptions will become useless.
- A large body of software (4 Mloc) already exists. This means that descriptions should be constructed a posteriori for large portions of the architecture.

Both the cycle time and the software size mandate the use of automatic tools that maintain the link between architecture and code. Such tools should reconstruct the architectural description from the code or synthesize code from the architectural description (and preferably both).

The ADL should not only be machine processable, other objectives include its eventual adoption as a «design esperanto» used by the developers for informal sketching on blackboards during architectural discussions as well as a more formal means to describe architectural aspects in the documentation.

4. What about existing ADLs?

Instead of reinventing the wheel, a sound engineering and scientific practice is to look at existing ADLs in order to check if they meet these requirements. Unfortunately, current ADLs did not appear to be satisfactory for several reasons:

- Many ADLs focus on the verification [1] or the simulation [2] of the interaction protocols between components. Although verification is an important business in building large software systems, it did not appear in the priority objectives of the expected ADL.
- Similarly, when tools are available for existing ADLs [2,3,4], these are mainly targeted towards behavioural aspects of the architecture. In our context, a mandatory requirement is the availability of tools that maintain the correspondence between the graphical view of the architecture and the actual code.
- Most ADLs are domain-independent, which makes them difficult to adapt to the Dassault Systèmes Context where a specific component technology is being used.
- Last but not least, most ADLs introduce design concepts (components, connectors, ports) that do not immediately match the coding concepts (classes, interfaces, ...). Architectural reasoning requires to make more precise the relationship between these abstract concepts and their realization at the code level. In the context of Dassault Systèmes, even the concept of component does not lead to an unanimous agreement on its realization (e.g. functional components differ from compile-time components in Fig. 1). Figuring out how an abstract notion like «connectors» can be matched to some coding reality, and hence adopted by software engineers, is a non obvious task.

Our conclusion was that today, the scientific and technical community has no ready-to-use ADL well-suited to Dassault Systèmes needs.

5. Our approach

In order to define a specific ADL which fits the needs of Dassault Systèmes, we adopted a bottom-up strategy which takes advantage of the architectural concepts of the Object Modeler and aims at eventually building several layers of abstraction on top of this information. This allows the construction of tools which will ensure a close coupling of architecture and code evolution.

The notion of functional component takes the central role in our ADL. The ADL expresses:

- How a component is constructed, taking into account various kinds of base implementations and extension mechanisms provided by the Object Modeler.
- Which interfaces are implemented by the component, taking into account the inheritance mechanisms of the Object Modeler.

- How it relates to other components. Relations between the components mainly correspond to method calls in some interface of another component.

All this information can be extracted from the code. We have already developed navigation and visualization tools based on architectural information extracted directly from the code and we plan the future development of simulation and edition tools.

Nevertheless, at this stage the ADL is not yet satisfactory. Although the Object Modeler provides higher level structural information than classical C++, further structuring mechanisms, which are not currently grounded in the code, should be added to the ADL. First the relations between components, are based on low level information (procedure calls). This is a place where we foresee a potential for the introduction of some notion of connector. We expect that a careful examination of the concrete links between components will help identify the concrete nature of potential connectors. Based on this concrete information, it might be possible to include the notion of connector into our ADL, and maybe to add this concept to the Object Modeler, which would allow to ground the notion of connector into the actual code.

A second weakness of this model is the non recursive nature of the notion of component : components are made of objects, not of other components. CATIA v5 includes about 4000 components. It is therefore interesting to define new levels of structuring above the current notion, which would lead to consider more abstract levels in the structure, following our bottom-up approach.

6. Conclusion, lessons learned so far...

This paper has presented our first efforts in the definition of an Architecture Description Language for a specific software company. The major objectives of this ADL are the representation of local aspects of the functional architecture and the possibility to build tools for tutorial, decision support and code generation purposes. Key elements in this context are the large body of existing software and the short cycle time between releases.

In this context, we ended up with a bottom up approach for the definition of an ADL grounded in the concepts of the Object Modeler. This approach has two main advantages. First, it relies on existing concepts which are used and applied every day by the software engineers of the company. This should favour an easier integration of the ADL in their software engineering practice. Second, the architecture information is directly extracted from the Object Modeler code. This allows the development of tools which provide an architectural representation consistent with the current version of the software. Our first prototype tools are thus working on the actual architecture of the whole CATIA system. This gives us a more accurate understanding of the complexity of the existing architecture.

As such, the ADL is not satisfactory because it does not provide enough structuring mechanisms to handle the complexity of the architecture. Further work includes the definition of additional structuring constructs, like hierarchical components, structured interfaces or various kinds of connectors. Their integration in both our ADL and the Object Modeler layer will keep the close coupling between code and architecture.

References

- [1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213--249, July 1997.
- [2] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336--355, April 1995.

[3] Keng Ng, Jeff Kramer, and Jeff Magee. A CASE tool for software architecture design. *Automated Software Engineering: An International Journal*, 3(3/4):261—284, August 1996.

[4] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60—76. Lecture Notes in Computer Science Nr. 1013, Springer–Verlag, September 1997.