Using an Architecture Description Language to Model a Large-Scale Information System – An Industrial Experience Report

Eoin Woods
Artechra
Hemel Hempstead, Hertfordshire, UK
eoin.woods@artechra.com

Rabih Bashroush
University of East London
London, UK
rabih@uel.ac.uk

***Abstract* — An organisation that had developed a large Information System wanted to embark on a programme of significant evolution for the system. As a precursor to this, it was decided to create a comprehensive architectural description. This undertaking faced a number of challenges, including a low general awareness of software modelling and software architecture practices. The approach taken for this project included the definition of a simple, specific, architecture description language. This paper describes the experiences of the project and the ADL created as part of it.**

Keywords-Architecture Description Language; ADL; Software Architecture Discovery; Software Architecture for Legacy Systems; Industrial Experience Report.

I. INTRODUCTION

There has been a great deal of academic and some industrial research into the definition of Architecture Description Languages (ADLs) to assist with the task of defining the architecture of software intensive systems and a significant amount of research is still underway today [1, 2]. However, for reasons that have been noted elsewhere [3, 4], there has been little significant industrial use of ADLs, particularly in the Information Systems domain.

Recently, one of the authors of this paper had the opportunity to lead the creation of a large architectural description (AD) for a complicated Information System. This paper describes the experience of that project, which was used as an opportunity to explore the use of a simple, domain specific, ADL in an industrial context.

This paper provides an overview of related work on ADLs in both industry and academia in Section 2. Section 3 provides background information about the work and the context of the project. The approach used is described in section 4. The ADL design, along with the system architectural style is presented in section 5. The experience and lessons learned from the project are discussed in sections 6 and 7 respectively. Finally, section 8 completes the paper with the summary and conclusions.

II. RELATED WORK

Over the past two decades, an increasing number of ADLs have been developed, largely within academia [5, 6]. Although some ADLs have been put to industrial use in specific domains [7], the majority of ADLs remain confined to laboratory-based case studies.

While ADLs originated in academia such as ACME[8], Wright[9], Rapide[10], SADL[11], xADL[12], pi-ADL[2], ArchiMate[13] and ByADL[1], to name a few, all exhibit novel approaches to architecture description, but most are vertically optimized, restricting their application in industrial settings. In general, academic ADLs focus more on analytical evaluation and rigour while in this project the focus was more on practicality and obtaining a system-wide view of the application. (ArchiMate is the exception, being an enterprise-architecture ADL, whereas the focus of this project was system architecture description.)

Most industrial applications of ADLs have been in the area of embedded systems, from consumer electronics (e.g. Koala[7]) to automotive systems (e.g. AADL[14]), supporting automated system analysis and automated code generation, which were not primary concerns in this project.

III. OVERVIEW OF THE PROJECT

This project was undertaken in a financial services firm that had developed a large custom application suite to run its business. The software has been developed over a period of about 15 years and has grown from modest beginnings to a large system comprising about 20 major subsystems and over 10 million lines of Java, C++, C# and Perl, sharing a large multi-terabyte relational database. Today it has grown to a size that means no individual understands it all, even at a reasonably high level of abstraction.

At the start of the project, there was no overall unified system description, but the organization wanted to perform some wide ranging evolution and modernization of the system, so it was decided to undertake the creation of a unified description of the system's architecture. Two experienced architects were assigned this task.

One immediate complication was that it wasn't clear what the AD would be used for once created, so in order to make progress, some assumptions were made and these were:

- The point of the exercise was to (a) understand what was there today (catalogue); (b) allow change to be planned (allow impact analysis); and (c) provide a reference for people to build knowledge (communicate); and
- The audience for the documentation was architects & designers, so precision and completeness were important attributes.

Using these assumptions, the architects tasked with the project defined an approach that allowed them to capture a suitable AD for the system.

IV. The Approach Used

When the project was discussed with the software development teams it quickly became clear that while there was general enthusiasm for the idea, there was very little appetite for actually performing the work. This led to the use of a simple, low-ceremony, tailored and prescriptive approach to minimise the effort required from the teams and to avoid creating inconsistent artefacts.

Using a tailored (profiled) version of UML was seriously considered, however the organisation did not have sufficient UML tooling available and even a tailored UML tool tends to need some background knowledge of UML that was lacking in most of the teams. Existing ADLs (see section 2 above) were also briefly considered, but none of these appeared to offer any great benefit over UML and, like UML, all would have needed significant tailoring, training and tool support. Therefore, we decided to develop a simple graphical and textual language to model the system.

The discussions with the teams revealed a varied understanding of modelling and abstraction, which led to the realisation that the best approach was going to be creating models that captured the physical structure of the software (processes and inter-process communication channels) rather than more abstract concepts such as software components and responsibilities. Otherwise, the project was going to collapse under the weight of debatable abstractions that could not be validated against the existing implementation.

Given the environment, it was decided to use a wiki to capture the data underpinning a graphical representation (the system element descriptions, connection definitions and so on). The wiki captured this information in an accessible way, but allowed very restricted formats to be prescribed that would be amenable to basic machine parsing later if needed.

The wiki approach of creating hyperlinked pages also allowed the AD to be decomposed into a set of manageable pieces, linked together to provide cross referencing and navigation through the documentation.

V. The Style and its Description Language

A. The Architectural Style

An analysis of the system implementation revealed that it followed a discernable architectural style (although there was no explicit awareness of the concept an architectural style within the organisation).

A few basic definitions were used to provide people with a common starting point for understanding key abstractions:

- Subsystem - a subset of the system that has a well-defined, cohesive, set of responsibilities, a well-defined boundary and set of interfaces to its services.
- Component - a tangible software artefact which is delivered to the production environment and which is "executed" in some way at runtime. (In line with other software architecture literature, components are referred to as “elements” elsewhere in the paper).
- Connector - the mechanism by which two or more components collaborate. Examples are a message destination, a file system file, or a database table.

The specific types of system element used within the system are summarised in Table I.

TABLE I. Types of Architectural Elements

User Interfaces	
- GUI	A traditional GUI client written in Java Swing, C# WebForms or C++ Motif.
- WebUI	A user interface implemented as a set of web pages (e.g. CGI scripts or a Java webapp)
- Command Line	A user interface implemented as a command line program, such as a Python script.
Servers	
- Message Driven Server	A server whose operation is driven by the recipt of messages from the message bus
- Server	A server whose operation is driven by a mechanism other than messages (such as RPCs or temporal schedules)
- Batch Program	A program that is run from a scheduler and runs in a single execution, without input from system element or humans.
- Data Loader	A program that extracst data from a source and moves it to a destination, typically transforming it during the transmission.
Data Stores	
- System database	The system database or a set of tables from it
- File	A file on the file system
External Entities	
- Subsystem	Another subsystem that communicates with this one in some way
- External System	A system outside our system that a subsystem communicates with in some way
- External Data Source	A Data Source outside our system that a subsystem receives data from (such as a source of security prices)

The fairly restricted set of inter-element connectors in general use throughout the system is described in Table II.

TABLE II. Types of Architectural Connectors

RPC	A synchronous inter-process procedure call (usually XML over HTTP)
Direct Invocation	An in-proess direct procedure invocation
Database Data Flow	Writing data to a database table or tables to allow it to be used by another element
File Data Flow	Writing data to a filesystem file to allow it to be used by another element
System Messaging	Dispatch and receipt of messages over the system bus via a messaging destination

In order to allow for the inevitable special cases that are found in a system of this scale, an “other” type was allowed for elements and connectors, which could be annotated using a UML style stereotype to make its type clear.

Most architectural styles limit the element and connector combinations that they allow. In this style, there weren’t really any such constraints defined formally, although there were combinations that were encouraged and discouraged (e.g. UI Clients should connect to Message Driven Servers, but not access the database). However, most combinations of element and connector types could be found somewhere in the system! A number of the desirable patterns were captured as examples in the notation documentation.

A couple of examples of the patterns identified are shown in Figure 1. The notation used to express the examples is hopefully fairly obvious but is explained in the next section.

B. The Architecture Description Language

Once the required element and connector types were understood, a notation was required to represent them. Given people's preference for diagrams over text, a graphical notation was created rather than a textual one. When defining the graphical detail of the notation, the advice in [15] was particularly useful, in particular the exhortation to avoid construct overload, deficit, redundancy or excess, the suggestion to systematically consider the various visual variables of each shape (shape, size, colour, orientation, brightness and texture) and the need for deliberate selection of shapes so that their appearance suggested their meaning, in order to achieve semantic transparency.

The graphical notation was designed by selecting a base shape for each major type of element (server, user interface, data store, external entity) and a variation on the shape using the dimensions of shape, line and texture was identified for each subtype. Examples of the notation for some of the element types are shown in Figure 2.

As can be seen from the diagram, a triangle was used as the base shape for user interfaces and a rectangle for server resident components. The triangle was chosen as it hinted at the head and shoulders shape of a user and the triangles were then modified slightly for each type of user interface (the thick client having sharp corners, the web user interface having rounded corners as it blurs the distinction between "client" and "server" and the command line utility having a graphical representation of a command line interface added to it). Similarly, a rectangle is the base shape for server elements (based on long accepted conventions) with a stereotype being used to indicate the type of server and a "lozenge" variant being used to indicate a data loader (hinting at pieces of data being transmitted through it).

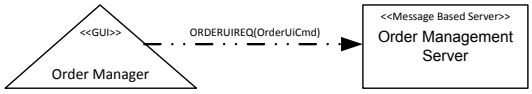
An arrow of some form was used to represent all connector types, with the arrowhead usually indicating the direction of data flow. All connectors were defined to be one way connections, with the exception of connectors to files, which could indicate read and write activity with arrow heads at both ends of the connector if appropriate. The convention for RPC connectors was defined to be a one-way arrow from the caller to the target, textually annotated to indicate what it transmitted (message data type, table or record names or service invocation name). Examples of the notation for the main connector types are shown in Figure 3.

The RPC or direct procedure call is shown using a solid arrow and messaging is shown using a line with embedded dots, suggesting messages flowing over it, while data access is shown using a regular chain line, suggesting records being read or written over the connector.

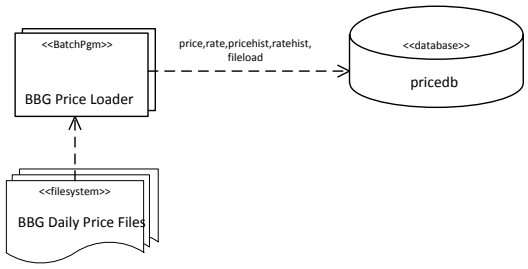
A general mechanism used on elements and connectors was the stereotype, copied from UML, where the type of an architectural element is made clear by annotating it with a type name using the convention "«typename»" on the symbol concerned.

In order to ensure that the process produced more than just pictures, a set of required attributes for each type of element and connector was defined and wiki table templates created to allow them to be captured in a standard form.

In order to simplify and standardise the subsystem descriptions, a set of wiki page templates and a comprehensive Microsoft Visio stencil were created, along with clear instructions, quick reference material and – most crucially – a fully worked example of the documentation for one subsystem.



(a) Thick client UI and message driven server



(b) File Loader, reads files, writes to database

Figure 1. Examples of the ADL Notation in Use

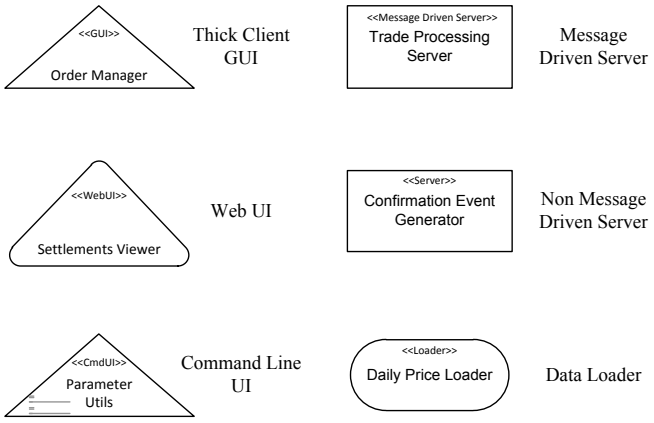


Figure 2. ADL Element Types

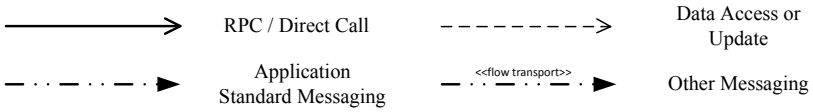


Figure 3. ADL Connector Types

VI. The Experience Gained

A. Creating the Architecture Description

The development teams were tasked with the creation of architecture description documents for their subsystems. The success of this approach varied, with some teams producing their documentation largely unaided, while others needed significant assistance from the architects running the project.

There were varied reasons for the difficulties that some development teams faced. In some cases it was simply a lack of interest, while in other cases there seemed to be a genuine difficulty in understanding how to represent their subsystem. In general, this seemed to stem from an inability to abstract from the implementation, resulting in a confusing mix of concrete and abstract concepts in their models.

Another interesting problem was tooling. Everyone in the organisation could use the wiki, but many did not have Microsoft Visio and of those that did, some couldn't use it. This was a useful lesson and confirmed that avoiding specialised modelling tools had been a good decision.

Over the period of a couple of months, a useful body of subsystem descriptions emerged, which allowed the architects to create a summary level description that showed how the subsystems related to each other. This was a manual process, aided by some drawing tool macros and some use of scripting to process wiki text.

The process of capturing the AD took about six months, with the architects working on it approximately 60% of their time and the development teams working on it as their project schedules allowed.

B. The Results of the Project

The outputs of the project were:

- A consistent AD that provided an accurate view of subsystems, components and their dependencies.
- An informal definition of the architectural style used across most of the system.
- A degree of oversight and understanding of the structure, scale and connectedness of the system.

As mentioned earlier, there weren't clear goals for the AD but it was found to be insightful and there seemed to be a general consensus that it was a useful description. However organisational priorities meant that the architects then moved on to other work, so the project effectively ended.

C. Evaluating the Usefulness of the ADL

By the time that the descriptions for key subsystems were complete, the notation and approach were judged to be fairly successful (an outcome which was not widely predicted at the start of the project). Early experience led to some rapid refinement of the notation to remove ambiguities and to introduce some missing concepts, but after three or four teams had used the approach the ADL remained stable.

During the project it became clear that teams who could identify clear abstractions found the ADL helpful and they had little difficulty in representing their models using it. In contrast, teams who struggled to identify good abstractions never really grasped how to use the ADL and needed constant assistance from the architects running the project.

We viewed this experience as a basic validation of the approach. People who could create models and knew what they wanted to represent were able to use the ADL effectively, with minimal training, which is an important validation point for an ADL. However, the approach did not help people who found modelling difficult. We had hoped that the straightforward and prescriptive approach would help people to create models even if they did not find modelling easy, so it was disappointing that it failed to achieve this.

VII. Lessons Learned from the Project

The lessons learned during the course of the project were:

- A specialised ADL can have benefits over a general modelling language like UML and even a simple ADL can be used to create useful results.
- The more directly that an ADL matches the concepts of the system being modelled, the easier people seem to find it to use. While this may seem obvious, it contradicts the conventional approach of using a general modelling language like UML or SysML.
- Designing the detail of the graphical notation carefully pays off. Using shapes that hint at their meaning and a range of graphical dimensions to differentiate shapes helps people to remember them.
- Consistency in the notation is very important and having a base shape for a general concept with refinements to it for different sub-concepts appears to help people when interpreting the diagrams.
- It is important to provide high quality support materials such as templates, an example-based description of the approach, and a number of realistic examples. People are better at "filling in the gaps" rather than creating something new.
- Utilising familiar tools helps with acceptance. In this case a Wiki was immediately accepted whereas a ubiquitous commercial drawing tool caused problems, even with a carefully tailored template.

These lessons may not be surprising, but the importance of quite simple factors was surprising to us and is useful to bear in mind for the future. It is also worth noting that these lessons may well have general applicability, but only in the broad sense. People like to be guided and prefer familiar tools and techniques, but the tools or techniques that work will be specific to an environment and people in different environments will have different levels of enthusiasm for learning new approaches. However, in general, familiarity and accessibility help greatly with acceptance.

VIII. Summary and Conclusions

An organisation in the financial services industry wanted to create an AD for a large system. A simple, custom ADL was defined in order to make the process of capturing the AD as simple and prescriptive as possible, which proved to be a helpful tool for capturing this specific architecture.

However the ADL did not help those who found modelling difficult. People who found abstraction difficult found it just as difficult to use a specific notation as a general-purpose notation, which was surprising.

The key factors that appear to have made the approach successful where its specific tailoring to the situation, its simplicity (which traded sophistication for accessibility), a carefully designed, consistent graphical notation, the availability of a large amount of tutorial and reference material, and the use of already familiar tools.

References

[1] D. Di Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio, "ByADL: an MDE framework for building extensible architecture description languages," in *Proceedings of the 4th European Conference on Software Architecture*, Copenhagen, Denmark, 2010, pp. 527-531.

[2] F. Oquendo, "Pi-ADL: an Architecture Description Language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures," ed. SIGSOFT Softw. Eng. Notes, 2004, pp. 1--14.

[3] E. Woods and R. Hilliard, "Architecture Description Languages in Practice Session Report," in *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, Pittsburgh, Pennsylvania, USA, 2005, pp. 297-304.

[4] R. Bashroush, I. Spence, P. Kilpatrick, and T. Brown, "Towards More Flexible Architecture Description Languages for Industrial Applications," presented at the EWSA 2006, Nantes, France, 2006.

[5] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering,* vol. 26, pp. 70-93, 2000.

[6] P. C. Clements, "A Survey of Architecture Description Languages," in *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996, p. 16.

[7] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *IEEE Computer,* vol. 33, pp. 78-85, 2000.

[8] D. Garlan, R. T. Monroe, and D. Wile, "Acme: architectural description of component-based systems," in *Foundations of component-based systems*, T. L. Gary and S. Murali, Eds., ed: Cambridge University Press, 2000, pp. 47-67.

[9] R. Allen and D. Garlan, "The Wright Architectural Specification Language," Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA1996.

[10] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using Rapide," *IEEE Transactions on Software Engineering,* vol. 21, pp. 336-354, 1995.

[11] M. Moriconi and R. A. Riemenschneider, "Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies," SRI International,1997.

[12] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and R. N. Taylor, "xADL: enabling architecture-centric tool integration with XML," in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, 2001, p. 9 pp.

[13] M. M. Lankhorst, H. A. Proper, and H. Jonkers, "The Architecture of the ArchiMate Language," in *Proceedings of the 10th International Workshop on Enterprise, Business-Process and Information Systems Modeling (BPMDS 2009) held at CAiSE*, Amsterdam, Netherlands, 2009, pp. 367-380.

[14] "Standard AS5506/1: SAE Architecture Analysis and Design Language (AADL)," ed: SAE International, 2006.

[15] D. Moody, "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering," *IEEE Transactions on Software Engineering,* vol. 35, pp. 756-779, 2009.