Towards More Flexible Architecture Description Languages for Industrial Applications

R. Bashroush, I. Spence, P. Kilpatrick, TJ. Brown

School of Electronics, Electrical Engineering and Computer Science,
Queen's University Belfast, N. Ireland, UK
{r.bashroush, i.spence, p.kilpatrick, tj.brown}@qub.ac.uk

Abstract. Architecture Description Languages (ADLs) have emerged in recent years as a tool for providing high-level descriptions of software systems in terms of their architectural elements and the relationships among them. Most of the current ADLs exhibit limitations which prevent their widespread use in industrial applications. In this paper, we discuss these limitations and introduce ALI, an ADL that has been developed to address such limitations. The ALI language provides a rich and flexible syntax for describing component interfaces, architectural patterns, and meta-information. Multiple graphical architectural views can then be derived from ALI's textual notation.

Keywords: Software Architecture, Architecture Description Languages, Architectural Patterns.

1 Introduction

Architecture Description Languages (ADLs) have emerged as viable tools for formally representing the architectures of systems at a reasonably high level of abstraction to enable better intellectual control over the systems [1]. ADLs usually help in architectural analysis with issues such as consistency, modifiability, performance, etc. However, there is no general agreement on what ADLs are expected to capture/represent about an architecture (behavior, structure, interfaces, etc.). Most work on ADLs today has been undertaken with academic rather than commercial goals in mind and they tend to be very vertically optimized towards a particular kind of analysis [2].

The ADL community generally agrees that a Software Architecture is a set of components and the connections among them conforming to a set of constraints. Component interfaces usually comprise a set of provided and required services (a service could be a function call, a message type, etc.).

Although some ADLs have been put to industrial use [3], the majority of ADLs have not scaled up well, and their use remains confined to small-scale case studies.

In this paper we discuss a number of limitations evident in most current ADLs which might have constrained their use to small-scale academic applications. We then present the major concepts behind the ALI ADL which has been designed with the identified limitations in mind. ALI also built upon our experience with the ADLARS

[4] ADL and adopted much of the solution space provided by ADLARS such as its support for Software Product Lines.

In the following, we begin in Section 2 by discussing the limitations within current ADLs. Section 3 then highlights the rationale behind the ALI language. Finally, discussion and future work is presented in Section 4.

2 Limitations within Existing ADLs

In this section we discuss the potential limitations identified by examining a number of existing and mature ADLs selected from across the literature to reflect the state-of-the-art in the domain. Among these ADLs are: ACME [7], Koala [3], Rapide [8], and Wright [9].

It is worth mentioning here that the Unified Modeling Language (UML) [5], even though it is used within different stages of the development process (and without doubt a *de facto* modeling language), is not considered a strong candidate as an ADL due to many issues including it being a pure graphical notation and the fact that it does not treat connectors as first class citizens (even though UML 2.0 [6] took one step further in the ADLs' direction with the introduction of ports and interfaces). Furthermore, UML initially was geared more towards code description rather than architecture description.

We have examined and experimented with these ADLs to identify the novelty and the strengths of each. We have also identified a number of shared limitations, particularly in the context of real-life applications. These are summarized below.

2.1 ADLs are Over-constraining

Current ADLs force architects to use specific styles/interface types throughout their architecture by providing a single component interface type model. For example, while interfaces are described in terms of input and output ports in Wright, interfaces are described in terms of services provided/required in Koala, and messages sent/received in ADLARS [4]. With current advances in different domains including Service Oriented Architectures (SOA) and adaptive systems, within a single system we could have a number of different interface types used (which is often the case). Capturing such architectures with most current ADLs entails abstracting a number of interface types to the single interface type supported by the ADL. This could be problematic especially when the interface types form a crucial part of the architecture description (e.g. in SOAs). Also, by requiring that components have specific types of interfaces (hardware-like input/output ports, e.g. ACME; message based communication, e.g. ADLARS; etc.), ADLs may be indirectly enforcing the style of communication to be used in the system on the architect.

2.2 ADLs Provide a Single View of the System

It has become widely recognized in the software architecture community that software architectures contain too much information to be adequately captured and displayed in one view. Multiple views are needed to describe an architecture where each view can encompass a set of related concerns. This has been recognised in a number of industrial approaches [10, 11] and standards [ANSI/IEEE 1471-2000] (while others went one step further to consider also perspectives [12]). When this is the trend in industry, there is no reason why ADLs should not support multiple views. The reason why most ADLs are restricted to one view of a system may be attributed to the fact that ADLs inherently focus on the structural aspects of the architecture which has traditionally been the central issue. Hence, ADLs provide only the structural view of the system. Today's concerns have gone beyond purely structural factors, and issues such as quality attributes, design decisions, etc. are now considered an intrinsic part of architecture description [13].

2.3 ADLs Lack Proper CASE Tool Support

CASE tool support availability varies from one ADL to another. Some ADLs have parser/syntax validation tool support, others have basic simulation tools, while others have no tool support at all. For an industrial buy-in, tool support is a major selling point for any ADL due to the size and complexity involved in real-life systems. Even for those ADLs with tool support, most of the tools developed do not scale up to work with large system descriptions (e.g. hundreds of components and connectors). While some simulators are unable to cope with systems comprising over 100 components, most graphical tools have no mechanism to properly display systems with 30-40 components or more. This problem, however, differs from the previous two in the sense that for a commercial level tool support to be developed for an ADL, the ADL should be adopted by a tool vendor. For a tool vendor to adopt an ADL, the ADL should demonstrate a commercial potential (which is best done using proper tools!). A potential solution to this problem would be to make use of existing tool support for other notations such as UML in the first stage. This could perhaps be done by transforming back and forth between the ADL notation and UML (e.g. using meta ADLs like in [14]).

In the following section, we will introduce the rationale behind the ALI language which was designed with the aforementioned limitations in mind.

3 ALI Rationale

ALI has been designed on the basis of our previous work on ADLs, including the ADLARS notation [4]. It seeks to address a number of the issues discussed above.

While adopting successful concepts from ADLARS, such as the relationship between the feature model [15] and the architectural structure [16], ALI introduces, among other things, a high level of flexibility for interface description. Major concepts behind the ALI ADL are discussed in this section.

3.1 Flexible Interface Description

Revisiting the first limitation discussed in the previous section, current ADLs allow only for fixed interface types. Providing a specific interface type restricts the usage of an ADL to domains where most components would only have that particular type of interface. This is in addition to restricting the architect to use a specific style of communication among components (e.g. message-based, method invocation, hardware-like ports, etc.).

The ALI ADL attempts to address this limitation by providing no pre-defined interface types. Instead, ALI introduces a sub-language (which is a sub-set of the JavaCC [17] notation) that gives users the flexibility to define their own interface types.

For example, consider a simple web service having a WSDL (Web Services Description Language) interface and containing a number of components which are described with input/output ports as interfaces. Assume also, that each component contains a number of objects/classes that have interfaces defined in terms of functions provided/required (summarized in Fig. 1). This is a fairly standard level of nesting/abstraction within today's service oriented architectures.

If we were to model this using any of the existing ADLs, we would have to abstract the different interface types with the single interface type supported by the ADL used. By doing so, we would be unnecessarily abstracting away useful and important architectural information - especially in domains such as SOA where interface descriptions/types are of important architectural value.

It would also be difficult to identify a comprehensive set of interface types beforehand to be provided by an ADL due to the large number of interface types that already exist in the literature. In addition, new interface types emerge with the advancement of different technologies (e.g. GWSDL emerging from the work on grid computing, etc.). So, an ADL may benefit from a flexible mechanism that allows the architect to define his/her own interface types along with the binding constraints. This is the model that is adopted by ALI.

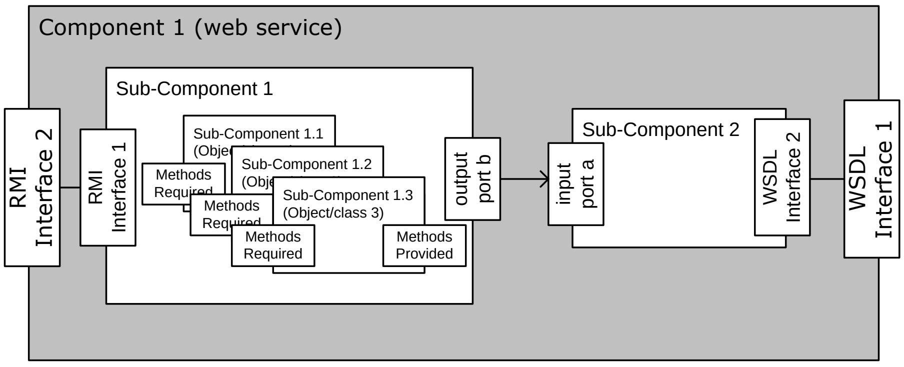

Fig. 1. An example architecture of a simple web service.

3.2 Architectural Pattern Description

Architectural patterns (or architectural styles) express a fundamental structural organization or schema for software systems and sub-systems. As these patterns are often reused within the same system (and sub-systems) or across multiple systems, providing syntax for capturing/describing these patterns to enable better pattern reuse is important. This is another major aspect of the ALI notation. ALI envisages architectural patterns as the architectural level equivalent of functions (methods) in programming languages.

Within ALI, patterns are defined and reused as functions. *Pattern templates* are first defined by specifying the way components are connected to form the architectural pattern. Then, these pattern templates are instantiated throughout the architecture definition to connect sets of components (whose interfaces are passed as arguments to the pattern template) according to the pattern template definition (e.g. Fig 2).

As shown in Fig. 2, simple architectures can be constructed through the usage of a number of patterns.

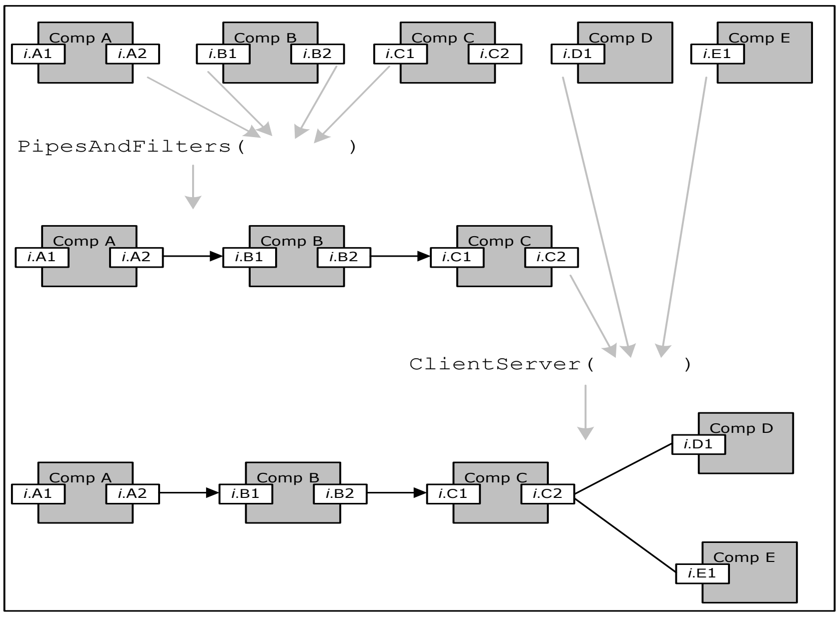

Fig. 2. A simple architecture assembled from a number of components using two pattern templates: *PipesAndFilters* and *ClientServer*

3.3 Formal Syntax for Capturing Meta-information

As discussed in section 2, there is more to architecture than the structural aspects of the system. Issues such as component implementation cost/benefit, design decisions, versions, quality attributes, etc. have not been the focus of most existing ADLs. ADLs such as ADLARS [4] and few others allow the addition of free textual comments to the architecture description using standard commenting syntax similar to that used in programming languages (e.g. through the usage of “/*”, “//”, etc.). This, however, proves to be problematic if CASE tools are to be used to analyze or produce useful documentation from the free textual comments.

One of the challenges with formalizing the syntax for capturing the meta-information is in deciding on the information to be captured in the architecture description. Although there is some information that is usually captured in most architecture documentations (e.g. design decisions, quality attributes, etc.) some other information may vary from one domain to the other and from one enterprise to another (depending on the nature of the domain, the structure of the enterprise, etc.).

In ALI, a special syntax has been introduced to allow for creating *meta types.* Different meta types can be created within a system to act as packages of information (quality attributes, versions, design decisions) which could be attached to different architectural structures throughout the system description.

3.4 Linking the Feature and Architecture Spaces

As Feature Models [18] are built to capture end-users' and stake-holders' concerns and architectures are designed from technical and business perspectives, a gap exists between the two spaces. This gap introduces a number of challenges including: feature (requirements) traceability into the architecture; the ability to verify variability implementation (in Software Product Lines), etc.

ALI attempts at bridging this gap by allowing the architect to link directly the architectural structures to the feature model. Within ALI, it is possible to relate components, connectors, patterns etc. in an architecture description to features in the feature model using first order logic. This permits the capture of complex relationships that might arise between the two spaces in real-life systems.

ALI has adopted and enhanced this concept from ADLARS [4] which was the first ADL to introduce support for linking the feature space to architectural components.

4 Discussion and Future Work

In this paper we have discussed the main issues that might be restricting most current ADLs to small-scale case-studies rather than real-life industrial applications. Restrictive syntax/structure, lack of tool support, and single view presentation are among the limitations discussed.

ALI was created with these limitations in mind and was designed to provide a blend between flexibility and formalism. While flexibility gives freedom for the architect during the design process, formalism allows for architecture analysis and potential automation using proper CASE tool support (e.g. on-the-fly architecture documentation, code generation, etc.).

This paper has focused on the concepts behind ALI. Further information about the ALI notation can be found in [19].

ALI adopts a flexible model for its graphical notation. The textual notation serves as a central database of the architecture description. CASE tools use this information as the source to derive the different relevant architectural views (which can be customized using CASE tools). This model will help alleviate the problem of mismatches among multiple views of the system when maintained separately.

As different architects in different domains (e.g. IS, Telecom, Grid, etc.) would be more comfortable drawing or representing architectures using their own set of symbols/figures (e.g. a cylinder to show a database rather than the standard box of ADLs, etc.), ALI allows for replacing boxes in the graphical notation with any figure the architect chooses as long as interfaces are displayed and labeled properly on that figure. As a comparison between the two approaches (boxes vs figures to represent components), the problem with boxes is that all boxes look basically alike, so it would be relatively difficult to identify and locate a component in a large architecture. On the other hand, the problem with having different images for different components is that, with a large number of component types, the architecture may appear unduly cluttered. So, whether to use boxes or images is left to the architect to decide upon based on the nature and size of the system in any particular project.

As for future work, two major issues top the list for the work on the ALI project:

– *Tool support*: while the work on a toolset for ALI is in progress (using the ADLARS toolset as a starting point), the plan is to make the ALI toolset (and the notation) an open source project. In this way the notation and the toolset will, it is hoped, benefit from a broad range of contributions, both from industry and academia.
– *Providing "round-trip" to code*: the ability to go from architecture to code and back has always been an appealing concept for people working in industry. Work on Model Driven Architectures (MDA) is one successful example of communities working on code generation from architecture specification. In ALI, the possibility of attaching code to components (and glue code to connectors) will be studied. This, if found feasible, will potentially allow for automated generation of substantial parts of the system implementation.

Finally, as the major idea behind ALI is to bridge the gap between industry and academia in the field of ADLs, devising a proper "roll-out" plan for the adoption of ALI in industrial pilot projects (in the first instance) will be considered. Once experience is gained with the language in industrial settings, the aim is to have libraries of meta types, interface types, connector types, etc. for each application domain which architects could then use off-the-shelf.

Acknowledgments. We would like to thank Felix Bachmann, Senior Member of Technical Staff at the Software Engineering Institute, Carnegie Mellon University as well as the Product Line Practice PLP group for their valuable feedback during the initial stage of this work. We would also like to thank David Garlan, Professor, School of Computer Science, Carnegie Mellon University, for his constructive comments on the notation when it started taking shape a few years ago.

References

1. P. Clements, R. Kazman, and M. Klein, Evaluating Software Architecture: Methods and Case Studies. 2002: SEI series in software engineering. Addison-Wesley.
2. E. Woods and R. Hilliard, WICSA 5 Working Group Report "Architecture Description Languages in Practice". November 2005.
3. R.van Ommering, F. van der Linden, J. Kramer, and J. Magee, The Koala Component Model for Consumer Electronics Software. IEEE Computer, March 2000: p. 78-85.
4. R. Bashroush, T.J. Brown, I. Spence, and P. Kilpatrick. ADLARS: An Architecture Description Language for Software Product Lines. Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop. April 2005. Greenbelt, Maryland, USA.
5. G. Booch, I. Jacobson, and J. Rumbaugh, The Unified Modeling Language User Guide. 1998: Addison-Wesley.
6. OMG, UML 2.0 Specification. October 2004, http://www.uml.org.
7. D. Garlan, R. Monroe, and D. Wile, Acme: Architectural Description of Component-Based Systems, in Foundations of Component-Based Systems, G.T. Leavens and M. Sitaraman, Editors. 2000, Cambridge University Press. p. 47-68.
8. D.C. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. Proceedings of DIMACS Partial Order Methods Workshop IV. July 1996. Princeton University.
9. R. Allen, A Formal Approach to Software Architecture. 1997, CMU: PhD Thesis.
10. C. Hofmeister, R. Nord, and D. Soni, Applied Software Architecture. 2000, Boston: Addison-Wesley.
11. P. Kruchten, Architectural Blueprints - The "4+1" View Model of Software Architecture. IEEE Software, November 1995. 12 (6): p. 42-50.
12. N. Rozanski and E. Woods, Software Systems Architecture. 2005: Addison Wesley.
13. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, Documenting Software Architectures. SEI Series on Software Engineering. 2002: Addison Wiesley Longman.
14. Smeda, M. Oussalah, and T. Khammaci. Mapping ADLs into UML 2 Using a Meta ADL. Proceedings of The 5th IEEE/IFIP Working International Conference on Software Architecture. November 2005. Pittsburgh, USA.
15. T.J. Brown, R. Gawley, R. Bashroush, I. Spence, P. Kilpatrick, and C. Gillan. Weaving Behavior into Feature Models for Embedded System Families. Proceedings of the 10th International Software Product Line Conference SPLC 2006 [to appear]. August 2006. Baltimore, Maryland, USA.
16. T.J. Brown, R. Bashroush, C. Gillan, I. Spence, and P. Kilpatrick. Feature Guided Architecture Development for Embedded System Families. Proceedings of the 5th IEEE Working Conference on Software Architecture WICSA-5. November 2005. Pittsburgh, PA, USA.
17. The Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator. https://javacc.dev.java.net/.
18. K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Patterson, Feature Oriented Domain Analysis (FODA) feasibility study. 1990, Software Engineering Institute, Carnegie Mellon University.
19. R. Bashroush, T.J. Brown, I. Spence, and P. Kilpatrick. Flexible Component-Based Architecture Description using ALI. Submitted to the 13th IEEE Asia Pacific Software Engineering Conference (APSEC06). December 2006. Bangalore, India.