# Expressiveness in Architecture Description Languages

Rich Hilliard
Integrated Systems and Internet Solutions, Inc.
rh@isis2000.com

Tim Rice
Integrated Systems and Internet Solutions, Inc.
tbrice@isis2000.com

**Abstract**

This paper explores some issues in the expressiveness of Architecture Description Languages (ADLs) based on our work architecting large, software-intensive systems for command and control and related domains. We briefly outline and motivate several cases where current ADLs lack architecturally useful forms of expression and suggest approaches to addressing some of these cases.

## 1 Architectural Description

Architects don't make architectures, they make *representations* of architectures. *Architectural description* – the means to record architectural information in a form such that it may be used (communicated, analyzed, manipulated) and re-used – is central to the emerging practice of software systems architecture.

Interest in architectural description has taken various forms. One active area of current research is *architecture description languages* (ADLs) as witnessed by:

- a proliferation of ADLs, such as: ACME, Darwin, LILE-Anna, MetaH, QAD, Rapide, UniCON, and Wright; and investigations into exploiting other notations (e.g., Unified Modeling Language) for architectural use [18];
- efforts to categorize, compare and evaluate ADLs: [3, 15, 21]; and,
- attempts to define requirements and desirable features for ADLs [7, 14, 19].

In another effort, the IEEE has undertaken the development of a *Recommended Practice for Architectural Description* [9] to reflect current practices in an ADL-independent manner and provide a foundation for the evolution of the field.

The very notion of representation presumes some conventions for encoding information. Current ADLs take a linguistic approach to an area which has been the province of informal diagrams. The linguistic approach is usually premised on a textual model, perhaps because ADLs have been developed by analogy with programming languages and module interconnection languages.

These ADLs are oriented toward capturing the major structural constituents of a system and their interconnections. Within the field called Software Architecture, the *de facto* vocabulary for describing architectures includes *components* and *connectors*.

In this paper, we outline and motivate other aspects of architectural expression, which we have encountered in the roles of architect [5] or architectural evaluator [8], and which are supported to varying degrees by current ADLs. The purpose of the examples in section 2 is to motivate discussion with ADL developers.

**Scope** There are many potential uses of architectural descriptions (ADs), and thus, of ADLs. These range from creation, analysis, maintenance, and evaluation, through system synthesis (the composition of actual systems from ADL descriptions).

In this paper, we focus on description and communication aspects of AD usage. For large projects, these are the most common and most critical uses of architectural information, and therefore the most critical roles an ADL may play. In such projects, architectural descriptions are needed to communicate and share information among clients and other stakeholders, designers, and maintainers, and for the analysis of architecture descriptions, relative to stakeholder concerns.

We do not address synthesis issues in this paper.

## 2 Can Your ADL Do This?

In the remainder of the paper, we briefly outline some expressive challenges for current ADLs; each has been motivated by one or more actual cases.

### 2.1 Multiple Viewpoints

The notion of multiple viewpoints – a virtual holy grail in many parts of software engineering and computer science – has not fared as well in Software Architecture. Multiple viewpoints provide a way to separate concerns within a representation, and thereby manage descriptive complexity. This is as useful in architecture as in those fields (such as requirements engineering and design) where multiple viewpoints are commonly used.

Although multiple viewpoints are typically advocated by industrial methods (e.g., [5, 11]), many ADLs do not sup-

port multiple viewpoints as they are found, for example, in modeling languages such as UML.

ADLs, for the most part, emphasize what might be called an implicit, structuralist viewpoint (as in the ontology of components and connectors).

The IEEE *Recommended Practice for Architectural Description* attempts to define a framework for declaring viewpoints and populating them to develop meaningful, multiple views of an architecture.

### 2.2 Closed-World

Architectural decision-making takes place at the juncture of the problem and solution spaces – between client's needs for the system, and possible design approaches [17]. Frequently, the architect must feed back the implications of a design alternative to the client, possibly in the form of trade-offs among requirements. Within an architectural description, one would like to be able to clearly delineate the "dependent" and "independent variables" of an architecture. For example, it would be nice to be able to delineate:

- decisions made by the client (these independent variables are usually called "requirements");
- "givens" of the environment (e.g., the laws of physics or of another domain) which are outside the control of the architect but may influence the result; and
- decisions made by the architect in response to the client needs and environmental influences (dependent variables).

It would also be useful to delineate which decisions between client and architect are negotiable and record their resolutions at any point in time.

In our work, the needs of the client and other stakeholders are documented separately from the architectural description. Traceability relations are maintained between elements of the AD and the stakeholders' needs to insure coverage, consistency, and satisfaction.

### 2.3 Capturing Architect's Intent

Much of the work of the architect is to articulate various system characteristics. These characteristics may be structural, stylistic, interactional, etc. With regard to these various system characteristics, the architect usually needs to express at least three degrees of intent (or "force"), in relation to subsequent design activity. The architect needs to capture *commitments* – decisions that a designer is not at liberty to change; *obligations* – lower-level decisions a designer must address; and *freedoms* – those things left to the implementation. (See [2] on commitments and obligations, see [12] on freedoms.)

Current ADLs have an implicit, declarative force and appear not to be designed to capture this range of intents. Acme has partial support for two forms of obligations: *ensures relationships* and their desired logical consequences in terms of *derives relationships* [16].

In our work previously, we adopted a "MIL-SPEC" approach to signalling the architect's intent, using the terms **will**, **shall**, and **may** to signal commitments, obligations, and freedoms, respectively [4].

More sophisticated forms of expression could be attained from examination of work in modal and deontic logics, and their recent application to knowledge representation.

### 2.4 First-Class Constraints

While components and connectors are widely accepted constructs in Software Architecture and enjoy a rough consensus on their meanings, constraints, though frequently discussed, are not yet understood in a common fashion. Where present in ADLs, constraints are usually modeled as predicates applied to entities of interest. E.g.,

$$\mathsf{WrittenInAda95}(\mathit{theServer})$$

The example is intended to suggest a constraint on the possible implementations of a component ($\mathit{theServer}$). (Note that this example begs the question posed in 2.3 with regard to intent – is it an obligation or commitment being expressed here?)

Rapide provides a rich vocabulary of constraints over sets of events [13]. Other ADLs provide partial solutions, often oriented toward the target domain, or underlying implementation model. What is lacking is a general model of constraints.

We have found the following model to be useful. An *architectural constraint* is characterized by:

**name** : a unique name for the constraint. By naming constraints they may be referred to, more readily combined with others and reused.

**constraint type** : the type defines the permissible operations and logical connectives which may apply to the constraint.

**expression** : An inscription which states the constraint (e.g., *number of simultaneous clients must be less than or equal to 3*). The expression should conform to the constraint type prescription.

**target set** : The set of elements (e.g., component and connection instances) to which the constraint applies.

**source set** : The set of elements from which the constraint originates, such as when one architectural decision affects others. Identifying the source provides a basis by which to propagate constraints and retract constraints when decisions about the source (which induced the constraint) are changed.

A source or target set may be all entities of a particular type; a source or target could be an entire architectural view (which in turn consists of a collection of components or connections). By adding the possibility that the source is unknown, one can handle the "degenerate" case of constraint as predicate above. This provides a way to express constraints which are "extra-architectural" such as those arising from needs, requirements, etc.

### 2.5 Computational Independence

Many ADLs, particularly those developed for domain-specific software architectures, have a built-in, implicit underlying model of computation (for discussion, see [1, pp. 7–8]).

In terms of economy of expression, this is quite efficient for application-level development. However, there are cases where the architect would like to defer the choice of the computational model. E.g., in product line development, one might select the computational model as a part of the design of individual products within the product line. Consider a product line for a MapServer intended to operate in a client-server environment, or 'stand-alone' as a library

linked into a single application, or distributed for replication purposes wrapped with an Object Request Broker. It should be possible to state certain elements of the architecture independent of these decisions. However, this stretches the capabilities of existing structure-based ADLs.

Inverardi and Wolf [10] suggest "an operational semantic formalism ... based on a more flexible, relatively neutral computational model," and investigate the application of the Chemical Abstract Machine in this regard. A general solution could involve the analogue of a meta-object protocol for computational models – reifying the major design constituents of a computing model [20] (certainly overkill for architectural description!).

### 2.6 Quantification

Most ADLs embody an ontology of types and instances, reflecting perhaps their programming language (or module interconnection language) heritage. While this works for small systems, for larger projects a wider range of facilities may be warranted.

First-order logic quantifiers ($\forall$, $\exists$) provide increased expressivity for some cases. Other quantifier-like decisions, e.g., *the server must handle 1 to 10 simultaneous clients*, may be captured via constraints. Whether more sophisticated quantification mechanisms (such as those found in description logics for knowledge representation) are useful is an open issue.

In a proposal for Dynamic Acme, [16] introduces *optional* and *multiple* elements, which fulfill some of the roles of quantifiers. UML is also an instructive example.

### 2.7 Architectural Patterns and Styles

Much current work in software design and architecture is centered on the articulation of patterns and styles. There are various ways to think of styles – as languages in themselves, as theories in their own right, etc. [6]. Similar considerations arise with patterns. One way to think of both patterns and styles is as idioms (or, recurring fragments) over some vocabulary. Design patterns – in the sense of the book of the same name – are defined *over* the vocabulary of object-oriented programming constructs (classes, methods, interfaces, inheritance relations). To support pattern and style making, ADLs should be sufficiently modular to allow expression of individual entities. Current ADLs approach this for components, but not for connections and constraints.

## 3 Conclusion

If there is an underlying thread to the cases above, it is this: architecting is decision making and the issues of expressiveness can be considered as documenting *when* certain decisions are made, *by whom* (client, architect, developers), *for whom* (client, user other stakeholders), and *in a way* that may be communicated to others. The consequences of this for actual ADLs is that expressiveness is improved by having explicit constructs for many aspects which have previously been implicit. Good language design suggests making such constructs necessary only when needed – that the defaults match current intuitions, while allowing overt expression when necessary.

## References


[1] Robert J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997. Distributed as CMU-CS-97-144.

[2] A. Burns and M. Lister. A framework for building dependable systems. *The Computer Journal*, 34(2), 1991.

[3] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the Eighth International Workshop on Software Specification and Design*. IEEE Computer Society Press, 1996.

[4] DII–AF Chief Architects' Office. *The Air Force's Command and Control System Target Architecture version 1.0*, 1998. http://www.esc-dii.hanscom.af.mil/afdiin/Chief_Architect/.

[5] David E. Emery, Richard F. Hilliard, and Timothy B. Rice. Experiences applying a practical architectural method. In Alfred Strohmeier, editor, *Reliable Software Technologies – Ada-Europe '96*, number 1088 in Lecture Notes in Computer Science. Springer, 1996.

[6] David Garlan. Research directions in software architecture. *ACM Computing Surveys*, 27(2):257–261, 1995.

[7] Richard F. Hilliard. Representing software systems architectures or, components, connections and (why not?) first-class constraints and views. In *Joint Proceedings of the SIGSOFT '96 Workshops*, 1996.

[8] Richard F. Hilliard, Michael J. Kurland, Steven D. Litvintchouk, Timothy B. Rice, and Stephen C. Schwarm. Architecture quality assessment. `http://katanga.mitre.org/ose/arch/AQAPage.html`.

[9] IEEE Architecture Working Group. *Recommended Practice for Architectural Description (draft version 3.0)*, July 1998. `http://www.pithecanthropus.com/~awg/`.

[10] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[11] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 28(11):42–50, November 1995.

[12] Phillip E. London and Martin Feather. Implementing specification freedoms. *Science of Computer Programming*, 2:91–131, 1982.

[13] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[14] David C. Luckham, James Vera, and Sigurd Meldal. Three concepts of system architecture. Technical Report CSL-TR-95-674, Stanford University, July 1995.

[15] Nenad Medvidovic. A classification and comparison framework for software architecture description languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, February 1997.

[16] R. Monroe, D. Garlan, and D. Wile. Acme StrawManual. Available from the ACME Web site at CMU.

[17] Eberhard Rechtin and Mark Maier. *The art of systems architecting*. CRC Press, 1996.

[18] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles, and David S. Rosenblum. Integrating architecture description languages with a standard design method. Presented at the Second EDCS Cross Cluster Meeting in Austin, Texas.

[19] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an emerging discipline*. Prentice Hall, 1996.

[20] Patrick Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, 1994.

[21] Steve Vestal. A cursory overview and comparison of four architecture description languages. Unpublished, 1993.