INTRODUCTION TO SOFTWARE ARCHITECTURE DESCRIPTION LANGUAGES

BY PANKAJ KAMTHAN

1. INTRODUCTION

I'd like to start with a diagram. It's a bunch of shapes connected by lines. Now I will say some impressive words: synchronized incremental digital integrated dynamic e-commerce space. Any questions?
— Dilbert[1], February 27, 2000

To **describe** anything, including a software system's architecture, there is a need for a **language**. This document explores software architecture description languages (ADLs).

The impetus for ADLs came from the need for both **humans and machines** to analyze software architecture descriptions for properties such as **correctness, completeness, and consistency**.

CONVENTION/NOTATION

For the sake of this document, the terms **'architecture description language'**, **'architecture specification language'**, and **'architecture modeling language'** are **synonymous**, and therefore interchangeable.

2. DEFINITION OF SOFTWARE ARCHITECTURE DESCRIPTION LANGUAGE

There are a number of definitions of ADL, including the following:

GENERAL DEFINITION

Definition [Architecture Description Language (ADL)] [SEI/CMU]. A language (graphical, textual, or both) for describing a software system in terms of its architectural elements and the relationships among them.

[1] URL: http://www.dilbert.com/ .

OBSERVATIONS

The definition suggests that an ADL can have multiple modalities.

SPECIFIC DEFINITION

Definition [Architecture Description Language (ADL)] [Medvidović, Dashofy, Taylor, 2007]. A modeling notation that provides facilities for capturing a software system's **components** (that is, computational elements), **connectors** (that is, interaction elements), and **configurations** (that is, overall structure).

OBSERVATIONS

- The definition acknowledges that architecture of a software system can be modeled.
- The definition has been used to compare and evaluate a number of ADLs [Medvidović, Taylor, 2000].
- The definition is used by **xADL**.

2.1. ADL AS A LANGUAGE

Usually, a 'conventional' ADL has **formal syntax and formal semantics**. This distinguishes an instance of an ADL from, say, a **boxes-and-lines diagram**.

For example, **System Engineering Modeling Language (SysML)** is such an ADL [Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord, Stafford, 2010, Appendix A].

SysML is a general-purpose systems-modeling language intended to support a broad range of analysis and design activities for **systems-engineering applications**. SysML is a standard maintained by the **Object Management Group (OMG)** and was developed by OMG in cooperation with the **International Council on Systems Engineering (INCOSE)**. SysML was developed as a **profile** of the **Unified Modeling Language (UML)**.

3. HISTORY AND EVOLUTION OF SOFTWARE ARCHITECTURE DESCRIPTION LANGUAGES

The **history** of ADLs has been traced in [Dashofy, Hoek, Taylor, 2005; Kruchten, Obbink, Stafford, 2006; Medvidović, Dashofy, Taylor, 2007; Qin, Xing, Zheng, 2009, Chapter 4; Babar, Dingsøyr, Lago, Vliet, 2009, Section 3.3.2; Pandey, 2010].

The 'era' of ADLs has seen **two generations**, one **before year 2000** and the other **after 2000** [Medvidović, Dashofy, Taylor, 2007]. The ADLs in the first generation were **domain-specific**, **formal**, and were **aimed essentially only at technical stakeholders**. The ADLs in the second generation are **broader in their support for different domains**, are **not necessarily formal**, and **aim to accommodate non-technical stakeholders**.

The following table lists some initial ADLs and their sources [Pandey, 2010]:

ADL	Source
DSSA ADAGE	IBM
Aesop	Carnegie Mellon University
C2	University of California, Irvine
Darwin	Imperial College, London
Meta-H	Honeywell
Rapide	Stanford University
SADL	SRI
UniCon	Carnegie Mellon University
Wright	Carnegie Mellon University

There are a number of **surveys**, including **comparisons**, of ADLs [Clements, 1996; Medvidović, Taylor, 2000; Chen, Li, Yi, Liu, Tang, Yang, 2010; Pandey, 2010]. The difference of time among the surveys is **relevant**. In particular, older surveys may **not** consider newer ADLs.

Appendix A demonstrates certain ADLs.

4. UNDERSTANDING SOFTWARE ARCHITECTURE DESCRIPTION LANGUAGES

Figure 1 shows ADL relative to other elements in software architecture description.

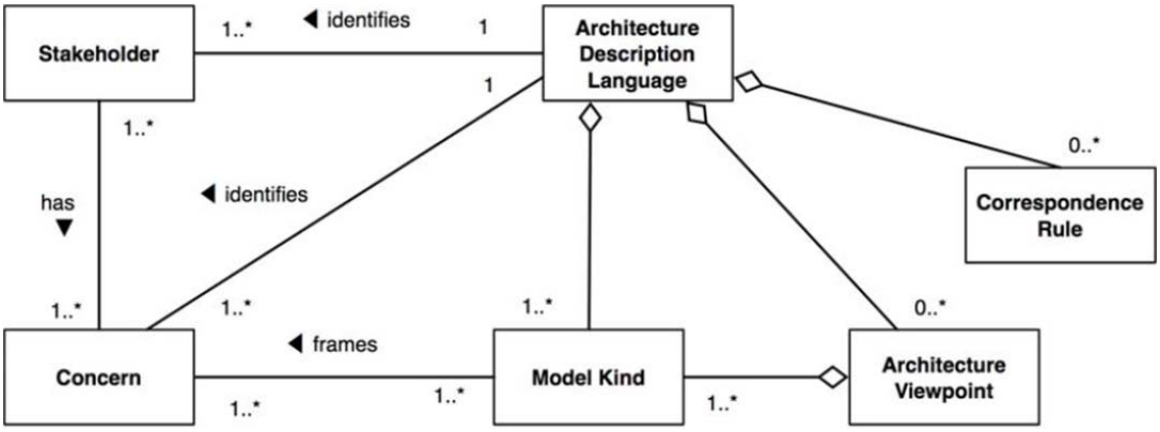


Figure 1. The conceptual model of architectural description language. (Source: [ISO/IEC/IEEE, 2011].)

4.1. A PERSPECTIVE OF SOFTWARE ARCHITECTURE DESCRIPTION LANGUAGES

There are three **lampposts** in the **software ecosystem** that provide the broad perspective on ADLs and their role in supporting software development [Medvidović, Dashofy, Taylor, 2007]:

1. Domain of Application
2. Business Concerns
3. Technology

The **lamppost metaphor** is used to exemplify the need for **adequate illumination at all major concerns** for an ADL to be successful in its goal of describing software architecture.

The criteria for selecting of an ADL should include the extent to which an ADL strikes a **balance** in supporting each lamppost. (The use of the term 'balance' automatically means that there is **no 'perfect' ADL**.) This means that the ADL should fall into the **intersection** of the lampposts depicted as sets in the Venn Diagram shown in Figure 2.

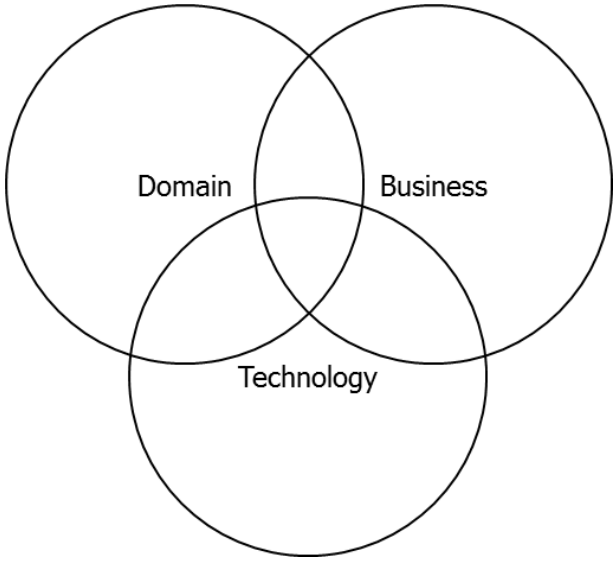


Figure 2. The three lampposts of an ADL: domain of application, business concerns, and technology. (Source: [Medvidović, Dashofy, Taylor, 2007].)

DOMAIN LAMPPOST

This lamppost is concerned with issues related to **application domain(s) (or problem domain(s))**, even if the ADL aims to be **application domain-independent**.

For example, for **MetaH** is an ADL that has been used for describing the software architecture of a **flight control system**, the application domain of which is **avionics**.

An ADL initially invented for a specific application domain may, over time, be found useful for other application domains.

BUSINESS LAMPPOST

This lamppost is concerned with issues related to the business of the organization: positioning with respect to the **long-term organizational vision**, and alignment with the **short-term product strategy**.

For example, features that have proven to be absolutely or relatively successful in the market, and stakeholder perceptions of the product, are of interest.

TECHNOLOGY LAMPPOST

This lamppost is concerned with issues related to **computer science** and **technical aspects of software engineering**: linguistics (syntax, semantics, pragmatics), representation, reasoning, transformation (onto implementation), and so on.

The **first generation** of ADLs focused essentially, and often **exclusively**, on this lamppost.

5. CHARACTERISTICS OF A SOFTWARE ARCHITECTURE DESCRIPTION LANGUAGE

From a language perspective, an ADL lies somewhere **between** a **modeling language** and a **programming language**.

There are a number of characteristics of an ADL [Clements, 1996; Garlan, Shaw, 1996; Medvidović, Taylor, 2000; Lago, Malavolta, Muccini, Pelliccione, Tang, 2015], including the following:

- The ability for **abstraction**.
- The ability to support **different domains**.
- The ability to represent **different views**.
- The ability to represent different **architectural styles (or patterns)**.
- The ability to present **multiple modalities**.
- The support for **composition**.
- The support for **heterogeneity**.
- The support for **verifiability**.
- The support for **typing and type checking**.
- The support for **extensibility**.

In a software architecture design process, there is involvement of both **technical as well as non-technical stakeholders**. It is therefore important that an ADL be a language that is **common** to **designers, implementers, and even users**.

5.1. XML AND ADL

Usually, a 'conventional' ADL has its own unique syntax. This can create an issue of **interoperability** across processing tools.

The **Extensible Markup Language (XML)** can be, and has been, used for representing the **syntax** of a number of ADLs [Dashofy, Hoek, Taylor, 2001; Dashofy, Hoek, Taylor, 2005; Medvidović, Dashofy, Taylor, 2007].

For example, **xADL**[2] is such an ADL. Furthermore, xADL, specifically xADL 2.0, is supported in **ArchStudio**[3], an open source development environment for software architecture.

6. UML FOR SOFTWARE ARCHITECTURE DESCRIPTION

The support for software architecture description in the **Unified Modeling Language (UML)** has **evolved** with the evolution of UML. In particular, the support for software architecture description in UML 2.x is an **improvement** over that in UML 1.x.

The suitability of **UML 1.4 versus UML 2.0** as languages to describe software architectures has been analyzed in [Pérez-Martínez, Sierra-Alonso, 2004].

There are a number of **advantages and disadvantages** of UML for describing software architectures [Woods, Hilliard, 2005; Bharathi, Sridharan, 2009; Pandey, 2010].

6.1. ADLS IN CONTEXT

In a large-scale survey (of multiple practitioners from multiple different companies in multiple countries), the **usefulness and utility of ADLs for industrial software development** is examined [Malavolta, Lago, Muccini, Pelliccione, Tang, 2013]. Figure 3 shows that the **use of UML** ranks significantly high compared to other ADLs, a trend that is not likely to change in the distant future.

[2] URL: http://www.isr.uci.edu/projects/xarchuci/ .
[3] URL: http://www.isr.uci.edu/projects/archstudio/ .

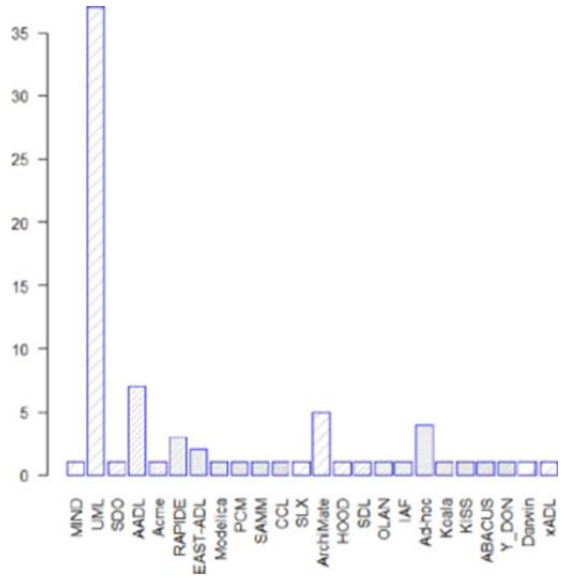


Figure 3. The results of a survey of the usefulness and utility of ADLs for industrial software development. (Source: [Malavolta, Lago, Muccini, Pelliccione, Tang, 2013].)

6.2. UML EXTENSION MECHANISMS

UML is aimed to be **general-purpose**. However, regardless of the version, it may **not be suitable** as-is for **all** modeling purposes. Therefore, UML is designed to propose **extensions** and support those extensions.

Indeed, the use of a **UML extension mechanism** can make a UML-based model **expressive**.

There are **three** mechanisms for extending UML diagram types [Booch, Rumbaugh, Jacobson, 2005]:

1. **Stereotype.** A stereotype allows the creation of new kinds of building blocks similar to existing ones but specific to a problem. A stereotype can be **standard**, that is, provided by UML, or **non-standard**. A stereotype is graphically rendered as a name enclosed by guillemets (« »), and usually placed **above** the name of another element. For example, «`include`» is a standard stereotype.

2. **Tagged Value.** A tagged value is a property of a stereotype that allows creation of new information in an element bearing that stereotype. A tagged value is graphically rendered as a string, and enclosed in **braces (or curly brackets)** in the form `{tag = value}` **within** an element. A comma is used to separate multiple tagged values. For example, `{author="John Smith", version = 1.0}`.

3. **Constraint.** A constraint is a specification of the **semantics** of a UML element, allowing addition of new rules or modification of existing rules. A graphical constraint is rendered as a string enclosed by braces, and **placed near** the associated element, or connected to that element or elements by a **dependency relationship**. A constraint specifies a condition that must be true. For example, `{AccountBalance >= 0}`.

6.3. EXAMPLES OF UML FOR SOFTWARE ARCHITECTURE DESCRIPTION

Figure 4 shows the use of **UML Component Diagram** for illustrating **one specific view** of the software architecture of an **order processing system**.

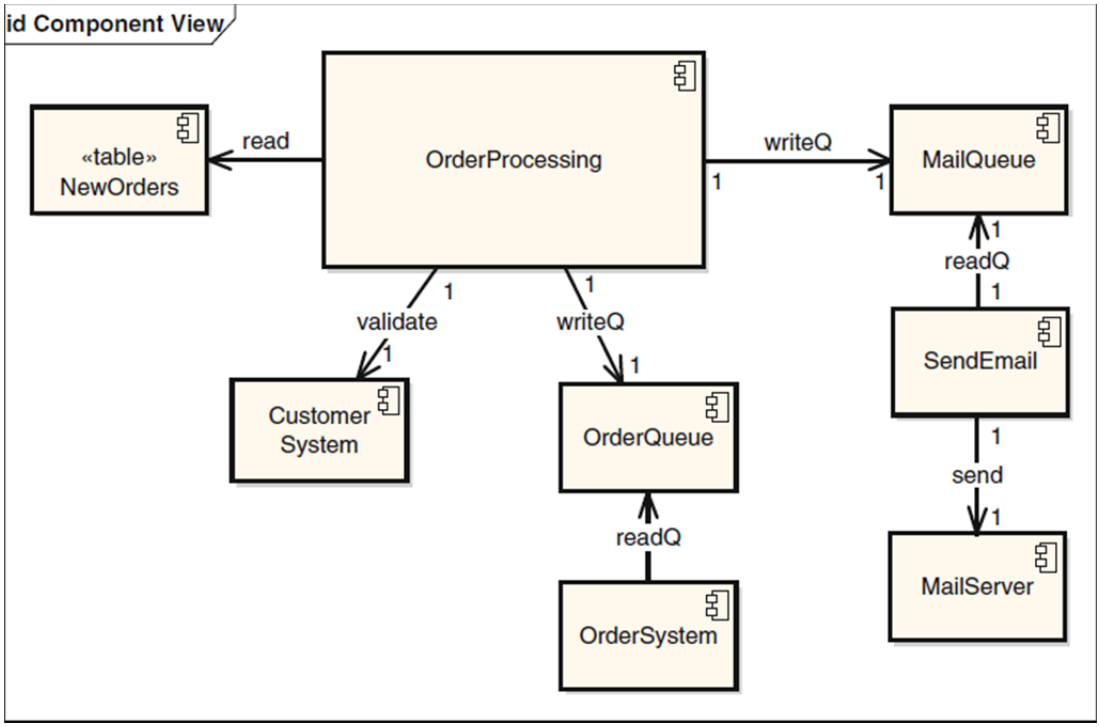


Figure 4. A UML Component Diagram for Order Processing. (Source: [Gorton, 2011, Figure 8.1].)

6.4. UML AS AN ADL

UML is a successor to many 'conventional' ADLs. However, UML, by many, is **not** considered as a 'pure' ADL [Kruchten, Obbink, Stafford, 2006; Pandey, 2010].

There are a number of **advantages and disadvantages** of 'conventional' ADLs over UML for describing software architecture [Woods, Hilliard, 2005; Shaw, Clements, 2006; Aldini, Bernardo, Corradini, 2010, Chapter 4; Pandey, 2010; Vogel, Arnold, Chughtai, Kehrer, 2011, Section 6.6.4].

6.5. ADLS IN CONTEXT: REPRISE

ADVANTAGES OF A 'CONVENTIONAL' ADL

The advantages of a 'conventional' ADL include the following:

- It **reduces ambiguity** in software architecture descriptions.
- It **enables automation**.

DISADVANTAGES OF A 'CONVENTIONAL' ADL

The disadvantages of a 'conventional' ADL include one or more of the following:

- It is **not a standard**.
- It is **restricted** to the application domain[4] or to a specific problem area[5] for which it was invented. (It is not broadly applicable.)
- It **ignores aspects** that are considered significant (by software architects) in software architecture practice (in industry).
- It does **not support multiple views**.
- It is **not be accessible** to non-technical stakeholders. (It uses mathematics and mathematical notation that is usually unfamiliar to non-technical stakeholders.)
- It has its own **unique syntax** (and therefore associated learnability and interoperability issues).

[4] For example, such domain could be avionics, automotives, or systems (hardware).
[5] For example, such problem could be deadlock detection in the software architecture of a concurrent system, which is the focus of Wright ADL.

- If based on text, its **syntax may be considered verbose**. (In case of most software systems of interest, considerable amount of text is required to describe the software architecture.)

- It **lacks broad and mature tool support**. In the event that the tool support exists, the tool does not integrate with other tools used regularly by software architects, that is, the tool is not interoperable.

ADVANTAGES OF UML FOR SOFTWARE ARCHITECTURE DESCRIPTION

The advantages of UML as an ADL include the following:

- It is a **standard**.

- It has a bourgeoning **community**.

- It addresses **business and technology lampposts** of an ADL [Medvidović, Dashofy, Taylor, 2007].

- It continues to **evolve**.

- It is **graphical**.

- It is **general** in its applicability.

- It is **extensible**.

- It can, upon basic training, be **accessible to non-technical stakeholders**. (It is therefore inclusive.)

- It is increasingly being acknowledged that there needs to be a **balance** between **expressiveness and understandability** of software architecture description. A commitment to UML for describing software architecture is a manifestation of such a balance [Christensen, Corry, Hansen, 2007].

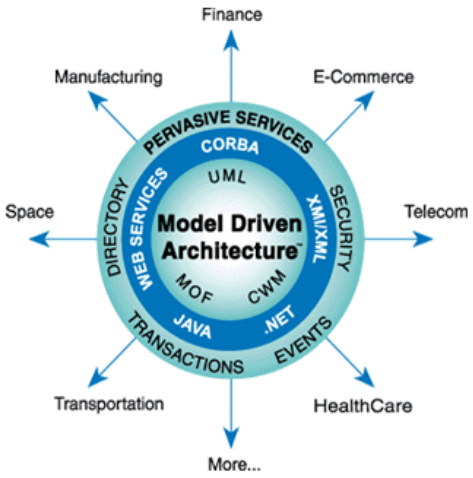
- It supports **multiple views**, a necessity for understanding software architecture and therefore a crucial consideration for an ADL [Medvidović, Dashofy, Taylor, 2007]. For example, UML can express the **4+1 View Model of Software Architecture** [Kruchten, 1995].

- It has **broad and mature tool support**.

- It has explicit **support in certain software development methodologies**, such as the **Rational Unified Process (RUP)**.

- It is **accepted in industry** [Navarčík, 2005]. The results of a survey on software architecture reviews [Babar, Gorton, 2009] indicate that UML in used for software architecture description in many organizations.

- It is part of the **Model-Driven Architecture (MDA)** [Kleppe, Warmer, Bast, 2003; Mellor, Scott, Uhl, Weise, 2004; Brambilla, Cabot, Wimmer, 2012; Brambilla, Fraternali, 2015].



(Source: OMG.)

- It can be **used in other activities and artifacts** in a software development process. This supports learnability by relying on a software engineer's familiarity with the language.

DISADVANTAGES OF UML FOR SOFTWARE ARCHITECTURE DESCRIPTION

The disadvantages of UML as an ADL include one or more of the following:

- It is **not necessarily suited** for **non-object-oriented** paradigms.

- It **does not have constructs to model all possible architectural concerns**. For example, there is no explicit support in UML for expressing product variants or for depicting the temporal evolution of a system's architecture [Medvidović, Dashofy, Taylor, 2007].

- It is **intentionally ambiguous** (so as to be application domain-independent). As a consequence, UML diagram types and symbols can be interpreted in different ways by different stakeholders.

- It **can be extended without any control** (over the extensions). For example, for a given set of concerns, it is possible to have multiple UML profiles, and a 'conventional' UML modeler is supposed to be insensitive towards the semantics of these profiles.

- It **inherits the shortcomings of generality** (or aiming to be **Swiss Army Knife** [Medvidović, Dashofy, Taylor, 2007]). In particular, it may not be suitable for the needs of specific application domains. For example, UML is **not** designed for cloud applications. This has led to extensions of UML, such as the **Cloud Application Modeling Language (CAML)** [Bergmayr, 2016].

- It, excluding certain exceptions and non-standard extensions, **does not have formal semantics**, and therefore a software architecture description based on UML cannot be assured of certain desirable properties. This limitation can be overcome, to a certain extent, by the use of the **Object Constraint Language (OCL)** [Warmer, Kleppe, 2004], although the current support in tools for OCL remains scarce.

- It has **inadequate support for automation**. For example, a software architecture description based on UML cannot be evaluated automatically.

7. LIMITATIONS OF ADLS FOR SOFTWARE ARCHITECTURE DESCRIPTION

There are certain inherent limitations of ADLs (that is, both a 'conventional' ADL and UML) towards software architecture description. They do not have support for the **reasoning** underlying the decisions in a software architecture description, and do not have support for the **implications** of those decisions. In other words, ADLs address the HOW, but not the WHY.

ACKNOWLEDGEMENT

The inclusion of images from external sources is only for non-commercial educational purposes, and their use is hereby acknowledged.

REFERENCES

[Aldini, Bernardo, Corradini, 2010] A Process Algebraic Approach to Software Architecture Design. By A. Aldini, M. Bernardo, F. Corradini. Springer-Verlag. 2010.

[Babar, Dingsøyr, Lago, Vliet, 2009] Software Architecture Knowledge Management: Theory and Practice. By M. A. Babar, T. Dingsøyr, P. Lago, H. van Vliet. Springer-Verlag. 2009.

[Bergmayr, 2016] An Architecture Style for Cloud Application Modeling. By A. Bergmayr. Ph.D. Thesis. Technical University of Vienna. Vienna, Austria. 2016.

[Bharathi, Sridharan, 2009] UML as an Architecture Description Language. By B. Bharathi, D. Sridharan. International Journal of Recent Trends in Engineering. Volume 1. Number 2. 2009. Pages 230-232.

[Booch, Rumbaugh, Jacobson, 2005] The Unified Modeling Language User Guide. By G. Booch, J. Rumbaugh, I. Jacobson. Second Edition. Addison-Wesley. 2005.

[Brambilla, Cabot, Wimmer, 2012] Model-Driven Software Engineering in Practice. By M. Brambilla, J. Cabot, M. Wimmer. Morgan and Claypool. 2012.

[Brambilla, Fraternali, 2015] Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML. By M. Brambilla, P. Fraternali. Morgan Kaufmann. 2015.

[Chen, Li, Yi, Liu, Tang, Yang, 2010] A Ten-Year Survey of Software Architecture. By Y. Chen, X. Li, L. Yi, D. Liu, L. Tang, H. Yang. The 2010 International Conference on Software Engineering and Service Sciences (ICSESS). Beijing, China. July 16-18, 2010.

[Christensen, Corry, Hansen, 2007] An Approach to Software Architecture Description Using UML. By H. B. Christensen, A. Corry, K. M. Hansen. Technical Report. Department of Computer Science. University of Aarhus. Aarhus, Denmark. 2007.

[Clements, 1996] A Survey of Architecture Description Languages. By P. C. Clements. The Eighth International Workshop on Software Specification and Design (IWSSD-8). Schloss Velen, Germany. March 22-23, 1996.

[Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord, Stafford, 2010] Documenting Software Architectures: Views and Beyond. By P. Clements, F. Bachmann,

L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford. Second Edition. Addison-Wesley. 2010.

[Dashofy, Hoek, Taylor, 2001] A Highly-Extensible, XML-Based Architecture Description Language. By E. M. Dashofy, A. van der Hoek, R. N. Taylor. The 2001 Working IFIP Conference on Software Architecture (WICSA 2001). Amsterdam, The Netherlands. August 28-31, 2001.

[Dashofy, Hoek, Taylor, 2005] A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. By E. M. Dashofy, A. van der Hoek, R. N. Taylor. ACM Transactions on Software Engineering and Methodology. Volume 14. Number 2. 2005. Pages 199-245.

[Garlan, Shaw, 1996] Software Architecture: Perspectives on an Emerging Discipline. By D. Garlan, M. Shaw. Prentice-Hall. 1996.

[Gorton, 2011] Essential Software Architecture. By I. Gorton. Springer-Verlag. 2011.

[ISO/IEC/IEEE, 2011] ISO/IEC/IEEE 42010:2011. Systems and Software Engineering -- Architecture Description. The International Organization for Standardization (ISO)/The International Electrotechnical Commission (IEC). 2011.

[Kleppe, Warmer, Bast, 2003] MDA Explained: The Model Driven Architecture™: Practice and Promise. By A. Kleppe, J. Warmer, W. Bast. Addison-Wesley. 2003.

[Kruchten, 1995] The 4+1 View Model of Architecture. By P. B. Kruchten. IEEE Software. Volume 12. Issue 6. 1995. Pages 42-50.

[Kruchten, Obbink, Stafford, 2006] The Past, Present, and Future for Software Architecture. By P. Kruchten, H. Obbink, J. Stafford. IEEE Software. Volume 23. Issue 2. 2006. Pages 22-30.

[Lago, Malavolta, Muccini, Pelliccione, Tang, 2015] The Road Ahead for Architectural Languages. By P. Lago, I. Malavolta, H. Muccini, P. Pelliccione, A. Tang. IEEE Software. Volume 32, Number 1. 2015. Pages 98-105.

[Malavolta, Lago, Muccini, Pelliccione, Tang, 2013] What Industry Needs from Architectural Languages: A Survey. By I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, A. Tang. IEEE Transactions on Software Engineering. Volume 39. Number 6. 2013. Pages 869-891.

[Medvidović, Dashofy, Taylor, 2007] Moving Architectural Description from Under the Technology Lamppost. By N. Medvidović, E. M. Dashofy, R. N. Taylor. Information and Software Technology. Volume 49. Issue 1. 2007. Pages 12-31.

[Medvidović, Taylor, 2000] A Classification and Comparison Framework for Software Architecture Description Languages. By N. Medvidović, R. N. Taylor. IEEE Transactions on Software Engineering. Volume 26. Issue 1. 2000. Pages 70-93.

[Mellor, Scott, Uhl, Weise, 2004] MDA Distilled: Principles of Model-Driven Architecture. By S. J. Mellor, K. Scott, A. Uhl, D. Weise. Addison-Wesley. 2004.

[Navarčík, 2005] Using UML with OCL as ADL. By M. Navarčík. The First Student Research Conference in Informatics and Information Technologies (IIT.SRC 2005). Bratislava, Slovakia. April 27, 2005.

[Pandey, 2010] Architectural Description Languages (ADLs) vs UML: A Review. By R. K. Pandey. ACM SIGSOFT Software Engineering Notes. Volume 35. Issue 3. 2010. 5 Pages.

[Pérez-Martínez, Sierra-Alonso, 2004] UML 1.4 versus UML 2.0 as Languages to Describe Software Architectures. By J. E. Pérez-Martínez, A. Sierra-Alonso. The First European Workshop on Software Architecture (EWSA 2004). St. Andrews, U.K. May 21-22, 2004.

[Qin, Xing, Zheng, 2009] Software Architecture. By Z. Qin, J. Xing, X. Zheng. Springer-Verlag. 2009.

[Shaw, Clements, 2006] The Golden Age of Software Architecture. By M. Shaw, P. Clements. IEEE Software. Volume 23. Issue 2. 2006. Pages 31-39.

[Vogel, Arnold, Chughtai, Kehrer, 2011] Software Architecture: A Comprehensive Framework and Guide for Practitioners. By O. Vogel, I. Arnold, A. Chughtai, T. Kehrer. Springer-Verlag. 2011.

[Warmer, Kleppe, 2004] The Object Constraint Language: Getting Your Models Ready for MDA. By J. Warmer, A. Kleppe. Second Edition. Addison-Wesley. 2004.

[Woods, Hilliard, 2005] WICSA 5 Working Group Report: Architecture Description Languages in Practice. By E. Woods, R. Hilliard. The Fifth IEEE/IFIP Working Conference on Software Architecture (WICSA 5). Pittsburgh, U.S.A. November 9-10, 2005.

APPENDIX A. EXAMPLES OF SOFTWARE ARCHITECTURE DESCRIPTIONS IN ADLS

This Appendix presents some examples of descriptions of certain software architectures using different ADLs [Qin, Xing, Zheng, 2009].

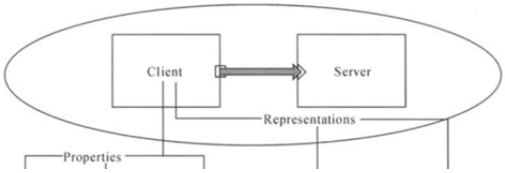
EXAMPLE 1

The following is a client-server system in Darwin:


```
Component Server{
provide p;
}
Component Client{
require r;
}
Component System{
inst
A: Client B:Server
bind
A.r — B.p
}
```

EXAMPLE 2

The following is a client-server system in ACME:




```
System simple_cs ={
   Component client ={
         Port send-request;
         Properties  { Aesop-style : style-id = client-server;
                   UniCon-style : style-id = cs;
                   source-code:external = "CODE-LIB/client.c"}}
   Component server ={
         Port receive-request;
         Properties  { idempotence : boolean = ture;
                   max-concurrent-clients : integer = 1;
                   source-code:external = "CODE-LIB/server.c"}}
   Connector rpc ={
         Roles {caller,callee}
         Properties  { synchronous : boolean  =true
                   max-roles : integer = 2;
                   protocol : Wright = " "}}
   Attachments {
         client.send-request to rpc.caller;
         server.receive-request to rpc.callee}

}
```