Describing the architecture: Creating and Using Architectural Description Languages (ADLs):

What are the attributes and R-forms?
Cognitive Map of 8090

IS Architectures as Strategy
Weill, Ross & Robertson, “Enterprise Architecture as Strategy”

Work Systems Models
S. Alter, “Work System Method”

IS architectural components and development
Mid-Range Modeling

ERP as instantiation of Architectures

Architectural Modeling options

‘Lower level’ architectures
Motivation: as an Enterprise Architects

- What are ADLs?
- Where will you use ADLs?
- How will you use ADLs?
- At what level of abstraction in the EA will you deploy ADLs?
What is an Architectural Description Language (ADL)?: definitions

- **Generic**: “Architectural Description Languages (ADL) describe the structure of a software system at a level of abstraction that is more closest to the intuition of a system designer.” (Arvind W. Kiwelekar, 2010)
- **System engineering community**: sees an ADL as a language and/or a conceptual model to describe and represent system architectures.
- **Software engineering community**: sees an ADL as a computer language to create a description of a software architecture.
- Communicating what to whom?
  - the technical architecture communicates to the software developers
  - functional architectures communicate to the stakeholder/user community.
- There exists no clear consensus on ….their focus and how they “describe”
ADLs are not... (they differ from)

• requirements languages, because ADLs are rooted in the solution space, whereas requirements describe problem spaces.

• programming languages, because ADLs do not bind architectural abstractions to specific point solutions.

• Modeling languages which represent behaviors, where ADLs focus on representation of components. However, there are domain specific modeling languages (DSMLs) that focus on representation of components.

Example ADLs and foci

- Many attempts large variety of foci
  - e.g., Aesop, ArTek, C2, Darwin, LILEANNA, MetaH, Rapide, SADL, UniCON, Weaves, Wright
  - i.e., *Rapide*- general purpose system description language
  - i.e., *Wright*- focuses on modeling connectors

<table>
<thead>
<tr>
<th>ADL</th>
<th>ACME</th>
<th>Aesop</th>
<th>C2</th>
<th>Darwin</th>
<th>MetaH</th>
<th>Rapide</th>
<th>SADL</th>
<th>UniCON</th>
<th>Weaves</th>
<th>Wright</th>
<th>Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural interchange, predominantly at the structural level</td>
<td>Specification of architectures in specific styles</td>
<td>Architectures of highly-distributed, evolvable, and dynamic systems</td>
<td>Architectures of highly-distributed systems whose dynamism is guided by strict formal underpinnings</td>
<td>Architectures in the guidance, navigation, and control (GN&amp;C) domain</td>
<td>Modeling and simulation of the dynamic behavior described by an architecture</td>
<td>Formal refinement of architectures across levels of detail</td>
<td>Glue code generation for interconnecting existing components using common interaction protocols</td>
<td>Data-flow architectures, characterized by high-volume of data and real-time requirements on its processing</td>
<td>Modeling and analysis (specifically, deadlock analysis) of the dynamic behavior of concurrent systems</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Medvidovic & Taylor, 2000, p. 78)
Addressing the question of:

- How do we specify architectural attributes?
  - *e.g.,*
    - Functionality
    - Quality
Functionality is?

- System’s ability to do that which was intended
  - implies
    - Coordination
    - Communication
    - Control
System quality attributes?

- Both architectural and non-architectural
  - e.g.,
    - Usability
    - Modifiability
    - Performance

- Never achieved in isolation

- Generally interrelated

- One architectural goal is to identify the tensions, trade offs and relationships
Historical problems with discussions of system quality attributes

- Definitions are not operational
  - E.g., “system must be modifiable.” OK how, under what conditions…

- Where does the attribute belong, to what system? Who ‘owns’ it?

- Many vocabularies for the same construct
Response to quality specification challenges

- Quality Attribute Scenarios
  - Address non operational definitions and overlapping attribute concerns
  - A description and discussion of the scenario address the issue of ‘vocabularies’
Quality Attribute Scenarios

- Six parts to a quality attribute scenario
  - Source of stimulus
    - What generates the stimulus. Machine or human
  - Stimulus
    - Condition considered when stimulus is received
  - Environment
    - Within what conditions
  - Artifact
    - what is acted upon or acts
  - Response
  - Response measure or metric
    - What is done, how quantified and measured/
General scenarios

- Spawn system specific scenarios
- Recursive iterations of scenarios
  - A kind of decomposition until complete
- They also provide a kind of template for commonly included quality attributes
  - Availability, modifiability, security, usability, testability etc.
Tactics: quality attribute decisions

- A design decision influencing the control of a quality attribute response
- A collection of tactics is an architectural strategy
- One or more tactics for each attribute
- Tactics impact other tactics
- Patterns are packages of tactics
Tactic example

- Derived from the “availability” attribute

- Wherein defined a failure vs. a fault
  - Failure discernible by users
  - Fault or faults may lead to failure

- Example: Tactics to keep faults from becoming failures
  - Ping/echo, heartbeat (dead man timer), exceptions

- Example: Tactics for fault recovery

- Hierarchical under the notion of the “availability” attribute
Tactics and Architectural patterns

- A pattern is a kind of template of tactics that may be used, revised, referenced in a design

- A pattern realized in software is a ‘Style’
  - C.f., figures 5.12-5.14
Architecture Description Languages (ADL)

- Focuses on the high-level structure of a system
  - As opposed to detailed implementation level

- Many
  - AESOP, ArTek, Darwin, LILEANNA, MetaH, Rapide, UniCon, Weaves, Wright.

- Architecture Exchange language
  - To map (translate) between the variations
  - Lack of a ‘standard’
What should be included?

- Opinions range between extremes on a continuum
  - One end of the spectrum sees the ADL as
    - A tool to aid communication and understanding
    - Simple, understandable, universal, graphical, maybe not formalized
  - The other end of the spectrum sees the ADL as providing:
    - Formal syntax, rules, semantics and tools
      - Analysis tools, syntax checkers, parsers, compilers, runtime support…

- A composite definition might suggest that the…
  - ADL must explicitly model, components, connectors and configurations; must provide tool-support for architecture-based development
Minimal requirements of an ADL

**The language must:**
- Be suitable for communicating an architecture to all interested parties
- Support the tasks of architecture creation, refinement and validation
- Provide a basis for further implementation, so it must be able to add information to the ADL specification to enable the final system specification to be derived from the ADL
- Provide the ability to represent most of the common architectural styles
- Support analytical capabilities or provide quick generating prototype implementations

ADLs have in common:

- Graphical syntax with often a textual form and a formally defined syntax and semantics
- Features for modeling distributed systems
- Little support for capturing design information, except through general purpose annotation mechanisms
- Ability to represent hierarchical levels of detail including the creation of substructures by instantiating templates
ADLs differ in their ability to:

- Handle real-time constructs, such as deadlines and task priorities, at the architectural level
- Support the specification of different architectural styles. Few handle object oriented class inheritance or dynamic architectures
- Support the analysis of the architecture
- Handle different instantiations of the same architecture, in relation to product line architectures

The good and bad of ADLs

- Positive aspects:
  - ADLs are a formal way of representing architecture
  - ADLs are intended to be both human and machine readable
  - ADLs support describing a system at a higher level than previously possible
  - ADLs permit analysis and assessment of architectures, for completeness, consistency, ambiguity, and performance
  - ADLs can support automatic generation of software systems

- Negative elements:
  - There is no universal agreement on what ADLs should represent, particularly as regards the behavior of the architecture
  - Representations currently in use are relatively difficult to parse and are not supported by commercial tools
  - Most ADLs tend to be very vertically optimized toward a particular kind of analysis

ADLs

• The positives
  • ADLs represent a formal way of representing architecture
  • ADLs are intended to be both human and machine readable
  • ADLs support describing a system at a higher level than previously possible
  • ADLs permit analysis of architectures — completeness, consistency, ambiguity, and performance
  • ADLs can support automatic generation of software

• The negatives
  • There is not universal agreement on what ADLs should represent, particularly as regards the behavior of the architecture
  • Representations currently in use are relatively difficult to parse and are not supported by commercial tools
  • Most ADL work today has been undertaken with academic rather than commercial goals in mind rather than commercial goals in mind
  • Most ADLs tend to be very vertically optimized toward a particular kind of analysis
The ADL Framework

- **Define Architecture** (on pg. 72, using Shaw and Garlan’s, 1996, definition) as:
  - “Software Architecture is a level of design that involves the description of: elements from which systems are built, interactions among those [systems] elements, patterns that guide their composition, and constraints on these patterns.”
Common ADL constructs

- Components
  - Either data store or unit of computation
- Connectors
- Systems
  - (configurations of components and connectors)
- Ports
  - (points of interaction with a component)
- Representations - for modeling hierarchical composition
- Rep-maps,
  - mapping composite component’s or connector’s internal architecture of the external interface
Components

- represent the primary computational elements and data stores of a system.
- Typical examples of component include such things as clients, servers, filters, objects, blackboards and databases.
- Components may have multiple interfaces, each interface defining a point of interaction between a component and its environment.

<table>
<thead>
<tr>
<th>Type of a Component</th>
<th>Interaction Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>Procedure Call, Data Sharing</td>
</tr>
<tr>
<td>Object</td>
<td>Method Invocation</td>
</tr>
<tr>
<td>Filter</td>
<td>Data Flow</td>
</tr>
<tr>
<td>Data File</td>
<td>Read, Write</td>
</tr>
<tr>
<td>Database</td>
<td>Schema, Query Language</td>
</tr>
</tbody>
</table>

Table 1.1: Types of Components and Interactions Supported by them
Connectors

Connectors represent interaction among components.
- They provide the glue for architectural designs.
- From the run time perspective, connectors mediate the communication and coordination activities among components.
  - Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast.

Connectors may also represent complex interactions, such as client-server protocol, or a SQL Link between a database and an application.

Connectors have interfaces that define the roles played by the participants in the interaction.
Systems

- System represents graphs of components and connectors.
  - A particular arrangement of components and connectors are defined as a system configuration.
  - In general, systems may be hierarchical.
  - Components and connectors may represent subsystems that have their own internal architecture.
Architectural Styles

- Architectural styles describe the families of system that use the same types of components, types of interactions, structural constraints, and analysis.

- Systems built within a single style can be expected to be more compatible than those that mix styles:
  - it may be easier to make them interoperate, and
  - it may be easier to reuse parts within the family.
Application Oriented Properties

- These properties describe the states of a data structure that are of significance to the processing elements manipulating that structure.

- They can be used for such things as controlling the order of processing, helping to define the effects of a processing element on a data structure and even helping to define operations needed by the processing elements to achieve those effects.
The ADL Framework

- **Architecture** -->…description of elements from which systems are built, interactions among elements, patterns guiding composition and constraints on the patterns.

- **Component** ≈ units of computation, data store. Loci of computation and of state. Big and small.

- **Connectors** ≈ building blocks modeling interactions among components and rules about interactions

- **Configurations** ≈ graphical descriptions of architectural structure
ADL classification hierarchy

- Components
  - Interface
  - Types
  - Semantics
  - Constraints
  - Evolution
  - Non-functional properties

- Connectors
  - Interface
  - Types
  - Semantics
  - Constraints
  - Evolution
  - Non-functional properties

Architectural configurations

- Understandability
- Compositionality
- Refinement and traceability
- Heterogeneity
- Scalability
- Evolution
- Dynamism
- Constraints
- Non-functional properties
Components are…

- A unit of computation OR a data store
  - Loci of computation and state
  - Can be small (single procedure) or large (an application)
  - May own or share execution space or data
- Features of component
  - Interface - interaction points to the outside world
  - Types - abstractions encapsulating functionality into reusable chunks
  - Semantics - high-level model of component behavior
  - Constraints - boundary condition defining where violation renders something unacceptable
  - Evolution - trace or log of modifications
  - Nonfunctional property - description of elements not directly derived form specification of behavior
Connectors

- Building block modeling interactions among components and rules governing the interactions
  - Features of component
    - Interface - interaction points to other components and connectors
    - Types- abstractions encapsulating communication, coordination and mediation decisions
    - Semantics - high-level model of connector’s behavior
    - Constraints ensure adherence to intended interaction protocols, establish inter connector dependencies and enforce usage boundaries.
    - Evolution - trace of modification to properties such as interface, semantics or constraints
    - Nonfunctional property - description of additional requirements for correct connector implementation
Configurations (topologies)

- **Graphs of components and connectors** that describe architectural structure.
  - Three categories of Features
    - Qualities of the description
    - Qualities of the described system
    - Properties of the described system
  - Understandability: must model structural info. Simply and with simple syntax
  - Compositionality: allows description at different levels of detail
  - Refinement and traceability: how to allow correct and consistent refinement
  - Heterogeneity: ‘openness’ allowing specification and development with heterogeneous components and connectors.
  - Scalability: allowing growth
  - Evolution: refers to ‘offline’ changes in an architecture and system
  - Dynamism: modification while the system is operating. Helps model dynamic changes and method for allowing it
  - Constraints:
    - Non-functional properties: system-level ways to select components and connectors, perform analysis, enforce constraints etc.
Kinds of (hoped for) Tool support in an ADL

- Active specification - contingent choices.
  - Help reduce design space dependent on other choices already made

- Multiple views-
  - supports different slices of reality/need

- Analysis - help map implications up and down stream

- Refinement –
  - during refinement and iteration of design help enforce correctness and consistency

- Implementation generation –
  - help produce source code (classes, objects, instances, etc.)

- Dynamism –
  - help map and follow changes
Example ADL models

- ACME (AIL) Architecture Interchange Language
- Rapide (ADL)
ACME -

an architecture interchange language

- “As close to consensus as exists” (Medvidovic & Taylor, 200, p.72.)

- Represents the least common denominator of minimally required parts to an ADL
  1. Components
  2. Connectors
  3. Systems or configurations of components and connectors
  4. Ports, points of interaction with a component
  5. Roles, or points of interaction with a connector
  6. Representations—for modeling hierarchical composition
  7. Rep-maps, mapping composite component’s or connector’s internal architecture of the external interface

- Does not classify elements in property lists or help understand elements
‘Rapide’ - an event based ADL (EADL) or Interface Connection Architecture

- **an Executable ADL**
  - An architecture in Rapide is an “executable specification of a class of systems. It can be any level of abstraction. An architecture consists of interfaces, connections and constraints.” (p. 717 Luckham and Vera)
  - Interface -- specify behavior of the system
    - The expected inputs and outputs of components
  - Connection -- define communications between components, using only features form the interfaces
  - Constraint -- restrict the behavior of interfaces and connections

- For more go to…
  - === or ===
  - http://www.sei.cmu.edu/str/descriptions/adl_body.html
Why have a prototyping EADL?

- Study and predict behavior of a system before building
- When Rapide is executed it tracks and checks constraints conformance.
Rapide’s notions of ADL simulation requirements

- Each interface must have a module (component)
- Each module must conform to the interface
- Components communicate ONLY as specified by the interface connections of the architecture
Rapide as Event Processing Simulation

- Event are?
  - Tuples of data
    - Who generated the event
    - What activity was done (called for)
    - Data values (state)
    - Time and duration

- Asynchronous events modeled
  - By allowing connections to react to and call other events
Interface Connection Architecture

- A set of interfaces, a set of connections rules and a set of constraints
  - Connection rules define relationships between events independently of any implementation
    - Connection abstract construct
    - Connections re defined using event patterns
- Patterns are...?
### TABLE 2
ADL Support for Modeling Components

<table>
<thead>
<tr>
<th>Features</th>
<th>Characteristics</th>
<th>Interface</th>
<th>Types</th>
<th>Semantics</th>
<th>Constraints</th>
<th>Evolution</th>
<th>Non-Functional Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
<td>Component; implementation independent</td>
<td>interface points are ports</td>
<td>extensible type system; parameterization enabled with templates</td>
<td>no support; can use other ADLs’ semantic models in property lists</td>
<td>via interfaces only</td>
<td>structural subtyping via the extends feature</td>
<td>allows any attribute in property lists, but does not operate on them</td>
</tr>
<tr>
<td>Aesop</td>
<td>Component; implementation independent</td>
<td>interface points are input and output ports</td>
<td>extensible type system</td>
<td>(optional) style-specific languages for specifying semantics</td>
<td>via interfaces and semantics; stylistic invariants</td>
<td>behavior-preserving subtyping</td>
<td>allows association of arbitrary text with components</td>
</tr>
<tr>
<td>C2</td>
<td>Component; implementation independent</td>
<td>interface exported through top and bottom ports; interface elements are provided and required</td>
<td>extensible type system</td>
<td>component invariants and operation pre- and postconditions in 1st order logic</td>
<td>via interfaces and semantics; stylistic invariants</td>
<td>heterogeneous subtyping</td>
<td>none</td>
</tr>
<tr>
<td>Darwin</td>
<td>Component; implementation independent;</td>
<td>interface points are services (provided and required)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MetaH</td>
<td>Process; implementation constraining</td>
<td>interface points are ports</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rapide</td>
<td>Interface; implementation independent</td>
<td>interface points are constituents (provides, requires, action, and service)</td>
<td>extensible type system; contains a typesublanguage; supports parameterization</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SADL</td>
<td>Component; implementation independent;</td>
<td>interface points are input and output ports (iports and oports)</td>
<td>extensible type system; allows parameterization of component signatures</td>
<td>none</td>
<td>via interfaces; stylistic invariants</td>
<td>subtyping by constraining supertypes; refinement via pattern maps</td>
<td>requires component modification (see Section 4.3.9)</td>
</tr>
<tr>
<td>UniCon</td>
<td>Component; implementation constraining</td>
<td>interface points are players</td>
<td>predefined, enumerated set of types</td>
<td>event traces in property lists</td>
<td>via interfaces and semantics; attributes; restrictions on players that can be provided by component types</td>
<td>none</td>
<td>attributes for schedulability analysis</td>
</tr>
<tr>
<td>Weaves</td>
<td>Tool fragments; implementation constraining</td>
<td>interface points are read and write ports; interface elements are objects</td>
<td>extensible type system; types are component sockets</td>
<td>partial ordering over input and output objects</td>
<td>via interface and semantics</td>
<td>none</td>
<td>allows association of arbitrary, uninterpreted annotations with components</td>
</tr>
<tr>
<td>Wright</td>
<td>Component; implementation independent;</td>
<td>interface points are ports; port interaction semantics specified in CSP</td>
<td>extensible type system; parameterizable number of ports and computation</td>
<td>not the focus; allowed in CSP</td>
<td>protocols of interaction for each port in CSP; stylistic invariants</td>
<td>via different parameter instantiations</td>
<td>none</td>
</tr>
</tbody>
</table>

(Medvidovic & Taylor, 2000, p. 78)
<table>
<thead>
<tr>
<th>Features of ( \text{ADL} )</th>
<th>Characteristics</th>
<th>Interface</th>
<th>Types</th>
<th>Semantics</th>
<th>Constraints</th>
<th>Evolution</th>
<th>Non-Functional Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
<td>Connector; explicit; interface points are roles</td>
<td>interface system, based on protocols; parameterization via templates</td>
<td>extensible type system, based on protocols</td>
<td>no support; can use other ADLs' semantic models in property lists</td>
<td>via interfaces and structural for type instances</td>
<td>structural subtyping via the \textit{extends} feature</td>
<td>allows any attribute in property lists, but does not operate on them</td>
</tr>
<tr>
<td>Aesop</td>
<td>Connector; explicit; interface points are roles</td>
<td>interface system, based on protocols</td>
<td>(optional) semantics specified using Wright</td>
<td>via interfaces and sematics; stylistic invariants</td>
<td>via semantics; stylistic invariants (each port participates in one link only)</td>
<td>behavior-preserving subtyping</td>
<td>allows association of arbitrary text with connectors</td>
</tr>
<tr>
<td>C2</td>
<td>Connector; explicit; interface with each component via a separate port; interface elements are provided and required</td>
<td>interface system, based on protocols</td>
<td>partial semantics specified via message filters</td>
<td>via semantics; stylistic invariants</td>
<td>context-reflexive interfaces; evolvable filtering mechanisms</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Darwin</td>
<td>Binding; in-line; no explicit modeling of component interactions</td>
<td>none; allows &quot;connection components&quot;</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>MetaH</td>
<td>Connection; in-line; allows connections to be optionally named</td>
<td>none</td>
<td>none; supports three general classes of connections: port, event, and equivalence</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Rapide</td>
<td>Connection; in-line; complex reusable connectors only via &quot;connection components&quot;</td>
<td>none; allows &quot;connection components&quot;</td>
<td>none</td>
<td>posets; conditional connections</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>SADL</td>
<td>Connector; explicit; connector signature specifies the supported data types</td>
<td>interface system, parameterized signatures and constraints</td>
<td>extensible type system</td>
<td>axioms in the constraint language</td>
<td>via interfaces; stylistic invariants</td>
<td>subtyping; connector refinement via pattern maps</td>
<td>requires connector modification (see Section 4.3.9)</td>
</tr>
<tr>
<td>UniCon</td>
<td>Connector; explicit; interface points are roles</td>
<td>predefined, enumerated set of types</td>
<td>predefined, enumerated set of types</td>
<td>implicit in connector's type; semantic information can be given in property lists</td>
<td>via interfaces; restricts the type of players that can be used in a given role</td>
<td>none</td>
<td>attributes for schedulability analysis</td>
</tr>
<tr>
<td>Transport services; explicit</td>
<td>interface points are the encapsulating socket pads</td>
<td>interface system, types are connector sockets</td>
<td>via naming conventions</td>
<td>via interface</td>
<td>none</td>
<td>none</td>
<td>allows association of arbitrary, uninterpreted annotations with transport services</td>
</tr>
<tr>
<td>Weaves</td>
<td>Connector; explicit</td>
<td>interface points are roles; role interaction semantics specified in CSP</td>
<td>connector glue semantics in CSP</td>
<td>via interfaces and semantics; protocols of interaction for each role in CSP; stylistic invariants</td>
<td>via different parameter instantiations</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
<td>---------------</td>
<td>------------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>ADL</td>
<td>Attachments: explicit; concise textual specification</td>
<td>provided via templates, representations, and rep-maps</td>
<td>rep-maps</td>
<td>open property lists; required explicit mapping across ADLs</td>
<td>aided by explicit configurations; hampered by fixed number of roles</td>
<td>aided by explicit configurations; first-class families</td>
<td>none</td>
</tr>
<tr>
<td>ACME</td>
<td>Configur. explicit; concise graphical specification, developed for visualization</td>
<td>provided via representations</td>
<td>none</td>
<td>allows multiple languages for modeling semantical supports development in C</td>
<td>aided by explicit configurations; hampered by fixed number of roles</td>
<td>no support for partial architectures; aided by explicit configurations</td>
<td>none</td>
</tr>
<tr>
<td>Aesop</td>
<td>Archit. explicit; Topology explicit; graphical specification</td>
<td>allowed; supports internal component architecture</td>
<td>none</td>
<td>enabled by internal component architecture; supports development in C++, Java, and Ada</td>
<td>aided by explicit configurations and variable number of connector ports; used in the construction of own tool suite</td>
<td>allows partial architectures; aided by explicit configurations</td>
<td>none</td>
</tr>
<tr>
<td>C2</td>
<td>Binding explicit</td>
<td>in-line textual specification with many connector details; provides graphical notation</td>
<td>supported via macros</td>
<td>supports system generation when implementing constraints</td>
<td>supports development in Ada; requires all components to contain a process dispatch loop</td>
<td>hampered by in-line configurations</td>
<td>no support for partial architectures; hampered by in-line configurations</td>
</tr>
<tr>
<td>Darwin</td>
<td>Connection in-line</td>
<td>in-line textual specification with many connector details; provides graphical notation</td>
<td>supported via language’s composite component feature</td>
<td>supports system generation; implementation constraints</td>
<td>supports development in Ada; non-functional constraints</td>
<td>hampered by in-line configurations</td>
<td>no support for partial architectures; hampered by in-line configurations</td>
</tr>
<tr>
<td>MetaII</td>
<td>Connect explicit</td>
<td>in-line textual specification with many connector details; provides graphical notation</td>
<td>supported via language’s component feature</td>
<td>supports system generation; implementation constraints</td>
<td>supports development in Ada; non-functional constraints</td>
<td>hampered by in-line configurations</td>
<td>no support for partial architectures; hampered by in-line configurations</td>
</tr>
<tr>
<td>Rapide</td>
<td>Configuration explicit</td>
<td>in-line textual specification with many connector details; provides graphical notation</td>
<td>supported via language’s component feature</td>
<td>supports system generation; implementation constraints</td>
<td>supports development in Ada; non-functional constraints</td>
<td>hampered by in-line configurations</td>
<td>no support for partial architectures; hampered by in-line configurations</td>
</tr>
<tr>
<td>SADL</td>
<td>Connect explicit</td>
<td>explicit, concise textual specification</td>
<td>supported through composite components and connectors</td>
<td>supports system generation; implementation constraints</td>
<td>supports only predefined components and connectors</td>
<td>aided by explicit configurations; used in large-scale project</td>
<td>no support for partial architectures; hampered by explicit configurations</td>
</tr>
<tr>
<td>UniCon</td>
<td>Connect explicit</td>
<td>explicit, concise textual specification</td>
<td>supported through composite components and connectors</td>
<td>supports system generation; implementation constraints</td>
<td>supports only predefined components and connectors</td>
<td>aided by explicit configurations; used in large-scale project</td>
<td>no support for partial architectures; hampered by explicit configurations</td>
</tr>
<tr>
<td>Weave</td>
<td>Connect explicit</td>
<td>explicit, concise textual specification</td>
<td>supported through composite components and connectors</td>
<td>supports system generation; implementation constraints</td>
<td>supports only predefined components and connectors</td>
<td>aided by explicit configurations; used in large-scale project</td>
<td>no support for partial architectures; hampered by explicit configurations</td>
</tr>
<tr>
<td>Wright</td>
<td>Attach explicit</td>
<td>explicit, concise textual specification</td>
<td>supported through composite components and connectors</td>
<td>supports system generation; implementation constraints</td>
<td>supports only predefined components and connectors</td>
<td>aided by explicit configurations; used in large-scale project</td>
<td>no support for partial architectures; hampered by explicit configurations</td>
</tr>
</tbody>
</table>

(Medvidovic & Taylor, 2000, p. 78)
<table>
<thead>
<tr>
<th>Features</th>
<th>Active Specification</th>
<th>Multiple Views</th>
<th>Analysis</th>
<th>Refinement</th>
<th>Implementation Generation</th>
<th>Dynamism</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
<td>none</td>
<td>textual; “weblets” in ACME-Web; architecture views in terms of high-level (template), as well as basic constructs</td>
<td>parser</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Aesop</td>
<td>syntax-directed editor for components; visualization classes invoke specialized external editors</td>
<td>textual and graphical; style-specific visualizations; component and connector types distinguished iconically</td>
<td>parser, style-specific compiler; type checker; cycle checker; checker for resource conflicts; and scheduling feasibility</td>
<td>none</td>
<td>build tool constructs system glue code in C for pipe-and-filter style</td>
<td>none</td>
</tr>
<tr>
<td>C2</td>
<td>proactive “architecting” process in DRADEL; reactive, non-intrusive type checker; design critics and to-do lists in Argo</td>
<td>textual and graphical; view of development process</td>
<td>parser; style rule checker; type checker</td>
<td>generates application skeletons which can be completed by reusing OTS components</td>
<td>class framework enables generation of C/C++, Ada, and Java code; DRADEL generates application skeletons</td>
<td>ArchStudio allows unanticipated dynamic manipulation of architectures</td>
</tr>
<tr>
<td>Darwin</td>
<td>automated addition of ports to communicating components; propagation of changes across bound ports; dialogs to specify component properties;</td>
<td>textual, graphical, and hierarchical system view</td>
<td>parser; compiler; “what if” scenarios by instantiating parameters and dynamic components</td>
<td>compiler, primitive components are implemented in a traditional programming language</td>
<td>compiler generates C++ code</td>
<td>compilation and runtime support for constrained dynamic change of architectures (replication and conditional configuration)</td>
</tr>
<tr>
<td>MetaH</td>
<td>graphical editor requires error correction once architecture changes are applied; constrains the choice of component properties via menus</td>
<td>textual and graphical; component types distinguished iconically</td>
<td>parser; compiler; schedulability, reliability, and security analysis</td>
<td>compiler; primitive components are implemented in a traditional programming language</td>
<td>DSSA approach; compiler generates Ada code</td>
<td>none</td>
</tr>
<tr>
<td>Rapide</td>
<td>none</td>
<td>textual and graphical; visualization of execution behavior by animating simulations</td>
<td>parser; compiler; analysis via event filtering and animation; constraint checker to ensure valid mappings</td>
<td>compiler for executable sublanguage; tools to compile and verify event pattern maps during simulation</td>
<td>executable simulation construction in Rapide’s executable sublanguage</td>
<td>compilation and runtime support for constrained dynamic change of architectures (conditional configuration)</td>
</tr>
<tr>
<td>SADL</td>
<td>none</td>
<td>textual only</td>
<td>parser; analysis of relative correctness of architectures with respect to a refinement map</td>
<td>checker for adherence of architectures to a manually-proved mapping</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>UniCon</td>
<td>graphical editor prevents errors during design by invoking language checker</td>
<td>textual and graphical; component and connector types distinguished iconically</td>
<td>parser; compiler; schedulability analysis</td>
<td>compiler; primitive components are implemented in a traditional programming language</td>
<td>compiler generates C code</td>
<td>none</td>
</tr>
<tr>
<td>Weaves</td>
<td>none</td>
<td>graphical only; component and connector types (sockets) distinguished iconically</td>
<td>parser; real-time execution animation; low overhead observers; analysis/debugging components in a weave</td>
<td>none</td>
<td>dynamic linking of components in C, C++, Objective C, and Fortran; no code generation</td>
<td>Jacquard allows unanticipated dynamic manipulation of weaves</td>
</tr>
<tr>
<td>Wright</td>
<td>none</td>
<td>textual only; model checker provides a textual equivalent of CSP symbols</td>
<td>parser; model checker for type conformance of ports to roles; analysis of individual connectors for deadlock</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>
For your consideration as an Enterprise Architects

- Where will you use ADLs?
- How will you use ADLs?
- At what level of abstraction in the EA will you deploy ADLs?