xADL – A Better way to Describe Architecture

Surya Prakash Kotha, Mtech, CSE, IIT Kanpur
Under the guidance of Prof. T.V. Prabhakar

***Abstract*— Software architecture can help people to better understand the total structure of the system (Software). In recent years software size and complexity has enormously increasing. This made the description of their architecture more complicate. So to describe these architectures many methods are invented. These are called as Architecture Descriptive languages. But many of these are not succeeded. Since these are devised for a particular style of architecture. This made the invention of an eXtensible Architecture Description language. Research and experimentation in the field of software architecture yielded invention of many ADLs (Architecture Description Languages). xADL 2.0 (pronounced as *"zay-dul"*) is one ADL which is designed to suit that type of problems. This can be used to describe the architecture of the various types of systems. It is developed in that way to adapt the changes in the architecture styles. This paper introduces and explains these characteristics of the xADL 2.0.**

***Index Terms*— xArch, Archstudio 3.0, XML, Software Architecture**

I. Introduction

Software architecture provides a way to view the system in an abstract way. It can be viewed as bridge between requirements to design. By this we can see the system as composed of components, connectors and interfaces and interactions among the components. Software architecture description can be viewed as abstract representation of software in an Architecture Description Language. Software Architecture is a tool to reduce or estimate the development cost of the system. It will provide a basic idea (blue print) for the system construction and design it may not be a software system. One needed quality of the software architecture is it should be manageable and can adapt changes. In this paper I will introduce software architecture in section 2, ADLs and their properties in section 3, about xADL in section 4, differences between xADL and other ADLs in section 5 and an example system architecture using xADL at the end. Various ADLs have been proposed in the community of SA, such as Darwin, C2, Rapide, Aesop, Wright, SADL, Acme. But none of them is really a good solution because each of them was designed for a particular system. xADL 2.0 is released in 12-11-2002.

II. Software Architecture

The concept of software architecture was evolved in early 60's but upto early 90's no one has showed real interest in this topic. In the early 90's the software has grown large in size and complexity. In particular there is no standard definition for software architecture.

Bass, et al., 1994--

Writing about a method to evaluate architectures with respect to the quality attributes they instill in a system, Bass and his colleagues write that

The architectural design of a system can be described from (at least) three perspectives functional partitioning of its domain of interest, its structure, and the allocation of domain function to that structure.

Garlan and Perry, 1995--

David Garlan and Dewayne Perry have adopted the following definition for their guest editorial to the April 1995 IEEE Transactions on Software Engineering devoted to software architecture

The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.

(The source of this definition was a weekly discussion group devoted to software architecture at the Software Engineering Institute.)

Boehm, et al., 1995

Barry Boehm and his students at the USC Center for Software Engineering write that

- A Software system architecture comprises
- A collection of software and system components, connections, and constraints.
- A collection of system stakeholders need statements.
- A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders need statements.

Bass, Clements, and Kazman.in Software Architecture in Practice [Bass 97]

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Software architecture can be viewed as a bridge between requirements and design. It also gives a higher lever of abstraction of the system architecture. It is a set of components with independent functions and explicit interfaces and interactions between the components. Software architecture description is the representation of abstract architectural model

in Architecture Description Language (ADL). If we see the architecture of the system as a graph, then Nodes in that graph can be viewed as components. These components can be database store, servers, clients, terminals, human interface systems, etc. The arcs in the graph can be viewed as connectors between these components. In one sense these will give the path of interaction between these components. We can simply say that these connectors will give the run time behavior of the system. To give the behavior of the system each element in the graph is associated with some properties or attributes. In the same way the connectors and components are also associated with some attributes. For example, properties associated with a connector might define its protocol of interaction, or performance attributes (e.g., delay, bandwidth). The architecture can also specify the list of constraints that must be maintained. The architecture very important to know evaluate the system (in terms of effort and cost). The architecture of a system also makes the evaluation of the system. Representing architecture as an arbitrary graph of generic components and connectors has the advantage of being extremely general and open ended. The architecture of system must address Extendibility and Interchangeability. An architectural style typically defines a set of types for components, connectors, interfaces, and properties together with a set of rules that govern how elements of those types may be composed. These architectures can be explained efficiently using architecture styles. Since architectural styles may address different aspects of software architecture, a given architecture may be composed of multiple styles. Likewise, a hybrid style can be formed by combining multiple basic styles into a single coordinated style.

Architecture style:

An **architectural style** is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style. Some well known architecture styles are Pipes and Filter style, Product line, Client Server Architecture style, Distributed Services etc.

There are advantages and disadvantages in using architecture styles. The disadvantage is designing software within an architecture style makes the development harder. But we can get beneficial system properties if we use these architecture styles. That's why we use a combination of architecture styles.

III. ARCHITECTURE DESCRIPTION LANGUAGE

Architectural Description language or architectural definition language (or an ADL) is an abstract level description of the software system using the components, connectors, links etc., in a formal language or notations. Actually there is no concrete definition for the ADL so far now. An ADL will provide its users *concrete syntax, formal semantics, conceptual framework*. These are helpful for explicitly modeling software systems. All these ADLs should have some properties they are *components, connectors, interfaces, links and interconnection between the components.*

The research on software architecture is growing many new software domains are developed. This will result in rapid growth in the ADLs. Each describing or having capabilities to describe a specific domain. These ADLs have their own specific notations and these notations are typically supported by the tools to facilitate better understanding, analyzing and visualizing the architecture of the software. Some of the ADLs so far known are Rapide, Darwin, Mae, Koala, C2SADEL, ACME etc..

A survey revealed that the minimum requirement for any language to be an ADL is to able to represent (or having the notations for) components, connectors, links, configurations. These ADLs should have good tool support for analyzing the software and visualizing. But the main concern is can we have a single ADL for all these domains of architectures. This is the motivating idea behind designing the xADL. (specifically 2.0 version)

IV. xADL

A. Motivating idea:

Each of the ADLs I have said in the previous section are useful for a specific domain. For example Koala is best suited for Product line Architectures, Mae is used for describing Configuration Management architectures, ACME is used as an interchange language between the ADLs rather than Architecture Description Language. But if a new domain comes (in present days it is possible as a result of the increasing research on the software architectures) these ADLs are not able to represent this software domain. Some of the ADLs and their specific domains are

ADL	Supporting Domain
Darwin, C2SADL	Dynamic systems
Mae	Configuration management
Koala	Product-line Families
Rapide	Event oriented system
Wright	Behavioral properties

These ADLs are for a specific domain if a new architecture type such as Mobile systems or mobile dynamic architecture comes any of the existing ADLs are not able to describe this architecture types. So the first solution is to design a new ADL which will support this domain which will not solve the purpose. The second solution is to design an ADL which is extendible to this new architecture which contains the common properties of these ADLs at core and can extend these schemas to be able to solve the new architecture. One issue is that the target ADL (new ADL to be developed) should be modularly extensible so that the users of the ADL can add new modules to the ADL without changing the basic properties of the ADL. Another concern is that ADL should support or can be extensible to different (possible conflicting) domains. So the

solution is an Extendible ADL which can be extendible to support different types of systems.

B. Introduction to xADL:

As described above the core of the xADL consists of common modules to describe *components, connectors, interfaces, links (configurations).* These modules are developed as XML schemas. The idea behind using XML is, it is very good for interchange of structures data exchange, extendible and it is a modular language. To maximize the reusability of these schemas these are made as generic as possible. That is one instance is written to represent components and connectors, but their behavior and communication are represented in another schema. Thus, aspects of elements like behaviors and constraints on how elements may be arranged are not specified. Such aspects are meant to be defined in extension schemas.

The important features of the ADL which are extended from the xADL2.0 are

1. *Separation between design time schemas and runtime schemas.*
2. *Implementation mappings that map the ADL specifications into code.*
3. *The ability to model the architecture of product line architectures and configuration management systems.*

1. Separation between design time schemas and runtime schemas:

All the traditional ADLs implemented only design time aspects of any system or a combined design time specification and run time specification. But the research on dynamic system, which will change their architecture at run time, needed the separation of these two schemas. The run time specification of the system tells about the state of the system such as running, blocked etc., and the design time schemas will provide the meta-information about the components such as author, date created et., the design time schemas will also tell about the constraints which are imposed on the system, and the expected behavior of the system.

xADL schemas will provide the information about the run time specification using the INSTANCES schema. The instances schema will provide the core set of the architectural constructs of the xADL.

The INSTANCES schema defines the core set of architectural constructs common to most ADLs. The INSTANCES schema provides definitions of:

- component instances;
- connector instances;
- interface instances (on components and connectors);
- link instances;
- sub-architectures (composite components and connectors with internal architectures); and
- general groups.

Constructs modeling design-time aspects of a system are defined in another schema, the STRUCTURE & TYPES schema. This schema provides definitions of:

- components;
- connectors;
- interfaces (on components and connectors);
- links;
- sub-architectures (composite components and connectors with
- internal architectures);
- general groups;
- component types;
- connector types; and
- interface types.

Purpose	Schema	Features
Design Time and Runtime Schemas	Instances	Run-time component, connector, interface and link instances; sub architectures; general groups
	Types and Structures	Design time Components, connectors, interfaces and links; sub architectures; general groups; components, connector and interface types
Implementat-ion mappings	Abstract implementations	Placeholder for implementation data for components, connectors and interfaces.
	Java implementations	Concrete implementation data for Java-language components, connectors and interfaces
Architecture Evolution Management / Product Line Architectures	Versions	Version graphs for component, connector; and interface types.
	Options	Optional design time components, connectors, and links.
	Variants	Variant design- time component and connector types.

In addition to providing a structural model of the system at design time, the STRUCTURE & TYPES schema also includes a generic type system for architectural elements. Types can be assigned to components, connectors, and interfaces, allowing an architect to reason about similarities among elements of the same type. Providing separate models for run-time and design-time aspects of the system ensures that they can be extended separately. While the models are similar, independent extensions to each schema can be created to model additional aspects of a running system or a system design.

2. Implementation Mappings:

All the need for designing any ADL is to maker coding easy. But it is very hard to determine (or we don't know in advance in which platform this ADL will be used). So there are two ways to implement code generation. The first way is to implement or include java classes and archives, windows DLLs, UNIX shared libraries and CORBA components and

etc., this will be a very long process and also making a feasible list is also very hard.

To avoid this xADL2.0 implements the code generation phases in two phases. To generate the code for the architecture in the first phase it implements abstract implementation.

The first level of specification is abstract, and defines *where* implementation data should go in an architecture description, but not *what* the data should be. The xADL 2.0 ABSTRACT IMPLEMENTATION schema extends the STRUCTURE & TYPES schema, and defines a placeholder for implementation data. This placeholder is present on component, connector, and interface types. As such, two elements of the same type share an implementation. The second level of specification is concrete, defining *what* the implementation data is for a particular platform or programming language. Concrete implementation schemas extend the ABSTRACT IMPLEMENTATION schema. xADL 2.0 includes a JAVA IMPLEMENTATION schema that concretely defines a mapping from components, connectors, and interface types to Java classes.

3. To model Product line architectures

Modeling product line architectures is a new concept and also a research area. To implement these architectures xADL2.0 introduces three new schemas they are

- Versions
- Variants
- Optional

Versions record information about the evolution of architectures and elements like components, connectors, and interfaces. Options indicate points of variation in an architecture where the structure may vary by the inclusion or exclusion of an element or group of elements. Variants indicate points in an architecture where one of several alternatives may be substituted for an element or group of elements. xADL 2.0 supports versions, options, and variants, each in a separate schema.

i) Versions:

Versions schema includes versioning constructs to xADL2.0. It defines

1. version graphs for component types
2. version graphs for connector types
3. Version graphs for interface types.

These version graphs capture the evolution of individual elements in an architecture, and, using the subarchitectures mechanism defined in the STRUCTURE & TTYPES schema, can capture the evolution of groups of elements or whole architectures. In keeping with generic nature of xADL 2.0 schemas, they do not constrain the relationship between different versions of individual elements —that they must share some behavioral characteristics or interfaces, for instance. Such constraints may be specified in extension schemas and checked with external tools.

ii) Variants:

The VARIANTS schema allows the types of certain design-time constructs to vary in an architecture. In particular, it defines constructs for:

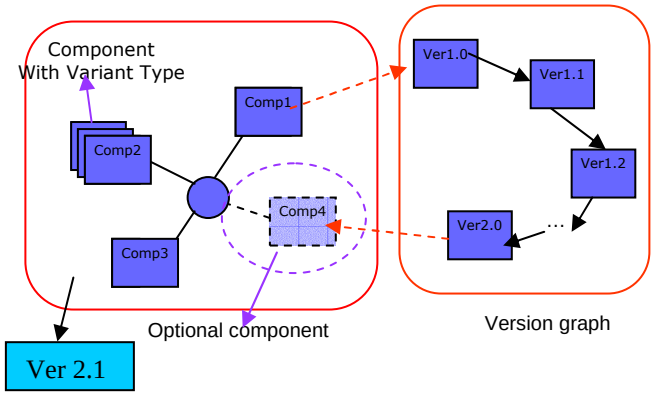
- variant component types; and
- variant connector types.

Variant types contain a set of possible alternatives. Each alternative is a component or connector type accompanied by a guard condition, similar to the one used in the OPTIONS schema. Guards for variants are assumed to be mutually exclusive. The guard conditions are evaluated when the architecture is instantiated. When a guard condition is met, its associated component or connector type is used in place of the variant type.

iii) Options

The OPTIONS schema allows certain design-time constructs to be labeled as optional in architecture. It defines constructs for:

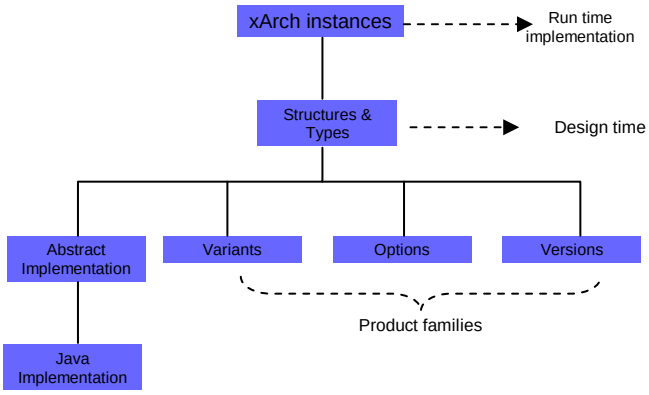
- optional components;
- optional connectors; and
- optional links.



Consider the figure as an example of some software for which the current version 2.1 and we want to represent the complete cycle of its evolution. We can represent using these three schemas.

The component in the previous versions are shown in the version graphs for components like this we can show the connector graphs that is the components and connector in the previous versions. The variant shows the changes in the existing components with respect to the previous versions. Optional schemas represent those components which may or may not present in the current version.

In abstract the schemas present in the xADL are shown as below.



In the software architecture community, the precise meaning and usage of *types* is still a hotly contested issue. Some approaches adopt a fairly traditional programming-language style types-and-instances model. Others adopt a types-as-constraints model, where a type is simply a constraint over elements; any element meeting that constraint is "of that type" (this implies that a single element like a component or connector could potentially have many, many types).

xADL 2.0 adopts the more traditional types-and-instances model found in many programming languages. In this model, components, connectors, and interfaces all have types (called *component types*, *connector types*, and *interface types*, respectively). Links do not have types because links do not have any architectural semantics. The relationships between types, structure, and instances are shown in the following table:

Instance (Run-time)	Structure (Design-time)	Type (Design-time)
Component Instance	Component	Component Type
Connector Instance	Connector	Connector Type
Interface Instance	Interface	Interface Type
Link Instance	Link	*(None)*
Group	Group	*(None)*

Components, connectors, and interfaces are typed because, in real software architectures, there may be multiple elements that are highly similar—perhaps they share behavior or an implementation. When designing xADL 2.0, we intended that types would be linked to implementations; as such, two structural elements (e.g. components) that shared a single type (e.g. component type) would imply that the two components shared an implementation (instantiated from the same class, perhaps, or were instances of the same shared library). However, this particular implication is not carried through in the language unless you choose to adopt the xADL 2.0 implementation schemas, discussed later. Cases where two components or connectors might be of the same type include:

- When there are multiple, generic connectors (e.g. message buses) with identical behavior
- Multiple clients in a client-server system
- Redundant servers in a client-server system
- Two instances of the same component with different inputs (e.g. two instances of the "tee" command in a pipe and filter system).
- etc.

Constructs Defined for Architectural Types

The constructs available for modeling types are also defined in the *Structure & Types* schema, along with the constructs for modeling structure. All the types-related constructs are grouped under a top-level XML element called `ArchTypes`. The xADL 2.0 constructs available for modeling architectural types are:

- Component Types
  - Subarchitectures for Component Types
  - Signatures
- Connector Types*
  - Subarchitectures for Component Types
  - Signatures
- Interface Types

C.Tool Support of the xADL

Using XML as the basis for ADLs makes tool support especially important. While XML documents are plain ASCII and can be written by hand and read visually, the amount of markup and namespace data usually present in an XML document makes it difficult to do so. This increases the need for APIs, editors, and viewers for documents that hide unnecessary XML details.

Various tools present

- COTS/Open Source XML tools
  - XML Authority, XML Spy, Apache Xerces
- In-house tools (developed by ISR, UCI)
  - *Apigen*: Generates DOM-based Java Libraries using only the XML schemas
  - *ArchEdit*: Syntax-directed graphical tree-based editor
  - *Menage*: Graphical editor focusing on product-line features
  - *"Visio for xADL"*: Microsoft Visio extensions provide graphical visualization and editing capabilities for xADL 2.0 architectures.

Visio for xADL serves a the front end for the ArchEdit Data binding libraries for the xADL

Data binding libraries of the xADL

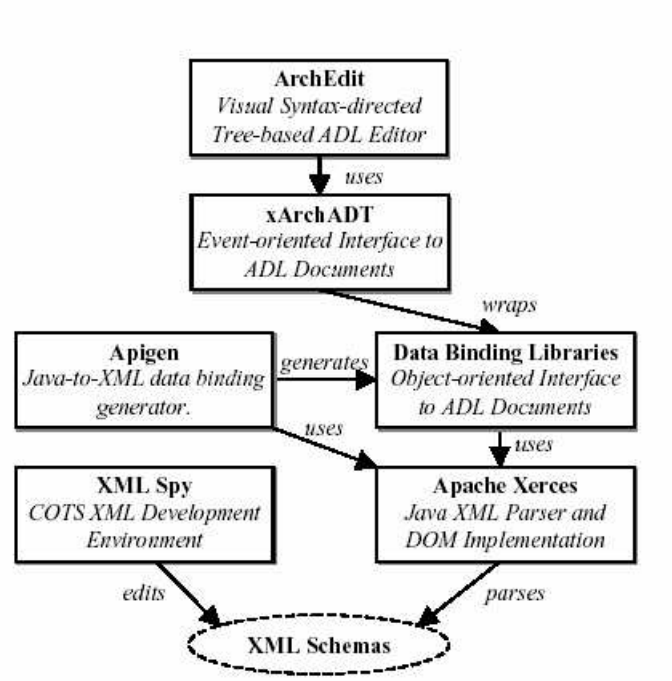
Our infrastructure includes a library of Java-to-XML data bindings that exposes an object-oriented, programmatic interface. This interface hides nearly all details of XML from the user. Using these data bindings significantly reduces the amount of effort needed to build a tool that can parse, understand, and manipulate architecture documents. Once a description of an ADL (in the form of XML schemas) is available, syntax-directed tools can be used to create and edit XML architecture descriptions. Syntax-directed tools hide most XML details from their users. They use the constructs defined by the XML schemas to direct the manipulation of documents and help to ensure syntactic correctness within the defined language. They provide a level of abstraction that is closer to the domain of the architect, exposing elements defined in the language like components, connectors, and interfaces. Data bindings map XML elements and attributes into pieces of code (usually objects), hiding XML details such as namespaces, header tags, sequence ordering, etc. The objects in this library correspond to the types defined in the XML schemas.

APIGEN:

If the data bindings need to be rewritten every time a schema is added, changed, or removed, then the benefit of having them is negated. The syntax information present in the ADL schemas is enough to generate the data bindings automatically. Several projects already generate data bindings for DTD-based languages, but none yet exist that can adequately deal with XML schemas (several projects are in very early 'alpha' stages of development). To remedy this, we built a tool called 'apigen' [11] (short for "API generator") that can automatically generates the Java data binding library, described above, for ADL schemas. Apigen reduces overall tool-building effort by providing the data binding library to tool builders automatically. When an ADL's schemas are changed, tool-builders simply re-run apigen over the modified set of schemas to generate a

new data-binding library. Of course, bindings for elements that did not change will be preserved in the library, minimizing the impact on tools that use the library.

The relationship between these tools:



V. ADVANTAGES OF XADL OVER OTHER ADLS

Actually xADL takes its ides from the ACME and on the survey about the ADLs about the commonalities in the ADLs. The advantages of the xADL over the other ADLs are

- It can be used to represent many architecture domains.
- If any new Architecture domain comes it can be made to describe that domain easily by extending the core of this xADL with other schemas particular to that domain.
- Although it is not used as architecture interchange language it can be used to interchange between the architectural description languages
- It can be used combining two existing domains.
- It serves as basis for experimenting with newer domains. ie., we can check the architecture of the domains for which the architecture is not specified.

Advantages of xADL over the UML

1. UML is vulnerable to changes.
2. UML diagrams are not easy to extend or modify.
3. We cannot model product line architectures efficiently with UML since their architecture changes from version to version.
4. UML encourages object diagrams than the component diagrams.
5. The main purpose of any ADL is to do prior design analysis. But UML does not have any good analysis tools.
6. Using UML diagrams we cannot generate the code which is main need of invention of ADLs. (we can do it UML2.0 through xml Schemas)
7. UML2.0 will not cover all the existing domains. and also if a domain comes it will not able to extend its features to accommodate the changes in its description.

Drawbacks of xADL2.0:

xADL 2.0 does not solve the feature interaction problem.

Feature interaction problem: In a software system, a feature is an optional unit or increment of functionality. If the system specification is organized by features, then it probably takes the form B + F1 + F2 + F3 . . ., where B is a base specification, each Fi is a feature module, and + denotes some feature-composition operation. A feature interaction is some way in which a feature or features modify or influence another feature in defining overall system behavior. Feature interaction is necessary and inevitable in a feature-oriented specification, because so little can be accomplished by features that are completely independent**.** A bad feature interaction is one that causes the specification to be incomplete, inconsistent, or unimplementable, or that causes the overall system behavior to be undesirable. Non-associatively of feature composition can also be considered a feature-interaction problem, on the grounds that for features to be truly optional, their compositions must yield the same behavior when performed in any order. Some forms of nondeterministic behavior can also be considered feature-interaction problems.

Using xADL we cannot find this feature interaction problem in the architecture of the system. Also when presented with two different schemas xADL will not model the architecture. It will only choose one schema over the other.

Other drawback of this ADL is as it is developed recently it does not have much practical experiences. And also the full documentation of the ADL is also not available from the homepage. So the information about the classes (java libraries are also not well documented on the site).

Future work going on xADL:

Arch "Diff" ing and Merging:

- To determine the differences between two xADL2.0 architectures, the work is going in this way.
- To differentiate between two architectures and constructing other architecture by merging these differences in the architectures with one of the architecture.
- Developing schemas to describe Dynamic Distributed Architectures.
- To develop schemas for implementing Graphical layout information for the architectures.

Experiences or practical evolution of the xADL:

AWACS system

To demonstrate the scalability of xADL it was used to model the AWACS aircraft's software architecture in an XML-based ADL. This architecture, consisting of several hundred components and connectors, was modeled using a subset of the xADL 2.0 schemas. It described the initial AWACS description programmatically with a small (1000- line) program. The result was an architecture document consisting of approximately 10,000 lines of XML.

Jet Propulsion Laboratories (JPL)

JPL has adopted xADL to support its Mission Data System (MDS) group, which is experimenting with modeling spacecraft software architectures. This domain induces new modeling needs, particularly a unique notion of component interfaces. JPL has built extensions to the xADL 2.0 base schemas to represent these interfaces, and has used apigen to generate data bindings for these new schemas. xADL 2.0's separation of run-time and design-time models has also been especially important for JPL, since run-time software updates of spacecraft software will be a priority for them. JPL has also created mappings between their XML-based ADL, built in our framework, and other proprietary notations that can be used to drive C++ code generators and software configuration engines already in use at JPL.

Interchange between ADLs Koala and Mae:

To demonstrate our infrastructure's ability to capture concepts from an emerging research area and add tool support for those concepts efficiently, schemas are created for the unique modeling constructs of Koala and Mae to our base xADL 2.0 schemas. Koala and Mae are two representation formats for capturing product line architectures. Each product line consists of multiple variants of software architecture.

SUMMARY:

- xADL takes the idea from ACME
- It is developed as modular language which is extendable
- It is a collection of modules written in XML
- It makes very easy to analyze the Software Architecture
- Using it we can generate code from the architecture *(future work)*

REFERENCES

1. http://www.isr.uci.edu/projects/xarchuci/index.html ---official website of xADL
2. http://www.isr.uci.edu/classes/ics123s02/reading.html
3. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. -Eric M. Dashofy, André van der Hoek Richard N. Taylor
4. xADL: Enabling Architecture-Centric Tool Integration With XML - Rohit Khare, Michael Guntersdorfer, Peyman Oreizy, Nenad Medvidovic, Richard N. Taylor
5. Representing Product Family Architectures in an Extensible Architecture Description Language -Eric M. Dashofy and André van der Hoek
6. Issues in Generating Data Bindings for an XML Schema-Based Language ---Eric M. Dashofy
7. www.U2-partners.0rg -----regarding UML2.0
8. http://wiki.cs.uiuc.edu/cs427/Is+UML+a+good+Architecture+Description+Language%3F ---- about comparisons (survey about UML as a good ADL)
9. **Representing Product Family Architectures in an Extensible Architecture Description Language** ---Eric M. Dashofy and André van der Hoek In *Proceedings of the International Workshop on Product Family Engineering (PFE-4)*, Bilbao, Spain, October 2001.
10. **www**.isr.uci.edu for General architecture of any open system. (example architecture)

Example Architecture:

The architecture shown below is General architecture for implementing any Application (For example I have shown (Internet Explorer)Web Browser and a Word processor).

The architecture is described in Manager (Architect) point of view. The basic components of xADL (components, connectors, interfaces, links) are shown in the figure. It is divided into the fallowing layers

1. Human computer interface
2. Application program
3. Operating system
4. Middle ware

I have taken this architecture from www.isr.uci.edu and described in xADL semantics.

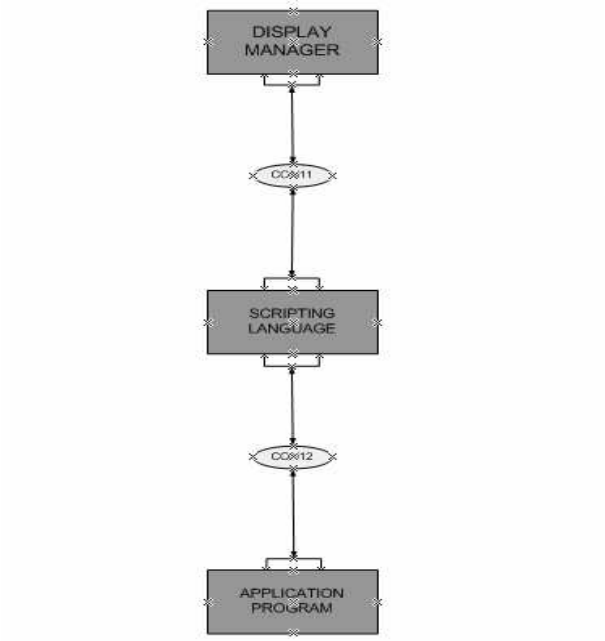
The human computer interface is made different from the application program because it is common for different applications.

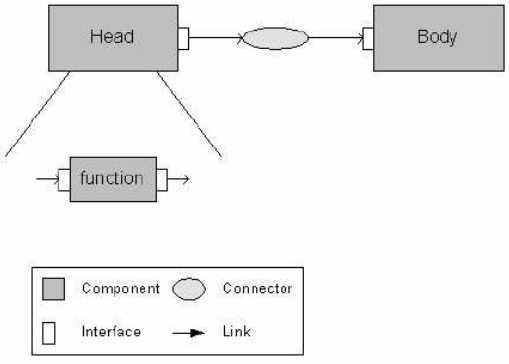
The application program is divided again into 3 parts as GUI display manager, scripting language, and application program for the Web Browser component. Then the subarchitecture can be shown in other figure with out loss of ADL semantics.

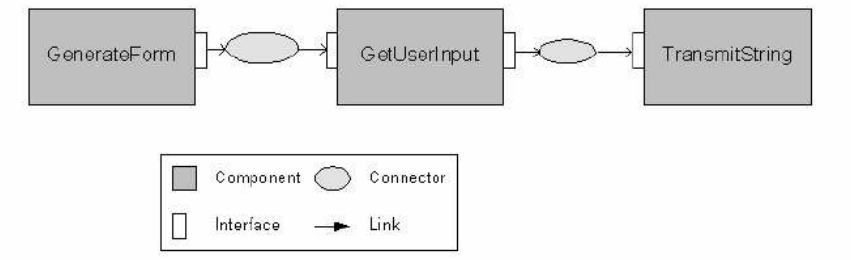
The middleware is used for distributed applications. Here I have shown one such System ORB (CORBA). The architecture of the operating system can also be shown by other diagram. This is known as Sub-architecture the sub –Architecture for the Web Browser is shown below.

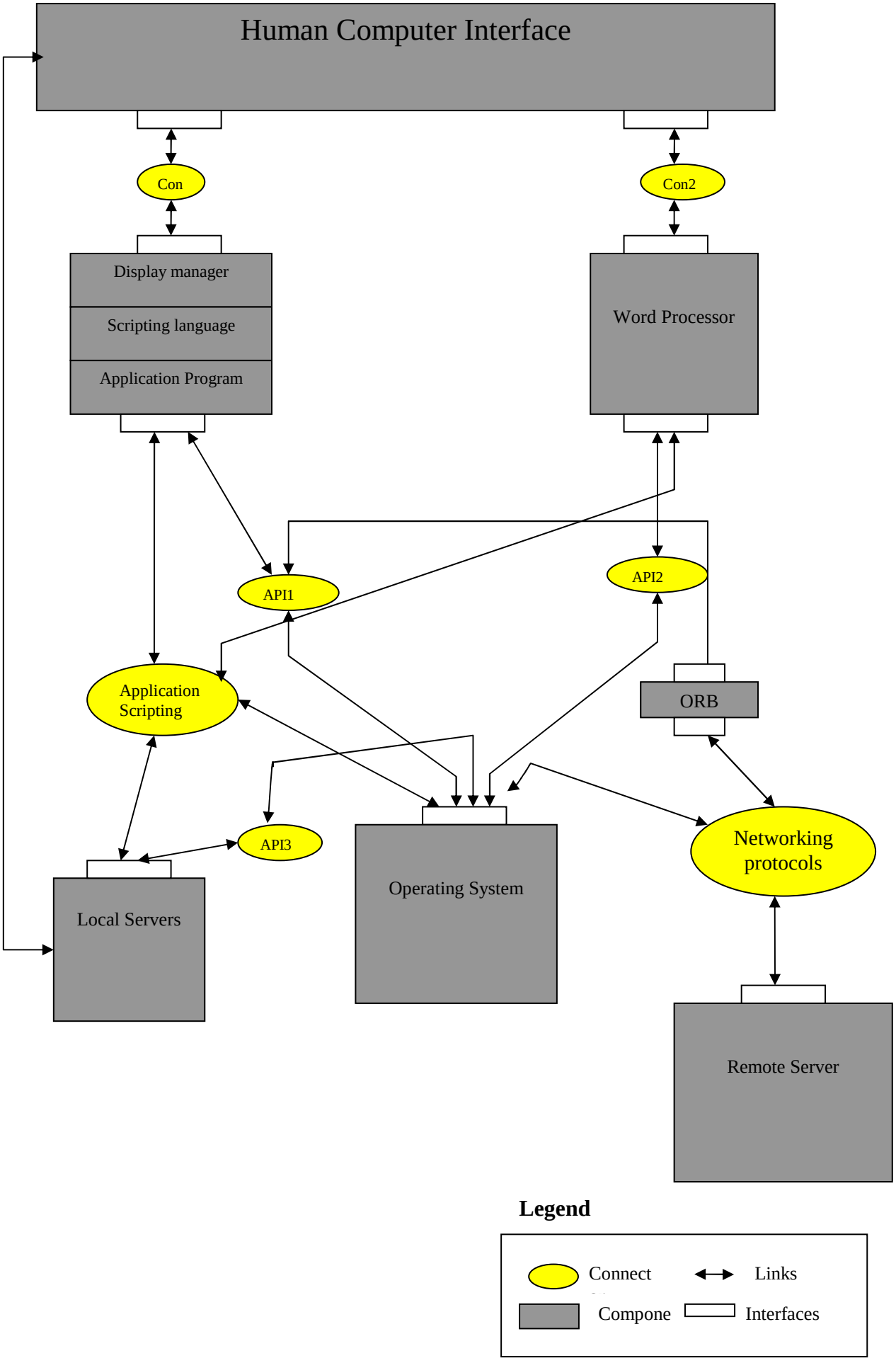
Display manager is used for displaying capability of the application.

Scripting language is used here is like Perl, TCl /Tk for generating forms etc., And Application program is like the actual web browser program. It has the code for executing HTML tags and etc., by this way we can show the design of any system using xADL.








Human Computer Interface
Con
Con2
Display manager
Scripting language
Application Program
Word Processor
API1
API2
Application Scripting
ORB
API3
Networking protocols
Local Servers
Operating System
Remote Server
Legend
Connect
Links
Compone
Interfaces