Testing Enterprise Application Architecture Patterns

By
Akber A. Choudhry

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

28th September 2006

ABSTRACT

Testing Enterprise Application Architecture Patterns

By
Akber A. Choudhry

Enterprise Application Architecture Patterns are a relatively new concept that has come of age as applications become more component-based and distributed. Object-oriented technologies and SOA are now in the mainstream of business application development. Orthogonal to these developments is the technology to centralize at the enterprise level certain specialized application functions like Caching, Configuration, Exception Handling, Logging and Security.

At the same time the need for software quality, automated software testing, and software security is ever-increasing due to shorter development cycles, agile development techniques and the sheer size of the code and the number of technologies involved.

A number of approaches to software architecture testing have been discussed by researchers and will be consulted for input into the design of the proposed framework. In this exercise, the use of xADL for prescribing patterns will be explored. Once patterns have been specified, the exploration will continue with the possible use of Schematron, ArchStudio, ArchEdit and related tools to verify an architecture against a pattern and report on issues.

Akber A. Choudhry

DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

[illegible]

Akber A. Choudhry

ACKNOWLEDGEMENTS

This work would not have been possible without the sponsorship and guidance of Dr. Osama Morad, who was the sponsor as well as the dissertation advisor.

I would also like to thank Dr. Henry Muccini for sharing his research (one of which was unpublished at the time), that lent insight into his thoughts on the subject.

Thanks are also due to the xADL team at the Institute for Software Research at the University of California, Irvine, for their excellent work on an architecture description language and the tools that they have built. The Eclipse development team and the Poseidon development team at Gentleware need to be acknowledged for their excellent tools.

For two periods in this project, I needed an escape where I could exercise total concentration and time away from the ringing phone and work burdens. The first time, the beautiful city of Agadir (Morocco) and its hospitable people provided it; the second time it was my good friend Dr. Tahir Qazi in Williamsville, New York, USA.

A big thank you to my children -- Sarah, Farhana and Mavra – for putting up with the lack of time I had for them and for being patient during this period.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 – INTRODUCTION AND BACKGROUND

THE PROBLEM

DEFINITIONS

Architecture is a relatively new field in computer software and came of age in the 1990s (Shaw and Garlan, 1996) and the authors provide a rather broad but acceptable definition of architecture and -- what makes this definition very useful in the present context -- patterns:

"Software architecture [is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns."

However, the picture becomes less clear at the enterprise level, architecture includes, among other aspects: security architecture, application architecture, security architecture, etc. The Open Group (The Open Group, 2006) defines enterprise architecture thus:

"The enterprise architecture is the all-embracing architecture for the business or organization, or a specific domain within the enterprise. The Enterprise Architecture crosses multiple systems and multiple functional groups within the enterprise. It is the design by which the organization achieves its business goals and delivers its business objectives. The technical IT architecture is a major enabling component of the Enterprise Architecture."

Regarding the testing of software systems, IEEE defines testability (IEEE, 1990) as :

"Testability. (1) degree to which a system or component facilitates the establishment of a test criteria and the performance of tests to determine whether those criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of a test criteria and the performance of tests to determine whether those criteria have been met."

RESEARCH QUESTION

Testing of architecture that binds components together is becoming increasingly important, especially with agent-based, model-based and service-oriented architecture (SOA), among others. While the theoretical work on architecture and testing is progressing, the usage of patterns in software architecture is more of an industry reality that is not specifically and comprehensively addressed by the researchers or the industry. Thus, the questions for this dissertation are:

Can existing research specifications and industry-standard testing frameworks be extended to enterprise software architecture? If so, how can we define an enterprise architecture pattern and then test an arbitrary architecture against it, using existing software and techniques? How can we apply it, and will it benefit the industry?

BACKGROUND OF THE PROBLEM

Software testing has been around almost since software development left the laboratory and started to be used for business needs. Testing started as what we now call unit testing, which has become increasingly automated and can be regression-tested (see definition in Table 1 below). As software becomes modular and components became re-usable and distributed, and recently became organized into agent-based and service-oriented designs, the testing of individual components is not enough to ensure the continued compliance of the system as a whole with requirements and the design and performance parameters.

Software architecture emerged in the 1990s to structure and document complex systems and to exploit commonalities (Muccini, 2002). Just as programming patterns evolved as techniques to solve similar problems, it was inevitable that architecture patterns would emerge, defining repeatable and tested techniques to structure a program, and provide the same at a higher level than the program or component itself.

WHAT'S NEW?

Let us take a current example. Most current information systems are being designed with increasingly patterned architecture, to the degree that entire architecture paradigms are based on patterns – for example, SOA – that allows existing or new components to be presented as services and be consumed in a distinct pattern. This represents an overriding trend: software has finally matured to the level where business requirements can be directly translated into service orchestrations. This is also known as the “business view of SOA” (IBM, 2006). In other words, software assembly is increasingly business-driven and business software components are increasingly atomic and fully re-usable. The architectural patterns and business logic that were found inside components are now increasingly being designed and defined outside the component itself, in increasingly sophisticated configurations.

Such patterned software architecture provides the flexibility and the design elegance to accommodate a variety of business applications. However, while each individual component may work flawlessly, the whole systems' assumptions cannot be tested in an automated way, or may not be testable as changes are made to the system or systems that use and share such components.

BUSINESS VALUE

Different applications with various enterprise architectures may use the same components, each architecture responding to a different business need. For example, the same back-end services may be used by many different applications, each in its own different way. Any changes to the services or their orchestration could impact the design or usage of any one of the application that uses it. Such a change may, or may not, lead to immediate problems, but it definitely causes latent loss to the enterprise as flexibility in the software life cycle is lost, and the original assumptions and tradeoffs that were part of the original architecture(s) have been lost.

It also becomes obvious that hitherto individually programmed functions like security, authentication, authorization, message transport and marshalling, networking, etc. will be kicked up to the architectural level. While this provides welcome flexibility, the enterprise architecture definition and compliance monitoring become the exclusive domain of highly-skilled architects of various disciplines (data architects, information architects, network architects and technical architects, etc.) who take on the “black-magic” role that programmers enjoyed in the early days of business software development. Consistency in description and compliance is financially beneficial for business as it increases predictability and reduces reliance on human resources. While architects may not be replaceable by robots in the foreseeable future, greater consistency can be achieved by using enterprise patterns, and greater compliance can be achieved by *TESTING* them.

Type of Test	***Goal***
Unit Test	Test programming units and algorithms
Integration Test	Test integration of components and interfaces
Regression Test	Test new changes to software components for continued functional compliance
Performance Test	Test against performance requirements
Stress Test	Testing under user and activity load
Functional Test	Test the business functionality of the system
Architecture Test	Test software architecture(s) for consistency and compliance with plan and/or patterns

Table 1: Scopes of Testing in Application Development

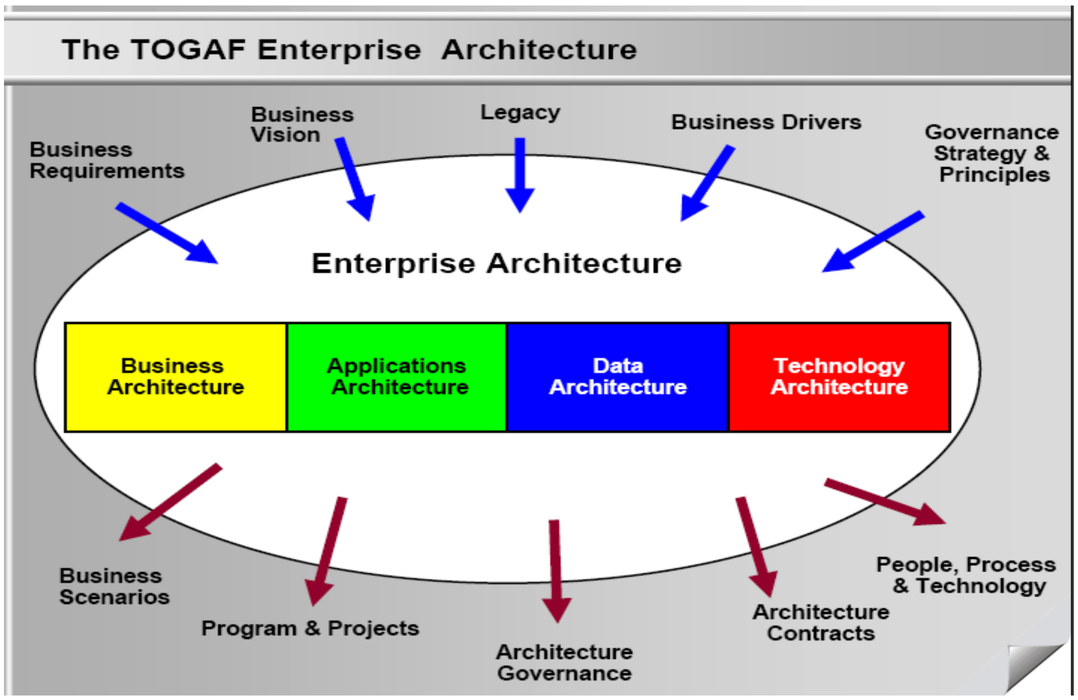


Figure 1: Enterprise Architecture according to TOGAF (The Open Group, 2006)

CHAPTER 2 – APPROACH AND REVIEW OF LITERATURE

APPROACH

Based on the problem and its background, and where architecture testing fits in to the general picture of software development, the approach to make enterprise application architectures testable, and then to test them against patterns, the EAAPT framework should have certain characteristics. These characteristics fall under two headings, in line with the dual definition of testability (IEEE, 1990). These characteristics can be best understood by thinking of architecture as a form of meta-software, just like meta-data is to data.

1. Enterprise Application Architecture must lend itself to testing:

   (a) it should be definable in a consistent language that can be represented in a machine-readable format. Due to the abstract nature of architecture and the many different ways in which it is implemented, this language must be flexible and expandable.

   (b) its definition should be linked directly to the implementation artefacts of the components, interfaces and configurations that it represents,

   (c) it should be able to depict various views of the architecture, and at various levels, in a hierarchical manner.

2. Requirements should be amenable to being defined as tests. As business requirements get translated into business architecture, functional architecture, and then into technical architecture (The Open Group, 2006), certain design guidelines are used by the architect to lower levels of detail. For the EAAPT framework, we are only concerned about the technical architecture. It is expected that over time, the higher levels of architecture will also be definable in standard notation. Let us take a usual scenario: What happens in an architect's mind when she sees a set of business requirements and after some thought and consultation, starts to scribble down some diagrams? The trained mind has detected a *pattern*, or a set of patterns that can be used to design the software. So, we can add these characteristics:

   (a) requirements, having evolved to functional and technical architecture, can be captured as pre-defined architecture templates called *pattern(s)*.

   (b) these patterns should become testable by defining rules, tolerances and other technical, logical and semantic criteria.

   (c) a set of generic *tests* (with the possibility to define pattern-specific tests) that test each facet of compliance between the pattern and the tested architecture.

3. A set of new tools, or extensions to available tools, that:

   (a) executes the generic tests against between the pattern and the architecture

   (b) are able to navigate the hierarchy of architectures, components, interfaces and configurations and re-iteratively run the tests within the appropriate scope.

(c) are able to identify a failed test and persist the information associated with the failure; and can remove that persisted information once the cause of the failure has been removed.

With these characteristics in hand, let us look at what we can find in existing literature.

REPRESENTING ARCHITECTURE: ADL(S), UML AND XML

Since software architecture is commonly understood to be the design of components, interfaces and messages that are specified for the express purpose of documentation, and implementation into an object-oriented programming language like C++ or Java – the obvious question arises, '*Why not UML?'*, as that is the notation of choice for such systems.

The short answer is: *'We are working at a supra-component level of architecture,* hence the word 'Enterprise' in the title of this paper and the framework itself. The word 'enterprise' is used to differentiate component-level architectures that define programming specifications from the concerns of how to assemble these components into a business system that works across different machines and platforms and requires security, network, runtime etc.

However, at the turn of the century, Architecture Description Languages (ADLs) were being commonly used for the notation of system architectures. As ADLs began to flourish, and UML 2.0 was introduced, this problem was realized by researchers. DUALLY (Inverardi, 2005) was one approach to create an extension to UML that would form a bridge between UML 2.0 and ADLs.

While UML allows us to design software components, UML is not there yet with respect to architectural needs. On the other hand, ADLs were proposed with arbitrary syntax and structure. Here is a sample of ACME, a popular ADL:

```
System simple_cs = {
    Component client = { Port send-request; };
    Component server = { Port receive-request; };
    Connector rpc = { Roels { caller, callee}};
    Attachments {
        client.send-request to rpc.caller;
        server.receive-request to rpc.callee;
    }
}
```

Table 2: Simple Client Server System in ACME (CMU, ACME)

Such languages worked with the familiar architectural components such as components, connectors, messages and interfaces. However, with the advent of HTML and subsequently usage of XML as a language that was readily read by machines and humans, it was only a matter of time before architecture would be represented in XML. The extensibility ('X' in XML) ensured that the language could be extended and standard XML authoring and processing tools could be used to process the architecture representations.

Such a language was xADL, developed by the Institute for Software Research (ISR) at the University of California, Irvine. (UCI-ISR, ADL). xADL was not only extensible, but also represented implementation run-time architectures in addition to the static design-time descriptions. Another effect of the extensibility was that users could use what they wanted and not understand the rest. *“Another significant benefit of xADL is that it explicitly separates the concepts of run-time (instances schema) and design-time architectures (structures and types schema). This language is not only useful for researchers, but also better suits to the needs of the practitioners, because they are not forced to learn all the features the ADL, but only those that they need.”* (Dimov, 2004)

Going back to the discussion about UML: XML ADLs can also ease the link between UML and ADLs as follows: Since UML models can be exchanged through XMI (an XML language), we can theoretically incorporate UML into XADL (see below) architectural specifications as necessary. In theory, an XADL extension can 'mount' XMI documents generated from UML as an extension and thus the specification can go deeper into the component's architecture while testing. This is possible future work that can extend

CHOICE OF ARCHITECTURE DEFINITION LANGUAGE (ADL)

When software architecture description was in its infancy, Tracz defined an ADL as consisting of four 'C's: components, connectors, configurations, and constraints (Tracz, 1993). This definition predated the ubiquity of XML but it does capture the essence of what defines an ADL. In 2000, a comprehensive classification and comparison framework for ADLs was developed (Medvidovic, 2000)

The developers of xADL have presented a substantial case for it in their exhaustive paper titled *'A Comprehensive Approach for the Development of Modular Software Architecture Description Languages'* (Dashofy, 2005a) *.* Although this paper preceded xADL, it gives us valuable criteria for a versatile ADL:

1. ability to model components, with property assertions, interfaces, and implementations
2. ability to model connectors, with protocols, property assertions, and implementations,
3. abstraction and encapsulation,
4. types and type checking, and
5. ability to accommodate analysis tools.

While the last two characteristics may be optional for other uses of ADLs, in our current need to define patterns as architecture depictions and then comparing architectures against those patterns, they are indispensable. XADL fulfils these criteria adequately and its development team also provides a nice array of tools to use with it, in addition to the general availability of XML-manipulation tools. XADL also picked up on the best features of a widely used ADL, C2 (Khare, 2001). Khare also describes the tools (such as ArchEdit) and provides a rationale of the structure of the language. I then went on to study how practical architecture languages and descriptions could be built in xADL (Aditya, 2003).

The choice of ADL for this project is xADL.

LINKING TESTING AND XADL

The need for such a framework was recognized in 2004 (Naslavsky, 2004): *“The main objective of this work is to bring together two lines of research in software architecture that are currently lacking in their integration . . . architectural analysis and testing . . . [and] . . . architecture definition languages (ADLs), specifically recent efforts toward flexible and extensible architectural representations.”*

However, the authors focused on component behaviour which is not our focus in pattern testing. In doing so, they used the same technique that I will use (see below) in extending xADL for pattern testing. This validates our approach to some degree, especially since the authors belong to the same organization as the developers of xADL and its related tools.

MUCCINI AND HIS TEAM'S WORK

Right from his Ph.D. thesis (Muccini, 2002), and continuing until now, Muccini has done extensive work on software architecture and testing using various frameworks and tools. While he does not use patterns and is limited to the component-level architecture (see distinction below), his model will come in handy as a general guideline that will be followed in trying to enable architecture pattern testing through xADL.

Our process will follow a simplified and tailored version of his model-checking driven testing schema. In our case, we will not use CHARMY.

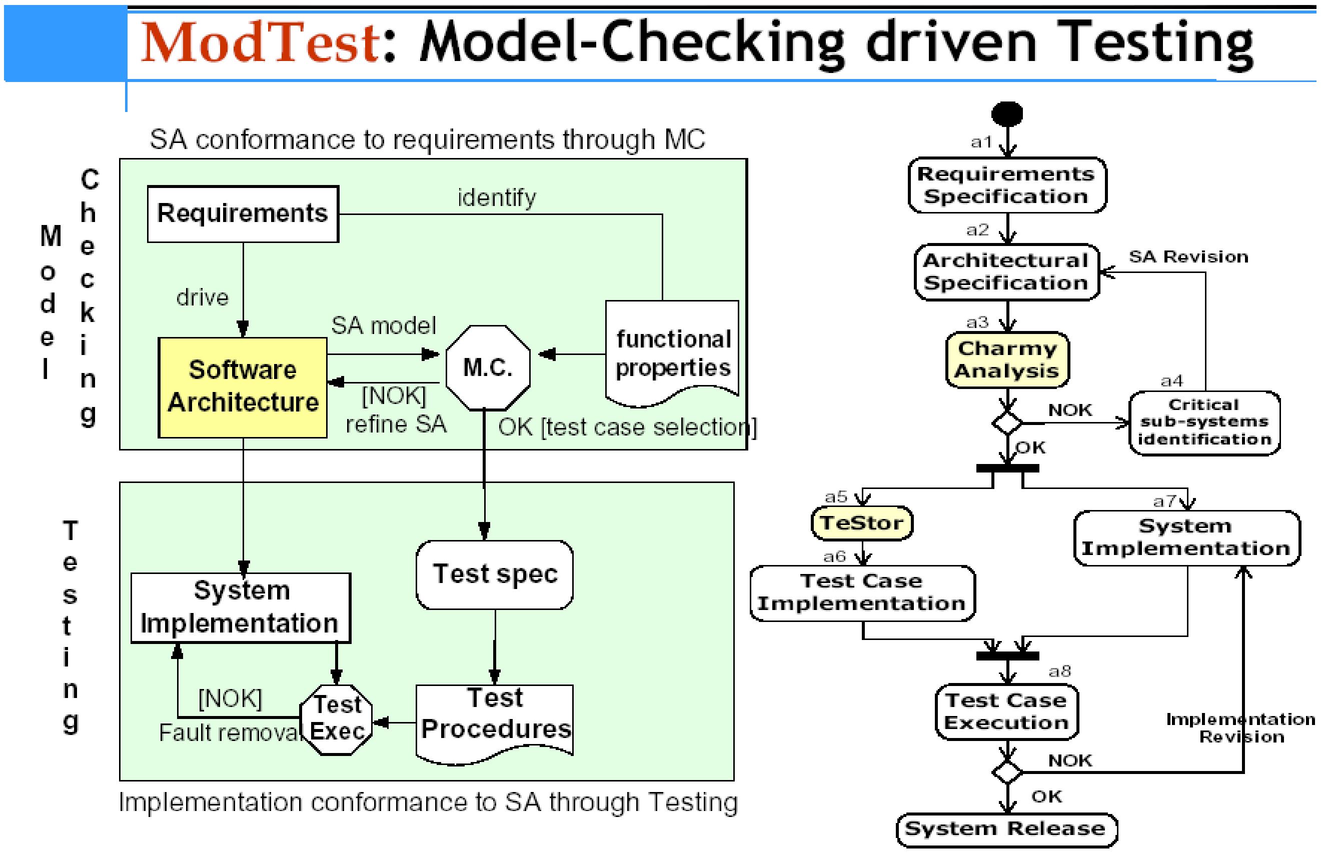


Figure 2: Muccini's model-based testing scenario (Muccini, 2005)

Inverardi, Muccini and Pelliccione constructed the CHARMY (CH ecking ARchitectural M odel consistencY) framework. They also proposed a UML-to-SA model software architecture (SA) validation system to validate an architecture against software requirements using automated tools. They used the the SPIN SA model-checking tool (http://spinroot.com/spin /whatispin.html) (Inverardi, 2001).

In 2004, other members of the same team showed the verification of middleware SA descriptions using UML-based SA representations with a theoretical hypothesis-proof model (Caporuscio, 2004).

XADL USAGE SIMILAR TO PATTERNS

Dashofy and Hoek (Dashofy, 2001) present an extension to xADL for product family architectures that uses XML Schema to extend the basic xADL definitions to represent a family of architectures with similar description. Using the *options, variants* and *versions* extensions to xADL, they define a *product family* extension that specifies an architecture that is known at a certain stage. Subsequent architectures in the family can then extend it. *"Our future work includes the development of additional XML schemas to provide an increasing set of "standard" product family architecture features, experimentation with new features for product family architectures."*

```
<xsd:complexType name="DiversityComponentType">
  <xsd:complexContent>
    <xsd:extension base="types:ComponentType">
      <xsd:sequence>
         <xsd:element name="diversity"
              type="DiversityInterface" minOccurs="1" maxOccurs="1" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Table 3: Component extension in xADL schema elements for Product Line Family

TESTING WITH ARCHITECTURE PATTERNS

Coelho et al. have discussed the benefit of architectural patterns in driving software unit and integration tests (Coelho, 2005). However, it does not mention the use of an ADL to specify the architecture. There are other papers that I will not mention here that also play discuss the feasibility and approaches to testing software architectures, or patterns. In the latter group, the patterns discussed are component-level patterns.

However, I could not find any paper that discussed enterprise-level patterns (see below) represented in a machine-readable format that is used by a framework to validate an architecture that has been specified in an ADL document.

This paper will attempt to fill this specific area.

CHAPTER 3 – DESIGN: PATTERN REPRESENTATION IN XADL

ENTERPRISE-LEVEL DESIGN PATTERNS

“In recent years, the focus of software development has progressively shifted upward, in the direction of the abstract level of architecture specification.” (Grassi, 2005)

Computer chips, firmware, operating system and even hardware systems have become increasingly commoditized and serve many common purposes for many different applications. With languages like C++ (available on different platforms) and Java (runnable on different platforms, and the ubiquitous TCP/IP network, the exact hardware, network and operating system rarely matter when making decisions about business applications.

In a similar fashion, software components like databases, shared libraries and graphical controls have made commodities out of software components. This is most evident in components developed in object-oriented languages. The promise of re-usability may finally be here. Late binding, especially through a service lookup pattern have made the re-use and independent lifecycle of components feasible.

So, we will define enterprise-level patterns as inter-component patterns. Software architecture at the enterprise level is gaining importance as software development becomes more and more componentised. Software innovations and higher network and processing speeds have created a trend to re-use existing components (which may have been developed in different languages) in ever-different loosely-coupled variations and to drive application architecture through modelling (MDA). A simple explanation: instead of having a software for each business function, software can be composed from many smaller components and deployed as an application.

Such patterned software architecture provide the flexibility and the design elegance to accommodate a variety of business applications. Testing of architecture that binds components together is becoming increasingly important and is going to preserve investment in Service-Oriented Architecture (SOA) and Agent-based software development -- as well as their ongoing support. In SOA, components are exposed through a published interface and are generally accessed using HTTP using markup languages such as XML and its derivatives. The orchestration of these services through a driver is the application *itself.*

As an example of such a heterogeneous system made of loosely-couple components, we will take SOA as an example.

CONCERNS WITH TESTING SOA ARCHITECTURE PATTERNS

With this scenario, three concerns are immediately visible:

1. While each individual component may work flawlessly, the whole system's architecture cannot be tested in an automated way. Traditional component-testing methods use unit testing and code and style analysis (white-box and black-box testing)

to test a component's function and source code and arrive at a limited validation of the component's architecture. In heterogeneous environments the inter-component architecture is not amenable to either technique.

2. A single test of all components may not be enough as changes are made to the system and the components. Components may participate in the orchestration of various applications and be bound by different transports. Each system would have to be tested individually to make sure that its architecture still complies with what was intended and that it matches up to a pattern.

3. What is to be tested may change from code to architecture description. Techniques used in unit, integration and regression testing that used source code and compiled programs to carry out their testing will no longer be feasible in this heterogeneous environment. As discussed above, only a representation of the architecture will be tested and the assumption made that the actual implementation mirrors the architecture specification. A tool that validates the architectural specification and compares it against a pattern can be extended to check for implementation fidelity as well.

The pre-defined top-level type in an xADL architecture document is the architecture structure (ArchStructure), this is the level at which a SOA architecture pattern will be defined.

```
<xsd:complexType name="ArchStructure">
  <xsd:sequence>
    <xsd:element name="description" type="archinstance:Description"/>
    <xsd:element name="component"   type="Component"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="connector"   type="Connector"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="link"        type="Link"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="group"       type="archinstance:Group"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="id"          type="archinstance:Identifier"/>
</xsd:complexType>
```

Table 4: xADL ArchStructure Type and its Constituents

COMPONENT-LEVEL DESIGN PATTERNS

In Java and other object-oriented languages, design patterns have been around for quite some time. While sometimes referred to as architectural patterns, they are not enterprise-level architecture patterns for the purpose of this paper.

In xADL, components are the pre-defined units that make up an architecture, together with connections, interfaces and others. In the implementation extension to xADL, the implementation can be defined, such as a Java implementation.

```
<xsd:complexType name="Implementation" abstract="true"/>
  . . .
<xsd:complexType name="JavaImplementation">
    <xsd:complexContent>
      <xsd:extension base="archimpl:Implementation">
        <xsd:sequence>
          <xsd:element name="mainClass" type="JavaClassFile"/>
          <xsd:element name="auxClass"  type="JavaClassFile"
                       minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

Table 5: Implemenation Extensions to xADL showing Component Class Names

A pattern specification will be attached at the component implementation level for component-level pattern. Since Java implementation is one of the possible implementations, a pattern should be specifiable at the abstract implementation level or the concrete language-specific implementation level.

In order to differentiate component-level design patterns from enterprise-level pattern defined above (for example, a certain SOA pattern), here is a table that shows a sample of component-level design patterns, retrieved from Martin Fowler's design pattern catalogue (Fowler, 2003) :

Table 6: Some Component Design Pattern Groupings

Pattern Group	***Pattern***	***Brief Description***
Domain Logic	Transaction Script	Application as a series of transactions
	Domain Model	Objects mirroring business entities
	Table Module	Domain Model with one class per table
	Service Layer	Provides interfaces to other applications and hides implementation and data sources
Data Source Architectural	Table Data Gateway	Method-to-SQL DML mapping for a table (O-R)
	Data Mapper	Maps complex data structure persistence and hides implementation from objects
Web Presentation	Model View Controller	A controller acts as a dispatcher of requests to the model which executes and passes to the view and back to the user.
	Application Controller	A variation on the above
Distribution	Remote Facade	represents a coarse-grained interface to one or several fine-grained objects.

It can be readily seen that the last two pattern groups in the table above are on the border-line between component- and enterprise-level patterns. The distinction that we will employ is that a multi-component design pattern does not necessarily mean that it is an inter-component (enterprise-level) pattern. So, a Web application pattern that results in a controller class, several model classes and JSP (Java Server Pages) pages does not necessarily become an enterprise pattern, as the various components are in the same language (Java) and are tightly coupled (direct calls within a JVM – Java Virtual Machine – or calls between containers using a vendor's proprietary protocol). It is not likely that any of the components will be used during run-time by another application.

With this definition, this distinction is not problematic and xADL can accommodate it by adhering to the following rules (ignore the specialized *pattern* types and elements: they are discussed in the next chapter):

1. An enterprise-level pattern will be referenced in an xADL architecture document in the ArchStructure or ArchInstance types (architecture instance not implemented in this project).

```
<types:archStructure types:id="ChatSystem"
             xsi:type="pattern:PatternedArchStructureType">
   <pattern:pattern>
     <pattern:patternExternal xlink:href="pattern1.xml"
             xlink:type="simple"/>
    </pattern:pattern>
```

2. A component-level pattern will be referenced in an xADL architecture document in the component, connector or interface types.

```
<types:componentType types:id="Server_type"
         xsi:type="pattern:PatternedComponentType">
     <pattern:pattern>
         <pattern:patternExternal
                  xlink:href="pattern2.xml" xlink:type="simple"/>
      </pattern:pattern>
. . .
```

3. Multi-component patterns, as desribed above, can use the same pattern instance for all components. The tool that checks the pattern can then make sure that the interfaces and messages between the components are consistent.

```
<types:componentType types:id="Server_type"
         xsi:type="pattern:PatternedComponentType">
     <pattern:pattern>
         <pattern:patternExternal
                  xlink:href="pattern3.xml#server" xlink:type="simple"/>
      </pattern:pattern>
. . .
<types:componentType types:id="Bus_type"
         xsi:type="pattern:PatternedConnectorType">
     <pattern:pattern>
         <pattern:patternExternal
                  xlink:href="pattern3.xml#bus" xlink:type="simple"/>
      </pattern:pattern>
```

CHAPTER 4 – XADL EXTENSIONS

We have chosen xADL to be the architecture description language and then shown briefly how patterns will be implemented in xADL types that have been specially extended from base types or other extensions in order to accommodate a linked pattern.

XADL EXTENSION GUIDELINES

Extension of xADL to achieve a new architectural purpose is by design and not to make up for any shortcoming. Although the method to extend is compliant with the way XML schemas are extended, there is a specific way (UCI-ISR, Way) to do this and it is copied here:

1. Identify discrepancies between what xADL 2.0 can model in its set of provided schemas and what you want to model for your project.
2. Decide which xADL 2.0 elements to extend to support your new modeling (sic) needs—do you need to add new kinds of data to structural elements like components and connectors? Their types? Or do you need new kinds of elements altogether?
3. Decide how to syntactically encode these new extensions or elements.
4. Write new XML schemas extending xADL 2.0 to add your new modeling constructs. Validate your new schemas with a tool like XSV.
5. Run Apigen (sic) to generate a new data binding library that includes your extensions.
6. Use syntax-based tools like ArchStudio 3's ArchEdit to extend existing architecture descriptions with new kinds of data in the format you specified.
7. Extend semantics based tools like ArchStudio 3's Archipelago and Tron to provide friendlier editors and analysis tools for your new notational extensions

ANALYSIS OF XADL PROVIDED SCHEMAS

This is a summary of the schema strategy of xADL designers that they have provided in the core and extension schemas, which were analysed from the perspective of adding the pattern capability:

1. *variants.xsd*: The basic types in *types.xsd* are extended to allow variants (variant elements). A *Variant* element is introduced with an *options:Guard* element. This allows multiple architectures for a component or connector to be chosen based on the guard. Interfaces cannot be variants for obvious reasons. This is functionality that is desirable for patterns, as a choice of component architecture may be required in complying with an enterprise pattern.

2. *implementation.xsd*: This schema defines an abstract *Implementation* type that will be extended by language-specific implementations such as *JavaImplementation.* It defines *InterfaceTypeImpl* (descending directly from the base type), *VariantComponentTypeImpl* and *VariantConnectorTypeImpl* (both descending from the *Variant* types. This schema is definitely needed by the pattern design as multiple

implementations can have different implementations. For examples, a certain service may be implemented in Java or in C++, or a component may be implemented using two different component-level patterns.

3. *versions.xsd:* Versions are orthogonal to actual architecture structure and pattern analysis will only include one version at a time. Versioned types are descended from *Variant* and *Implementation* types, but that is not necessary. The author of this schema writes in the schema comments: *"The actual dependency set of versioning is just types, but because of XSchema 1.0's single inheritance limitation, we decided to have it depend on implementation and variants as well. When XSchema 2.0 is released, supporting multiple inheritance, this dependency should be removed."* This is not needed as a parent to the pattern schema (if a schema is required, see below).
4. *messages.xsd:* This schema provides for a *MessageCausalitySpecification* element that encapsulates rules and messages that are passed between components in an architecture. A lot of thought went into deciding whether this schema was required in patterns, but it was ultimately rejected as it should not play a role in enterprise-level architecture patterns. Since messages between components can be considered as data flowing across interfaces, *messages.xsd* does not define an Interface type. Since interfaces will be patterned, it is unnecessary to pattern message formats. Pattern compliance does not include run-time data or event compliance, since we are not patterning instances of the architecture: that has been left as a future enhancement to this framework (see Chapter on Future Work). Also, this schema inherits from the versioned types and we have excluded that extension earlier.
5. *security.xsd:* Security specifications are both for architecture structure and instances, and like versions, are orthogonal to the architecture itself. While security can be part of patterns, it is left to be decided in the future work when architecture instances are incorporated into the pattern-testing framework.
6. *diff.xsd and pladiff.xsd:* These schemas define elements that show the result of a 'diff' operation on two schemas. Since pattern analysis will involve a similar process, the project design called for extending one of these schemas to show discrepancies after the pattern testing is done. *pladiff.xsd* schema is sufficient to hold the diff'ed elements for the simple comparison process that is in this project and it allows for optional elements in its structure.

HIERARCHICAL PATTERNS AND CONFIGURATION MANAGEMENT

One of the design goals for this framework was to be able to specify patterns in hierarchies. XML schema inheritance automatically achieves this goal as patterns can be specified at the architecture, component, connector or interface levels. A note of caution is: as we inherit from *Variants,* direct *SubArchitecture* will not be possible for any patterned element.

A pattern element will thus point to another element of the same type (as a reference to compare against) or to an external pattern sitting in its own file. To incorporate this framework into build systems and IDEs, a modification can be done whereby *javasourcecode.xsd,* *sourcecodeeclipse.xsd* and *lookupimplementation.xsd* can be used to point to component projects within an IDE like Eclipse or a lookup repository.

In this way, a configuration manager (build system maintainer) does not have to mirror project dependencies and relationships in a repository to the architecture document(s) and the applicable pattern element or document may be looked up from a repository based on the relationships that already exist between the implementations of the architecture (code).

XADL EXTENSION STRATEGY – PATTERNED TYPES

Since architecture patterns are not requirement features and neither are they product lines, we need to define a constraints system similar to PLA (Product Line Architecture) but with pattern-specific needs in mind. The xADL systems supports syntactic and semantic tools (Dashofy, 2005) and the pattern extension will be such as to fit with both. Architecture descriptions of software systems are generally composed of at least three key entities: components, connectors, and configurations. *Configurations describe how components and connectors are arranged, . . . it may include constraints or patterns on arrangements of components and connectors, or how they behave (Dashofy, 2005)*. The designers of xADL did not “dictate how they [components and connectors] must behave or how they can be linked together” (Dashofy, 2005) and recommended that this be done in extensions.

XADL contains design-time and run-time views that have been separated by the designers of xADL. By adding the pattern extension elements to the top-level element of each, we can define that the specific architecture adheres to a specific pattern – the pattern itself being an architectural specification. xADL tools contain the Critics Framework (now known as Tron) that allows the consistency analysis of the architecture document. By following xADL tool design, the patterns constraint can be checked using the Tron framework (in addition to the Eclipse TPTP interface as future work, described below).

Based on the discussion above, pattern references can be implemented as extension elements or existing elements can be used.

One of the existing element that was considered was the *Group* element in the *ArchInstance* type. The author writes in the schema comments: *“The Group type describes a logical grouping of architecture elements (components, connectors, and links). Groups may also be members of other groups. As such, the member XLink may point at a component instance, a connector instance, a link instance, or another group.”*

```
<xsd:complexType name="Group">
  <xsd:sequence>
    <xsd:element name="description" type="Description"/>
    <xsd:element name="member" type="XMLLink"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="id"          type="Identifier"/>
</xsd:complexType>
```

Table 7: Group Element Definition of xADL

We can use the G*roup* element to bind an architecture or sub-architecture to a pattern, but this would not be following the spirit of xADL design as well as a re-use of an existing feature for which it is not intended. This approach was dropped.

This leaves us with the option to define a *pattern.xsd* schema with its own types:

```
<!--
    TYPE: PatternedArchStructureType
    This element contains a pointer to the architecture
-->
<xsd:complexType name="PatternedArchStructureType">
  <xsd:complexContent>
   <xsd:extension base="types:ArchStructure">
      <xsd:sequence>
        <xsd:element name="pattern" type="Pattern" minOccurs="0" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
```

Table 8: PatternedArchStructureType - a Sample Patterned Element

The patterned component, connector and interface types are listed in the complete *pattern.xsd* schema in the Appendix. Only the *PatternedArchStructureType* extends the base *ArchStructure* type, the others extend *Variant...Impl* types (see above).

PATTERN TYPE

The extended types defined above include a *Pattern* element. In the *Pattern* type below, a pattern can reference an existing architecture structure, component, connector or interface. The referenced element need *not* be a PatternedElement. An external reference is to a an external file containing an architecture that should be used as the pattern.

```
<!--
    TYPE: Pattern
    This type describes a pattern.
-->
<xsd:complexType name="Pattern">
  <xsd:choice>
    <xsd:element name="patternArch" type="types:ArchStructure"
         minOccurs="0" maxOccurs="1"/>
    <xsd:element name="patternComp" type="types:ComponentType"
         minOccurs="0" maxOccurs="1"/>
    <xsd:element name="patternConn" type="types:ConnectorType"
         minOccurs="0" maxOccurs="1"/>
    <xsd:element name="patternInt" type="types:InterfaceType"
         minOccurs="0" maxOccurs="1"/>
    <xsd:element name="patternExternal" type="archinstance:XMLLink"
         minOccurs="0" maxOccurs="unbounded"/>
  </xsd:choice>
</xsd:complexType>
```

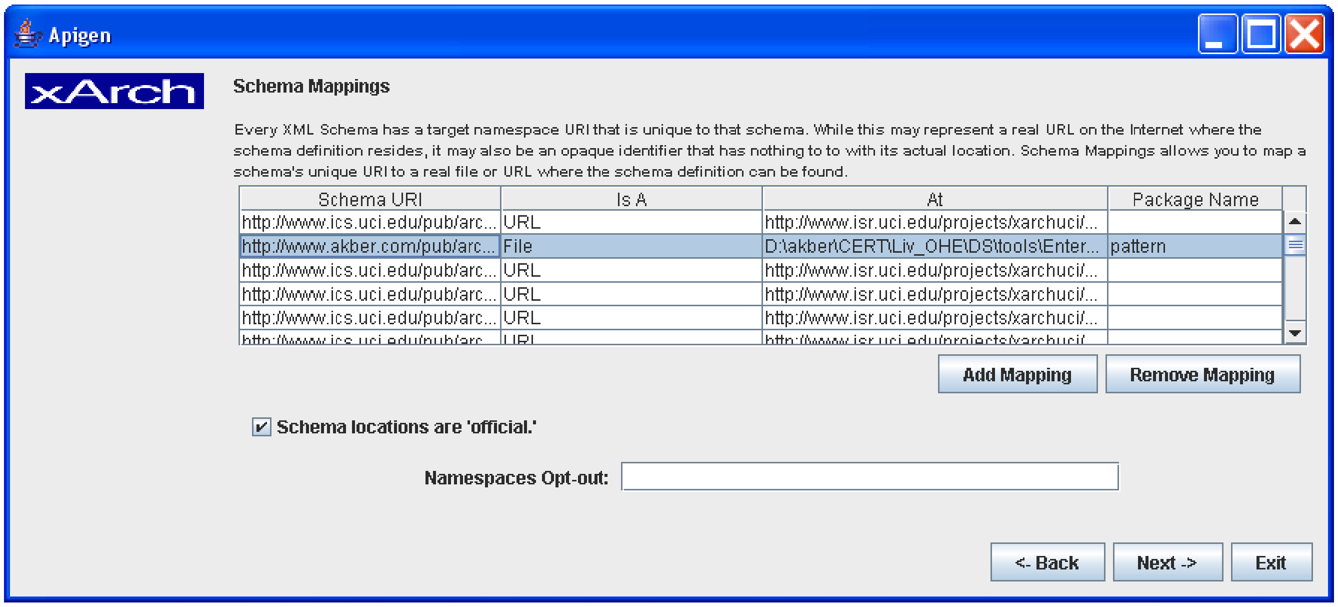
Table 9: Pattern Element that Defines a Referenced Pattern

Since XADL has been designed to be flexible and extensible, the pattern schema *pattern.xsd* will inherit the flexibility and extensibility and can be further extended.

XADL DOCUMENT-PROCESSING CODE

This code was generated through the APIGEN utility provided by the xADL team: *"Apigen is an XML schema-to-Java data binding generator for xArch schemas that automatically generates those libraries based only on the XML schemas. It supports a large subset of the XML schema language."*

The generated and manual code will also read in information contained in our extensions to XADL – the pattern and testing extensions.



Schema Mappings

Every XML Schema has a target namespace URI that is unique to that schema. While this may represent a real URL on the Internet where the schema definition resides, it may also be an opaque identifier that has nothing to to with its actual location. Schema Mappings allows you to map a schema's unique URI to a real file or URL where the schema definition can be found.

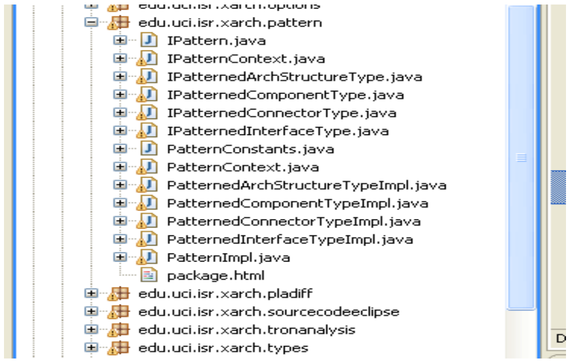
Schema URI	Is A	At	Package Name
http://www.ics.uci.edu/pub/arc...	URL	http://www.isr.uci.edu/projects/xarchuci/...	
http://www.akber.com/pub/arc...	File	D:\akber\CERT\Liv_OHE\DS\tools\Enter...	pattern
http://www.ics.uci.edu/pub/arc...	URL	http://www.isr.uci.edu/projects/xarchuci/...	
http://www.ics.uci.edu/pub/arc...	URL	http://www.isr.uci.edu/projects/xarchuci/...	
http://www.ics.uci.edu/pub/arc...	URL	http://www.isr.uci.edu/projects/xarchuci/...	
http://www.ics.uci.edu/pub/arc	URL	http://www.isr.uci.edu/projects/xarchuci/	

[✓] Schema locations are 'official.'

Namespaces Opt-out:

Figure 3: Using APIGEN to create xArch elements for pattern.xsd

These are the new Java classes generated. APIGEN is an alpha-version tool and the code was checked and tested in Poseidon to make sure that the interfaces were implemented and that the inheritance was as expected. (see UML in Appendix).



```
edu.uci.isr.xarch.options
edu.uci.isr.xarch.pattern
    IPattern.java
    IPatternContext.java
    IPatternedArchStructureType.java
    IPatternedComponentType.java
    IPatternedConnectorType.java
    IPatternedInterfaceType.java
    PatternConstants.java
    PatternContext.java
    PatternedArchStructureTypeImpl.java
    PatternedComponentTypeImpl.java
    PatternedConnectorTypeImpl.java
    PatternedInterfaceTypeImpl.java
    PatternImpl.java
    package.html
edu.uci.isr.xarch.pladiff
edu.uci.isr.xarch.sourcecodeeclipse
edu.uci.isr.xarch.tronanalysis
edu.uci.isr.xarch.types
```

Figure 4: Java Classes Generated by APIGEN

CHAPTER 5 – ENTERPRISE PATTERN TESTING

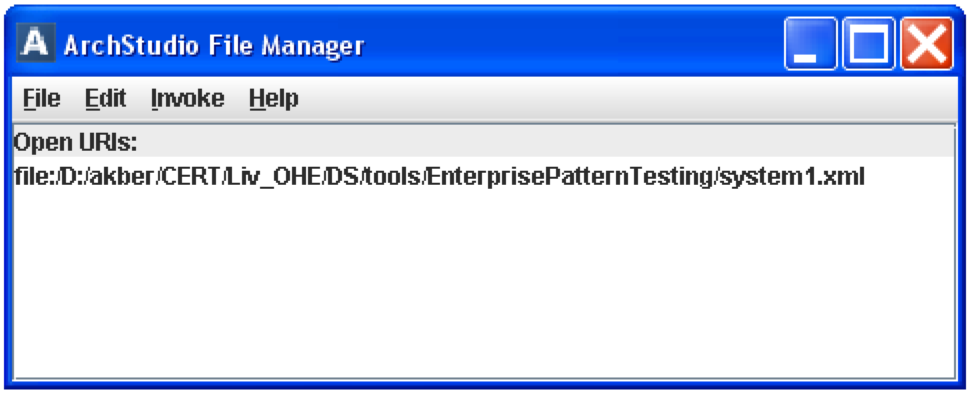
The project design called for a custom stand-alone component to act as the test driver, together with an Eclipse (a popular IDE) TPTP (Eclipse Test & Performance Tools Platform) driver. During the course of the project, I learnt to use ArchStudio 3.0, a modular architecture editing tool. ArchStudio is being integrated with Eclipse, and it would be unproductive to write a custom TPTP interface to run pattern analysis.

Armed with our new pattern schema and its hooks into the xADL framework, let us now use it to attach patterns to an architecture.

APPLYING PATTERNS TO AN ARCHITECTURE

First of all, the Schema API library that ArchStudio uses has to be replaced with the newly generated API. The configuration and startup scripts for ArchStudio have to be modified to accommodate the newly added schemas and packages.

Then, we load an architecture XML document file in which one of the types has been changed to *PatternedComponentType:*



Open URIs:
file:/D:/akber/CERT/Liv_OHE/DS/tools/EnterprisePatternTesting/system1.xml

Figure 5: Loading a Test Architecture into ArchStudio

Voila! We can add a Pattern element.

We then set the pattern file name as this is an externally defined pattern.

This is the editor screen:

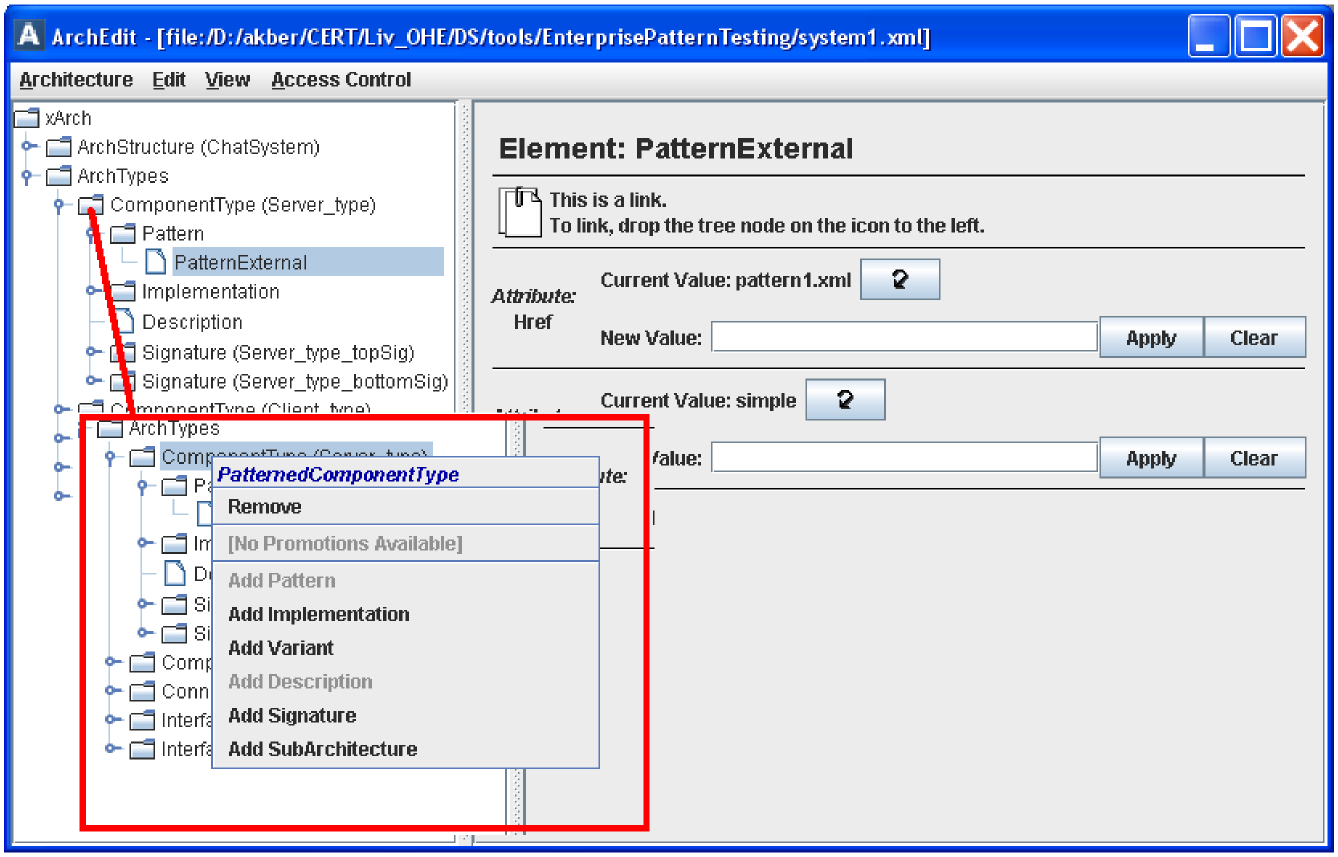

ArchEdit - [file:/D:/akber/CERT/Liv_OHE/DS/tools/EnterprisePatternTesting/system1.xml]
Architecture Edit View Access Control
xArch
ArchStructure (ChatSystem)
ArchTypes
ComponentType (Server_type)
Pattern
PatternExternal
Implementation
Description
Signature (Server_type_topSig)
Signature (Server_type_bottomSig)
Element: PatternExternal
This is a link.
To link, drop the tree node on the icon to the left.
Attribute: Href
Current Value: pattern1.xml
New Value:
Apply
Clear
Current Value: simple
PatternedComponentType
Remove
[No Promotions Available]
Add Pattern
Add Implementation
Add Variant
Add Description
Add Signature
Add SubArchitecture

Figure 6: Adding a Pattern Element in ArchStudio Editor

ARCHSTUDIO, CRITICS AND TRON

XADL has a tool that can be used for this purpose that hooks into ArchStudio, named Tron (UCI-ISR, Tron). It replaces the old "critics" infrastructure of ArchStudio. Critics ran continuously and were custom-written in Java. Tron runs on demand and uses Schematron (Schematron) to evaluate the XML for patterns and this is the core of the framework. Schematron is an XSL-based rules-based schema validator that can incorporate a wide variety of structural, comparison and transformation tests on XML documents with more reach than XML Schema.

With the built-in Schematron engine within Tron, we will write a custom Schematron rule set that that will take the patterned architecture and determine whether:

1. a pattern has been used
2. whether the pattern is shared across components, signifying a multi-component pattern.
3. the architecture conforms to the pattern.

As of the writing of this paper, Tron does not have Schematron-based *diff* or *pladiff* rule sets, from which we could extend and use the conformance option. However, the other two options were tested.

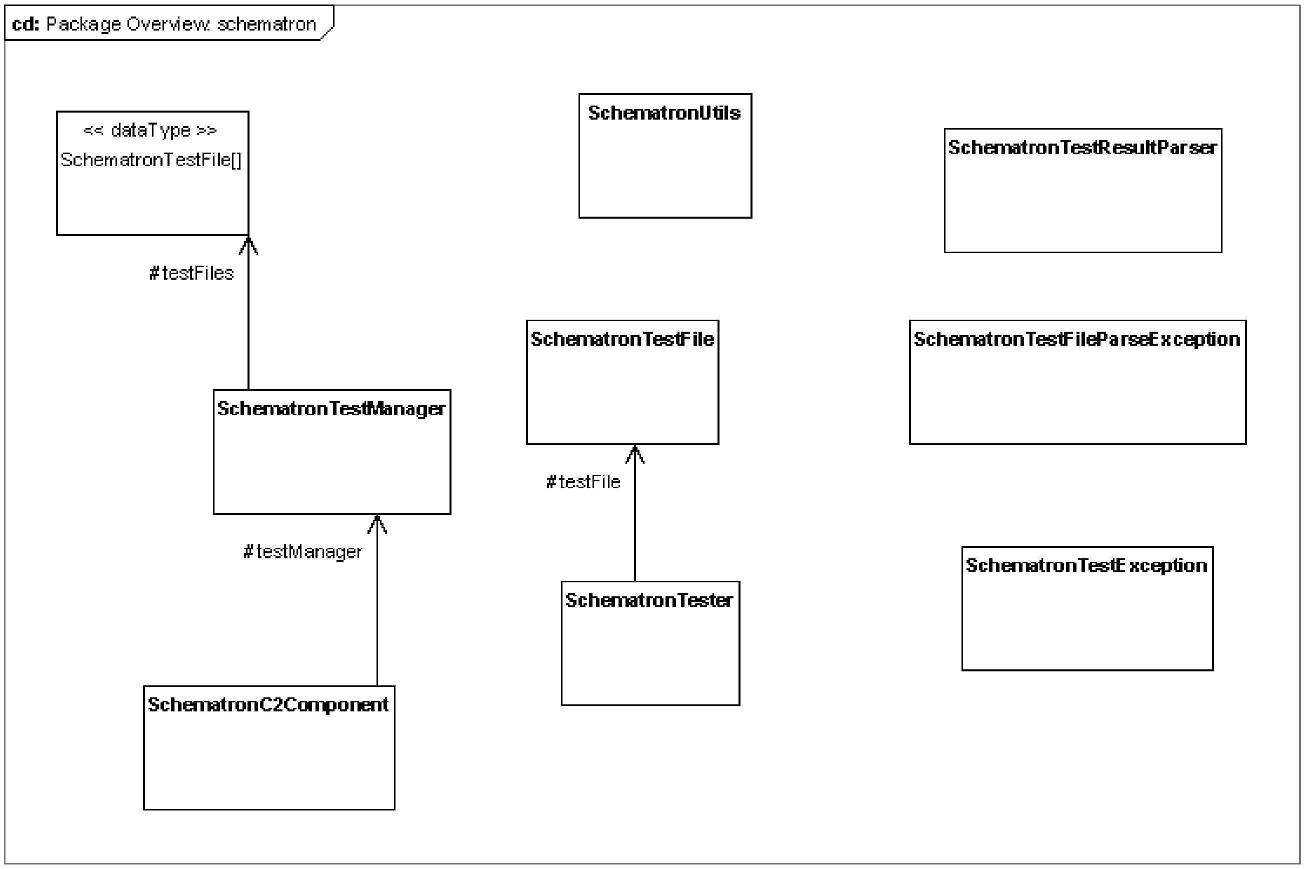
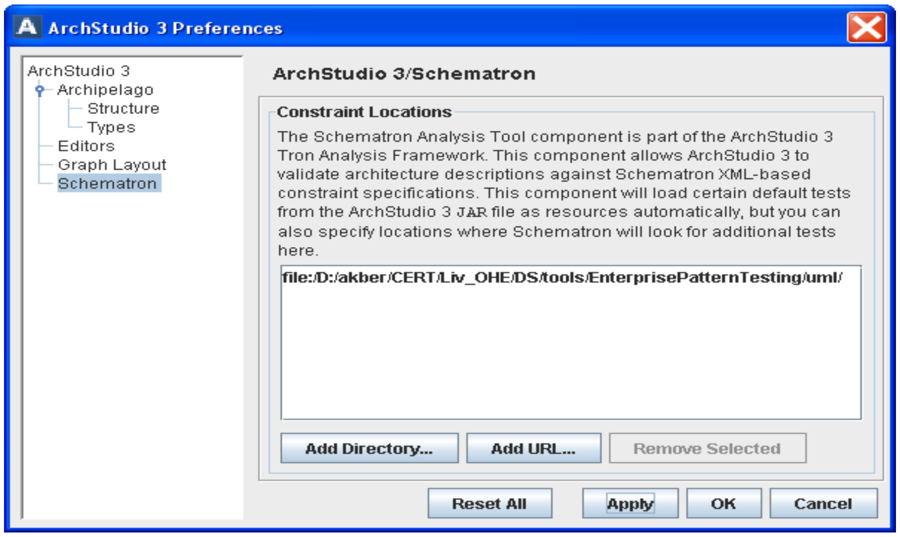


Figure 7: Existing ArchStudio Tron Schematron Driver design

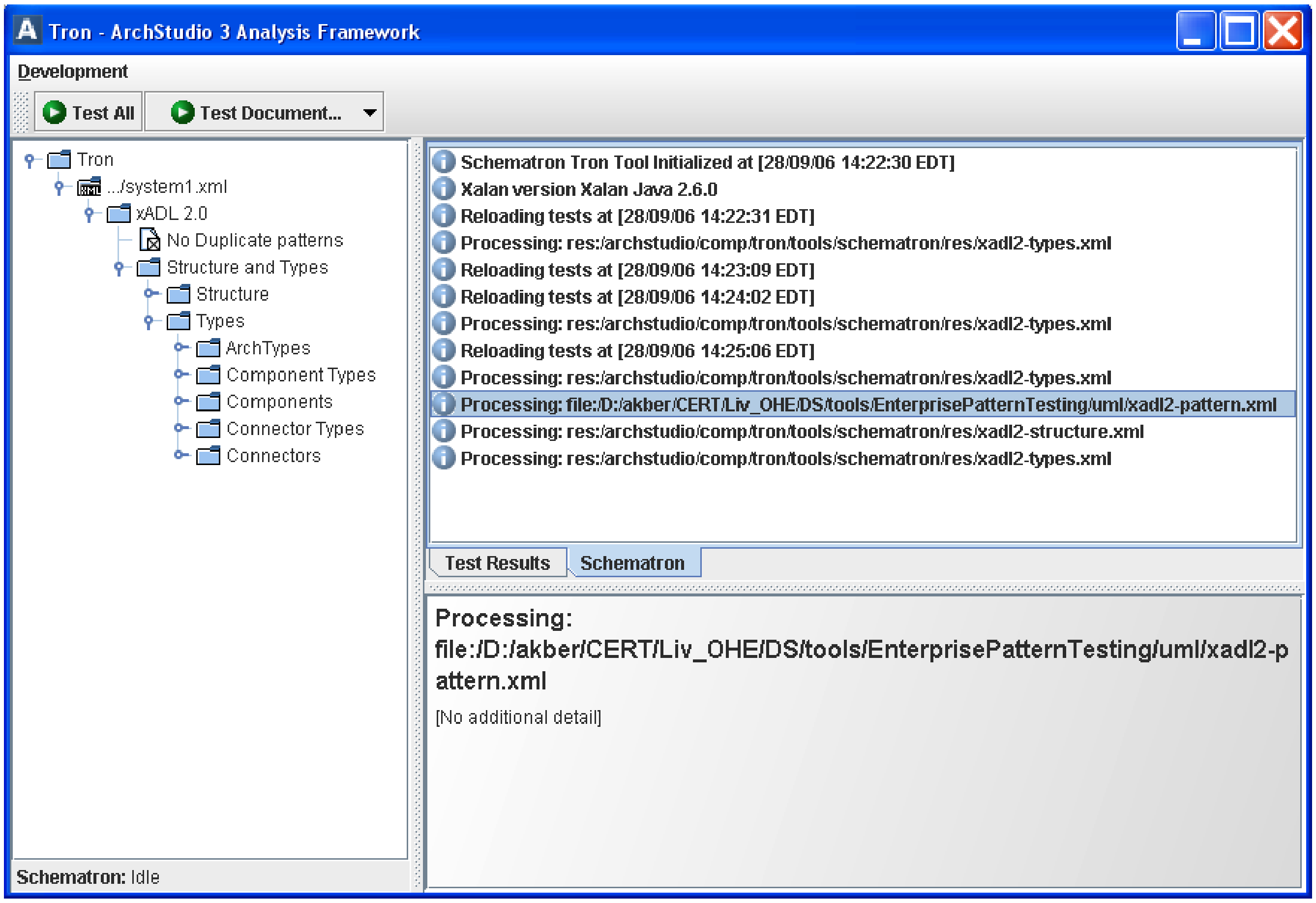
The pattern rule set was loaded into the Schematron engine:



The Schematron Analysis Tool component is part of the ArchStudio 3 Tron Analysis Framework. This component allows ArchStudio 3 to validate architecture descriptions against Schematron XML-based constraint specifications. This component will load certain default tests from the ArchStudio 3 JAR file as resources automatically, but you can also specify locations where Schematron will look for additional tests here.

```
file:/D:/akber/CERT/Liv_OHE/DS/tools/EnterprisePatternTesting/uml/
```

Figure 8: Loading Schematron Rule Sets



```
Schematron Tron Tool Initialized at [28/09/06 14:22:30 EDT]
Xalan version Xalan Java 2.6.0
Reloading tests at [28/09/06 14:22:31 EDT]
Processing: res:/archstudio/comp/tron/tools/schematron/res/xadl2-types.xml
Reloading tests at [28/09/06 14:23:09 EDT]
Reloading tests at [28/09/06 14:24:02 EDT]
Processing: res:/archstudio/comp/tron/tools/schematron/res/xadl2-types.xml
Reloading tests at [28/09/06 14:25:06 EDT]
Processing: res:/archstudio/comp/tron/tools/schematron/res/xadl2-types.xml
Processing: file:/D:/akber/CERT/Liv_OHE/DS/tools/EnterprisePatternTesting/uml/xadl2-pattern.xml
Processing: res:/archstudio/comp/tron/tools/schematron/res/xadl2-structure.xml
Processing: res:/archstudio/comp/tron/tools/schematron/res/xadl2-types.xml
```

```
Processing:
file:/D:/akber/CERT/Liv_OHE/DS/tools/EnterprisePatternTesting/uml/xadl2-pattern.xml
[No additional detail]
```

Figure 9: Running the Pattern Schematron Tests

Errors could not be produced as any error in the file would not allow it to be loaded as the *pattern.xsd* schema is already very restrictive. The simple pattern tests (in options 1 and

2 above) passed. When multiple file tests (pattern matching) are commissioned (option 3 above) then the initial file would be loaded without problems and would produce problems when matching against an external pattern file.

<u>WORKING PROTOTYPE DESIGN FEATURES AND SUCCESS</u>

With the pattern schema well in place in xADL and the Schematron engine working fine, any number of tests can be created and executed. The sophistication and complexity of patterns and associated tests are only limited by the constraints of the complexity of architectures allowed under xADL.

The architecture documents can also use versions, multiple implementations and security (among all the features of xADL) and the pattern testing should still work.

Users of this framework can extend this to suit their needs. In this prototype, we only used externally defined patterns. The framework allows (in the elements inside the Pattern element of the *pattern.xsd* schema) for a reference patterned type to be created within the architecture document itself and thus be an ad-hoc pattern. For example, an interface with a certain characteristic can be defined as a type within the document and other types can refer to it using the *patternInt* element to point to the pattern element. Apart from the pattern funcationality, this can also form a sort of rule-template that can be used to augment external Schematron rules that are checked by Tron in ArchStudio.

CHAPTER 6– ANALYSIS

This is the process that was followed in this project:

1. Determine the need for enterprise application architecture testing and the state of research in the area.
2. Pick an ADL that could be used to define architectures in a way that tests could be performed on it. Ideally, that ADL would let us define patterns as architectures, avoiding the need to invent a pattern language. XADL fitted the bill perfectly.
3. Analyze xADL schemas and carefully fit in the extensions into the xADL framework.
4. Use tools to bind the schema to the tools, thereby extending the tools provided by xADL.
5. Build a prototype and test it within the limitations of the tools chosen and the scope of the project.

Referring back to Muccini's testing process diagram (Figure 2), we can see that our revised process is something like this:

1. Define xADL testable architecture and attach it to pattern elements (components, interfaces, connectors or *ArchStructures*) or to an external pattern architecture.
2. Use standard Schematron rules to validate internal consistency of the patterned architecture. Write custom Schematron rules to validate the architecture against the pattern. Use the ArchStudio Tron tool to run the architecture and the pattern through the Schematron validation engine.
3. Use the issues generated by Tron to see where the discrepancies are, and resolve them.
4. Repeat the above process with each change in the architecture or the pattern. Script it or bind it to a build process to enable a form of regression testing.

CHAPTER 7 – CONCLUSIONS

Our design has been validated by testing and a viable enterprise application architecture testing framework has been developed.

The classification of design patterns into component-level, multi-component and enterprise-level enabled us to map the patterned architectural elements at the right level and to have the potential ability to design sophisticated pattern tests. As software development tends to 'rise' up to the architecture level, this classification of design patterns becomes necessary. While I have used the terms that I have defined, they may be defined differently by the industry as the need for this distinction grows.

The scope of this project would not have been possible without the excellent tools from the xADL team. While the tools are still buggy and incomplete, they were a great help, and helped to focus effort on analysis and research instead of re-inventing the wheel.

During the writing of this paper, Rational (a division of IBM) publicized SysML (Rational, 2006) as a system modelling language. While going through SysML, it was discovered that. It bridges the gap between requirements, design and code but it is not an enterprise-level language. This shows the gap between research projects like xADL and the commercial industry at this time. In my opinion this is just due to a lack of stable tools and a general acceptance of their value to businesses and organizations.

Software architecture still continues to be a human-generated 'black art' as programming once was. Based on the literature review at the beginning of this paper, it is a valid conclusion that while architecture may not be computer-generate any time soon, its description and depiction in a standard format and manipulation is due.

The need for Service Management in SOA architectures is becoming increasingly important. As services become ubiquitous and are not 'owned' by any specific business unit, cataloguing, maintaining and managing them is entering the commercial realm. An example are the new offerings from IBM Tivoli (IBM, 2006) (http://www-306.ibm.com/ software/ tivoli/ solutions/bsm/). As this automation increases, enterprise-level architecture patterns will increase in prominence and the need for automated testing of such architectures for compliance will not only save money but be regulated under existing and future IT governance legislation (like the Sarbannes-Oxley Act in the U.S.)

On the other hand, the movement by commercial tools vendors to close-loop the entire software development process from requirements to software deployment and maintenance will inevitably encounter hurdles at the enterprise-level architecture stage. For example, Rational tools now allow the elicitation of requirements and then tie them to UML design that then generates the software components used in the application. A big gap in between is the inter-component architecture that I have tried to address in this paper.

Last, but not the least, this paper demonstrates that *Enterprise Application Architecture Patterns* can be automatically *tested*!

CHAPTER 8 - IMPROVEMENTS AND FUTURE WORK

The future scope of this framework is vast. It includes, but is not limited to, the following:

1. XMI Bridge so that architecture documents can be imported into tools that use UML for designing the components.
2. Extending the pattern testing to xADL Architecture instances.
3. Reverse-engineering from source code annotations, specifying a pattern. A programmer can specify the pattern definition to be used and the architecture document can look up the pattern element and validate the rest of the architecture.
4. Use of abstract patterns that specify certain rules or guards while not going into further details – *but* testing would not be complete until a concrete pattern is found. This can be implemented just by extending the current framework.
5. Extending framework to security and versions (and other arbitrary run-time xADL schemas. This can be quite complex but it opens up an area of run-time pattern checking which can diagnose problems in complex systems like SOA orchestrations.
6. A separate pattern diff schema should be developed. This statement by an xADL develoeper can be used as a guide: *Perhaps the first and foremost question pertains to why we chose product line architectures as the central abstraction for organizing the activities in the software life cycle. Another viable choice, for instance,is the abstraction of features and the resulting approach of feature engineering [26]. We have no concrete evidence whether one or the other is better, but found product line architectures to be a useful abstraction since it serves as a bridge between features (identified in the requirements document) and their realization (codified in the implementation, configurations, libraries, binaries, etc.). Moreover, elements of a product line architecture can easily be represented and mapped onto different kinds of artifacts, something that is more difficult with features. Finally, we observe that features still play a role in product line architectures. Codified into attributes in guards, they are the determining factors for choosing particular system instances.* (Hoek, 2003)
7. Once ArchStudio is integrated into Eclipse, implement the TPTP interface and test. It will be executed from within Eclipse and will require all architecture locations to be Eclipse projects. Generated issues from Tron will be written to the Eclipse issues and problems logs. This automatically exposes the vast capabilities of TPTP such as monitoring, data gathering and reporting to our testing framework. Eclipse Plugin, User Interface and Reporting interfaces of Eclipse TPTP may be implemented if time permits. However, if this project generates interest in the Eclipse community, it will predictably increase in functionality as all the underlying tools of the Eclipse development platform (apart from TPTP) become available and can be implemented in support of the framework being developed in this project.
8. Pattern *diff* testing Schematron test suites once *pladiff* is available from the xADL team.

9. Junit-like regression testing scripts that can run regression tests on architecture with support for patterns during a build. This will be similar to the Java style-checking build system components like Jstyle.

10. Exercise the framework using options, guards and variants.

REFERENCES CITED

Aditya K. and Prabhakar T.V. (2003), *Implementing Architectural tactics in xADL2.0*, Project Report, Computer Science & Engineering, Indian Institute of Technology, Kanpur, India,

Caporuscio M, Inverardi P. and Pelliccione P. (2004) 'Compositional Verification of Middleware-Based Software ArchitectureDescriptions'. In:*Proceedings of the 26th International Conference on Software Engineering*, at, , pp.

CMU (), *An Overview Of Acme*. [ONLINE] . Available From: http://www.cs.cmu.edu/~acme/docs/language_overview.html (Accessed:5 August 2006).

Coelho R., Kulesza U. and Staa A. (2005) 'Improving Architecture Testability with Patterns'. In:*Conference on Object Oriented Programming Systems Languages and Applications*, atSan Diego, CA, USA, , pp.114 - 115

Dashofy E, Hoek A and Taylor R (2005a), *A Comprehensive Approach for the Development of Modular Software Architecture Description Languages* , ACM Transactions on Software Engineering and Methodology April 2005, Volume 14, Number 2, pp 199-245

Dashofy E. and Hoek A. (2001) 'Representing Product Family Architectures in anExtensible Architecture Description Language'. In:*Software Product-Family Engineering, 4th International Workshop, PFE 2001*, atBilbao, Spain, October 3-5, 2001, pp.

Dashofy, van der Hoek & Taylor (2005), *A Comprehensive Approach for the Development of Modular Software Architecture Description Languages*, University of California, Irvine,

Dimov A. and Ilieva, S. (2004) 'System level modeling of component based software systems'. In:*International Conference on Computer Systems and Technologies 2004*

Fowler, Martin (2003), *Catalog of Patterns of Enterprise Application Architecture*. [ONLINE] . Available From: http://martinfowler.com/eaaCatalog/ (Accessed:22 May 2006).

Grassi V., Mirandola R. and Sabetta A. (2005), *An XML-Based Language to Support Performance and Reliability Modeling and Analysis in Software Architectures*, Lecture Notes in Computer Science, Volume 3712, Number 2005, pp 71-87

Hoek, André van der (2003), *Design-Time Product Line Architectures for Any-Time Variability*, , Department of Informatics, School of Information and Computer Science, University of California, Irvine,

IBM (2006) 'Consultants & Systems Integrators Interchange'. In:*IBM*, atMontpellier France, Jun. 2006, pp.

IEEE (1990) *IEEE standard glossary of software engineering terminology*, IEEE, 610.12

Inverardi, Muccini & Pelliccione (2005) 'DUALLY: Putting in Synergy UML 2.0 and ADLs'. In:*5th Working IEEE/IFIP Conference on Software Architecture*, at, 2005, pp.251-252

Inverardi, Muccini and Pelliccione (2001), *Checking consistency between architectural models using SPIN*, , Dipartimento di Matematica, Universit´a dell'Aquila, Italy,

Khare et al. (2001) 'xADL: Enabling Architecture-Centric Tool Integration With XML'.

In:*System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on System Sciences 2001*, at, Jan 2001, pp.9

Medvidovic N and Taylor R (2000), *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,, Volume 26, Number 1, pp 70-93

Muccini, Dias and Richardson (2005) 'Software Architecture-basedRegression Testing'. In:*ICSE 2005 Workshop onArchitecting Dependable Systems*, at, , pp.

Muccini, H. (2002), *Software Architecture for Testing, Coordination and Views Model Checking*, PhD thesis, Universita degli Studi di Roma, 9

Naslavsky, Xu, Dias, Ziv, Richardson (2004) 'Extending xADL with Statechart Behavioral Specification'. In:*ICSE 2004 Workshop on Architecting Dependable Systems*, at, , pp.

Rational (2006), *An overview of the Systems Modeling Language for product and systems development*. [ONLINE] . Available From: http://www-128.ibm.com/developerworks/rational/library/aug06/balmelli/index.html (Accessed:26 September 2006).

Schematron (), *Schematron Overview*. [ONLINE] . Available From: http://www.schematron.com/overview.html (Accessed:5 August 2006).

Shaw M. and Garlan D. (1996) *Software Architecture: Perspectives on an Emerging Discipline*, , Prentice Hall, , pp. 67

The Open Group (2006) *Architecting the Enterprise: TOGAF 8 Certification for Practitioners*, The Open Group, 75-78

Tracz, W. (1993) 'LILEANNA: A Parameterized Programming Language'. In:*Proceedings of the Second Int'l Workshop Software Reuse*, at, March 1993, pp.66-78

UCI ISR (), *ArchStudio and Tron*. [ONLINE] . Available From: http://www.isr.uci.edu/projects/archstudio/tron.html (Accessed:2 August 2006).

UCI ISR (), *The xADL 2.0 Way*. [ONLINE] . Available From: http://www.isr.uci.edu/projects/xarchuci/way.html (Accessed:20 August 2006).

UCI-ISR (2005), *xADL 2.0*. [ONLINE] . Available From: http://www.isr.uci.edu/projects/xarchuci/ (Accessed:25 June 2006).

APPENDIX A - FULL CODE LISTINGS

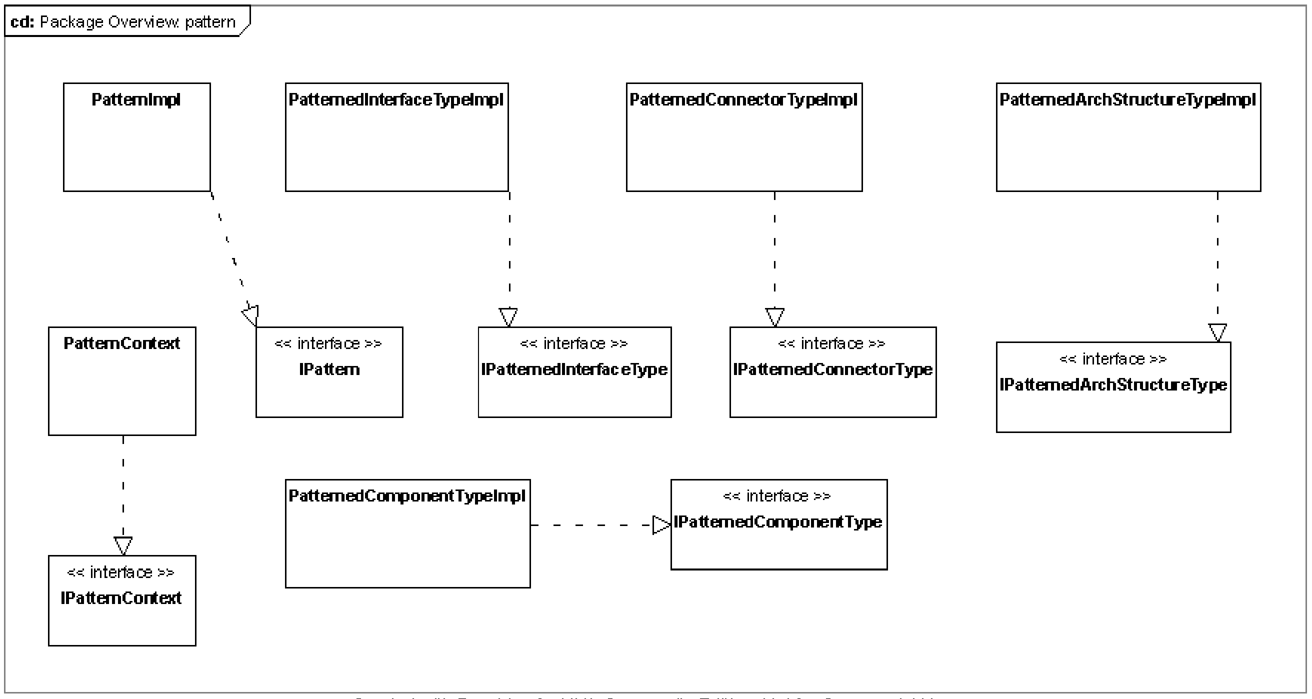
PATTERN SCHEMA – PATTERN.XSD

```
<xsd:schema xmlns="http://www.akber.com/pub/arch/xArch/pattern.xsd"
    xmlns:types="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:archinstance="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
    xmlns:impl="http://www.ics.uci.edu/pub/arch/xArch/implementation.xsd"
    targetNamespace="http://www.akber.com/pub/arch/xArch/pattern.xsd"
    elementFormDefault="qualified" attributeFormDefault="qualified">


    <!-- Import namespaces used -->
    <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
        schemaLocation="../xadl/types.xsd" />
    <xsd:import namespace="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
        schemaLocation="../xadl/instance.xsd" />
    <xsd:import
         namespace="http://www.ics.uci.edu/pub/arch/xArch/implementation.xsd"
        schemaLocation="../xadl/implementation.xsd" />


    <xsd:annotation>
        <xsd:documentation>
            xArch Pattern Schema v 1.0

            The following elements are used to define a pattern that
            an architecture or component, interface or connector follows.

            Change Log:
               2006-06-22: Akber Choudhry:
                  [akberc@gmail.com]:
                  Initial development for M.Sc. Dissertation.
                2006-07-14: Akber Choudhry:
                  [akberc@gmail.com]:
                  Changed to incorporate message and arch structure
        </xsd:documentation>
    </xsd:annotation>


<!--
    TYPE: Pattern
    This type describes a pattern.
-->
         <xsd:complexType name="Pattern">
           <xsd:choice>
             <xsd:element name="patternArch" type="types:ArchStructure"
                  minOccurs="0" maxOccurs="1"/>
             <xsd:element name="patternComp" type="types:ComponentType"
```

```
                  minOccurs="0" maxOccurs="1"/>
             <xsd:element name="patternConn" type="types:ConnectorType"
                  minOccurs="0" maxOccurs="1"/>
             <xsd:element name="patternInt" type="types:InterfaceType"
                  minOccurs="0" maxOccurs="1"/>
             <xsd:element name="patternExternal" type="archinstance:XMLLink"
                                   minOccurs="0" maxOccurs="unbounded"/>
            </xsd:choice>
         </xsd:complexType>

<!--
      TYPE: PatternedArchStructureType
      This element contains a pointer to the architecture
-->
    <xsd:complexType name="PatternedArchStructureType">
      <xsd:complexContent>
         <xsd:extension base="types:ArchStructure">
           <xsd:sequence>
             <xsd:element name="pattern" type="Pattern" minOccurs="0" />
           </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

<!--
    TYPE: PatternedComponentType

    This type is an extension to the ComponentType type
    (from the types schema).
-->
    <xsd:complexType name="PatternedComponentType">
      <xsd:complexContent>
         <xsd:extension base="impl:VariantComponentTypeImpl">
           <xsd:sequence>
             <xsd:element name="pattern" type="Pattern" minOccurs="0" />
           </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

<!--
    TYPE: PatternedConnectorType

    This type is an extension to the ConnectorType type
    (from the types schema).
-->
    <xsd:complexType name="PatternedConnectorType">
      <xsd:complexContent>
         <xsd:extension base="impl:VariantConnectorTypeImpl">
           <xsd:sequence>
             <xsd:element name="pattern" type="Pattern" minOccurs="0" />
           </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
```

```
<!--
    TYPE: PatternedInterfaceType

    This type is an extension to the InterfaceType type
    (from the types schema).
-->
    <xsd:complexType name="PatternedInterfaceType">
      <xsd:complexContent>
         <xsd:extension base="impl:InterfaceTypeImpl">
           <xsd:sequence>
             <xsd:element name="pattern" type="Pattern" minOccurs="0" />
           </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

</xsd:schema>
```

UML FOR SCHEMA BINDING CLASSES



Created with Poseidon for UML Community Edition. Not for Commercial Use.

SAMPLE XADL ARCHITECTURE DOCUMENT WITH PATTERNS

(client-server chat system architecture adapted from sample provided with xADL)

```
<?xml version="1.0" encoding="UTF-8"?>
<instance:xArch
  xmlns:implementation="http://www.ics.uci.edu/pub/arch/xArch/implementation.xsd"
  xmlns:instance="http://www.ics.uci.edu/pub/arch/xArch/instance.xsd"
  xmlns:javaimplementation="http://www.ics.uci.edu/pub/arch/xArch/javaimplementation.xsd"
  xmlns:lookupimplementation="http://www.ics.uci.edu/pub/arch/xArch/lookupimplementation.xsd"
  xmlns:pattern="http://www.akber.com/pub/arch/xArch/pattern.xsd"
  xmlns:types="http://www.ics.uci.edu/pub/arch/xArch/types.xsd"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ics.uci.edu/pub/arch/xArch/tronanalysis.xsd
            http://www.isr.uci.edu/projects/xarchuci/ext/tronanalysis.xsd
            http://www.ics.uci.edu/pub/arch/xArch/analysis.xsd
            http://www.isr.uci.edu/projects/xarchuci/ext/analysis.xsd
            http://www.ics.uci.edu/pub/arch/xArch/lookupimplementation.xsd
../xadl/lookupimplementation.xsd   http://www.ics.uci.edu/pub/arch/xArch/javaimplementation.xsd
../xadl/javaimplementation.xsd     http://www.ics.uci.edu/pub/arch/xArch/variants.xsd
../xadl/variants.xsd    http://www.ics.uci.edu/pub/arch/xArch/implementation.xsd
../xadl/implementation.xsd         http://www.ics.uci.edu/pub/arch/xArch/types.xsd
            http://www.isr.uci.edu/projects/xarchuci/ext/types.xsd
            http://www.ics.uci.edu/pub/arch/xArch/instance.xsd ../xadl/instance.xsd
            http://www.akber.com/pub/arch/xArch/pattern.xsd pattern.xsd">


  <types:archStructure types:id="ChatSystem"
    xsi:type="pattern:PatternedArchStructureType">
    <types:description xsi:type="instance:Description">
      Chat System Demo Architecture
    </types:description>
    <types:component types:id="Server" xsi:type="types:Component">
      <types:description xsi:type="instance:Description">
        Server Component
      </types:description>
      <types:interface types:id="Server.IFACE_TOP"
        xsi:type="types:Interface">
        <types:description xsi:type="instance:Description">
          Server Component Top Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2TopType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
        <types:signature xlink:href="#Server_type_topSig"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:interface>
      <types:interface types:id="Server.IFACE_BOTTOM"
        xsi:type="types:Interface">
        <types:description xsi:type="instance:Description">
          Server Component Bottom Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2BottomType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
        <types:signature xlink:href="#Server_type_bottomSig"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:interface>
      <types:type xlink:href="#Server_type" xlink:type="simple"
        xsi:type="instance:XMLLink" />
    </types:component>
    <types:component types:id="ChatClient1"
      xsi:type="types:Component">
      <types:description xsi:type="instance:Description">
        Chat Client 1 Component
      </types:description>
      <types:interface types:id="ChatClient1.IFACE_TOP"
        xsi:type="types:Interface">
        <types:description xsi:type="instance:Description">
          Chat Client 1 Component Top Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2TopType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
        <types:signature xlink:href="#Client_type_topSig"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:interface>
      <types:interface types:id="ChatClient1.IFACE_BOTTOM"
        xsi:type="types:Interface">
        <types:description xsi:type="instance:Description">
          Chat Client 1 Component Bottom Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2BottomType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
        <types:signature xlink:href="#Client_type_bottomSig"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:interface>
      <types:type xlink:href="#Client_type" xlink:type="simple"
        xsi:type="instance:XMLLink" />
    </types:component>
    <types:component types:id="ChatClient2"
      xsi:type="types:Component">
      <types:description xsi:type="instance:Description">
        Chat Client 2 Component
      </types:description>
      <types:interface types:id="ChatClient2.IFACE_TOP"
        xsi:type="types:Interface">
        <types:description xsi:type="instance:Description">
          Chat Client 2 Component Top Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2TopType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
        <types:signature xlink:href="#Client_type_topSig"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:interface>
      <types:interface types:id="ChatClient2.IFACE_BOTTOM"
        xsi:type="types:Interface">
        <types:description xsi:type="instance:Description">
          Chat Client 2 Component Bottom Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2BottomType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
        <types:signature xlink:href="#Client_type_bottomSig"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:interface>
      <types:type xlink:href="#Client_type" xlink:type="simple"
        xsi:type="instance:XMLLink" />
    </types:component>
    <types:connector types:id="Bus" xsi:type="types:Connector">
      <types:description xsi:type="instance:Description">
        The Bus
      </types:description>
      <types:interface types:id="Bus.IFACE_TOP"
        xsi:type="types:Interface">
        <types:description xsi:type="instance:Description">
          The Bus Top Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2TopType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
        <types:signature xlink:href="#Bus_type_topSig"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:interface>
      <types:interface types:id="Bus.IFACE_BOTTOM"
        xsi:type="types:Interface">
        <types:description xsi:type="instance:Description">
          The Bus Bottom Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2BottomType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
        <types:signature xlink:href="#Bus_type_bottomSig"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:interface>
      <types:type xlink:href="#Bus_type" xlink:type="simple"
        xsi:type="instance:XMLLink" />
    </types:connector>
    <types:link types:id="Server_to_Bus" xsi:type="types:Link">
      <types:description xsi:type="instance:Description">
        Server to Bus Link
      </types:description>
      <types:point xsi:type="instance:Point">
        <instance:anchorOnInterface xlink:href="#Server.IFACE_BOTTOM"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:point>
      <types:point xsi:type="instance:Point">
        <instance:anchorOnInterface xlink:href="#Bus.IFACE_TOP"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:point>
    </types:link>
    <types:link types:id="Bus_to_ChatClient1" xsi:type="types:Link">
      <types:description xsi:type="instance:Description">
        Bus to ChatClient1 Link
      </types:description>
      <types:point xsi:type="instance:Point">
        <instance:anchorOnInterface xlink:href="#Bus.IFACE_BOTTOM"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:point>
      <types:point xsi:type="instance:Point">
        <instance:anchorOnInterface xlink:href="#ChatClient1.IFACE_TOP"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:point>
    </types:link>
    <types:link types:id="Bus_to_ChatClient2" xsi:type="types:Link">
      <types:description xsi:type="instance:Description">
        Bus to ChatClient2 Link
      </types:description>
      <types:point xsi:type="instance:Point">
        <instance:anchorOnInterface xlink:href="#Bus.IFACE_BOTTOM"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:point>
      <types:point xsi:type="instance:Point">
        <instance:anchorOnInterface xlink:href="#ChatClient2.IFACE_TOP"
          xlink:type="simple" xsi:type="instance:XMLLink" />
      </types:point>
    </types:link>
</types:archStructure>
<types:archTypes xsi:type="types:ArchTypes">
  <types:componentType types:id="Server_type"
    xsi:type="pattern:PatternedComponentType">
    <types:description xsi:type="instance:Description">
      Server Component Type
    </types:description>
      <types:signature types:id="Server_type_topSig"
        xsi:type="types:Signature">
        <types:description xsi:type="instance:Description">
          Server_type_topSig
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2TopType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
      </types:signature>
      <types:signature types:id="Server_type_bottomSig"
        xsi:type="types:Signature">
        <types:description xsi:type="instance:Description">
          Server_type_bottomSig
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2BottomType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
      </types:signature>
      <implementation:implementation
        xsi:type="javaimplementation:JavaImplementation">
        <javaimplementation:mainClass
          xsi:type="javaimplementation:JavaClassFile">
          <javaimplementation:javaClassName
            xsi:type="javaimplementation:JavaClassName">
            c2demo.chatsys.ServerC2Component
          </javaimplementation:javaClassName>
        </javaimplementation:mainClass>
      </implementation:implementation>
      <pattern:pattern>
        <pattern:patternExternal xlink:href="pattern1.xml"
          xlink:type="file" />
      </pattern:pattern>
    </types:componentType>
    <types:componentType types:id="Client_type"
      xsi:type="implementation:VariantComponentTypeImpl">
      <types:description xsi:type="instance:Description">
        Client Component Type
      </types:description>
      <types:signature types:id="Client_type_topSig"
        xsi:type="types:Signature">
        <types:description xsi:type="instance:Description">
          Client_type_topSig
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2TopType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
      </types:signature>
      <types:signature types:id="Client_type_bottomSig"
        xsi:type="types:Signature">
        <types:description xsi:type="instance:Description">
          Client_type_bottomSig
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
        <types:type xlink:href="#C2BottomType" xlink:type="simple"
          xsi:type="instance:XMLLink" />
      </types:signature>
      <implementation:implementation
        xsi:type="javaimplementation:JavaImplementation">
        <javaimplementation:mainClass
          xsi:type="javaimplementation:JavaClassFile">
          <javaimplementation:javaClassName
            xsi:type="javaimplementation:JavaClassName">
            c2demo.chatsys.ClientC2Component
        </javaimplementation:javaClassName>
      </javaimplementation:mainClass>
    </implementation:implementation>
</types:componentType>
<types:connectorType types:id="Bus_type"
  xsi:type="implementation:VariantConnectorTypeImpl">
  <types:description xsi:type="instance:Description">
    Bus Connector Type
  </types:description>
  <types:signature types:id="Bus_type_topSig"
    xsi:type="types:Signature">
    <types:description xsi:type="instance:Description">
      Bus_type_topSig
    </types:description>
    <types:direction xsi:type="instance:Direction">
      inout
    </types:direction>
    <types:type xlink:href="#C2TopType" xlink:type="simple"
      xsi:type="instance:XMLLink" />
  </types:signature>
  <types:signature types:id="Bus_type_bottomSig"
    xsi:type="types:Signature">
    <types:description xsi:type="instance:Description">
      Bus_type_bottomSig
    </types:description>
    <types:direction xsi:type="instance:Direction">
      inout
    </types:direction>
    <types:type xlink:href="#C2BottomType" xlink:type="simple"
      xsi:type="instance:XMLLink" />
  </types:signature>
  <implementation:implementation
    xsi:type="javaimplementation:JavaImplementation">
    <javaimplementation:mainClass
      xsi:type="javaimplementation:JavaClassFile">
      <javaimplementation:javaClassName
        xsi:type="javaimplementation:JavaClassName">
        c2.legacy.conn.BusConnector
      </javaimplementation:javaClassName>
    </javaimplementation:mainClass>
  </implementation:implementation>
</types:connectorType>
<types:interfaceType types:id="C2TopType"
  xsi:type="implementation:InterfaceTypeImpl">
  <types:description xsi:type="instance:Description">
    C2 Top Interface
  </types:description>
  <implementation:implementation
    xsi:type="javaimplementation:JavaImplementation">
    <javaimplementation:mainClass
      xsi:type="javaimplementation:JavaClassFile">
      <javaimplementation:javaClassName
        xsi:type="javaimplementation:JavaClassName">
        c2.fw.SimpleInterface
      </javaimplementation:javaClassName>
    </javaimplementation:mainClass>
  </implementation:implementation>
  <implementation:implementation
    xsi:type="lookupimplementation:LookupImplementation">
    <lookupimplementation:name
      xsi:type="lookupimplementation:LookupName">
      IFACE_TOP
    </lookupimplementation:name>
  </implementation:implementation>
</types:interfaceType>
<types:interfaceType types:id="C2BottomType"
  xsi:type="implementation:InterfaceTypeImpl">
  <types:description xsi:type="instance:Description">
    C2 Bottom Interface
  </types:description>
  <implementation:implementation
```

```
        xsi:type="javaimplementation:JavaImplementation">
        <javaimplementation:mainClass
          xsi:type="javaimplementation:JavaClassFile">
          <javaimplementation:javaClassName
            xsi:type="javaimplementation:JavaClassName">
            c2.fw.SimpleInterface
          </javaimplementation:javaClassName>
        </javaimplementation:mainClass>
      </implementation:implementation>
      <implementation:implementation
        xsi:type="lookupimplementation:LookupImplementation">
        <lookupimplementation:name
          xsi:type="lookupimplementation:LookupName">
          IFACE_BOTTOM
        </lookupimplementation:name>
      </implementation:implementation>
    </types:interfaceType>
  </types:archTypes>
</instance:xArch>
```