An Infrastructure for the Rapid Development of XML-based Architecture Description Languages

Eric M. Dashofy André van der Hoek Richard N. Taylor

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
+1 949 824 2260
{edashofy,andre,taylor}@ics.uci.edu

ABSTRACT

Research and experimentation in software architectures over the past decade have yielded a plethora of software architecture description languages (ADLs). Continuing innovation indicates that it is reasonable to expect more new ADLs, or at least ADL features. This research process is impeded by the difficulty and cost associated with developing new notations. An architect in need of a unique set of modeling features must either develop a new architecture description language from scratch or undertake the daunting task of modifying an existing language. In either case, it is unavoidable that a significant effort will be expended in building or adapting tools to support the language. To remedy this situation, we have developed an infrastructure for the rapid development of new architecture description languages. Key aspects of the infrastructure are its XML-based modular extension mechanism, its base set of reusable and customizable architectural modeling constructs, and its equally important set of flexible support tools. This paper introduces the infrastructure and demonstrates its value in the context of several real-world applications.

Keywords

Architecture description languages, XML, extensibility.

1. INTRODUCTION

Software architectures [32] provide a way to reason about software systems at a level of abstraction above that of simple modules, objects or lines-of-code. To model systems at this level, architects must have expressive modeling languages and tools to manipulate models expressed in those languages.

The traditional way to represent a software architecture is to model it in an architecture description language (ADL). Many ADLs have been developed by both the research- and practice-oriented communities. A comprehensive survey of architecture description languages [27] reveals that most ADLs share a set of fundamental modeling constructs and concepts, including components, connectors, interfaces, and architectural configurations. ADLs are distinguished from one another by a small number of features, supported by tools, that have evolved from various areas of interest or need.

ADL research is still an active area, making it unlikely that a single, unified ADL will soon emerge. We can expect a continuing proliferation of new ADLs and ADL features for several reasons. First, the software architecture community continues to identify and experiment with new ADL features and combinations thereof. This is natural, given that different domains have widely different areas of concern, and depending on the purpose of the architectural model, certain constructs may or may not be appropriate or useful. Next, the software architecture community does not agree on what features should be present in an ADL, or precisely what these features should model [27].

Given the continuing succession of innovations from the research community, it would be useful to have an infrastructure with which to quickly construct new ADLs. Furthermore, it should be possible to do so efficiently by combining compatible features together and extending/modifying existing features. Unfortunately, no such infrastructure has been developed yet. Architects thus develop new ADLs by writing their own grammars in meta-languages of their choosing and building their own parsers, syntax checkers, and support tools.

In an effort to address this situation, we have developed an infrastructure for the rapid development of XML-based [7] ADLs. Our infrastructure provides:

1. an XML-based modular extension mechanism for rapidly developing new ADLs;
2. a base set of features that can be reused in ADL development; and
3. a flexible set of tools to support ADL development and use.

Using this infrastructure, architects can efficiently and effectively create new ADLs and modify or extend ADLs created in the infrastructure. This results in a significant overall reduction in effort.

Tool support is especially vital for the successful use of any ADL because architecture descriptions persist throughout the product development lifecycle and evolve along with the described software system. Tools are required to create, manipulate, and maintain these documents over time. Our infrastructure provides generic tools valuable to ADL developers like parsers, syntax checkers, and syntax directed editors for architecture descriptions. These tools can serve as the basis for more advanced tools that exploit new ADL features.

It is important to note that neither XML nor our infrastructure attempts to enforce semantic consistency within an ADL. The infrastructure is merely a way to define and manipulate represen-

tations of software architectures; interpreting the representations, or constraining *how* they may be manipulated, is the job of external tools. As such, issues such as resolving feature interaction problems and enforcing internal consistency of the model are not addressed. For instance, developers experimenting with modeling new aspects of software systems may use our infrastructure to temporarily specify redundant or conflicting features in their ADLs if they so desire. Investigating the semantic relationships between ADL features is an open research area, and our infrastructure can serve as the basis for tools and languages that support this research.

Our infrastructure has been used in several projects, within our research group, by industrial practitioners, and by other researchers. These experiences confirm the effectiveness of different aspects of our infrastructure. We demonstrate that our infrastructure is scalable by modeling and simulating a large military system. We show the value of our infrastructure's adaptable and extensible aspects and tool support in experiments modeling spacecraft software architectures. Finally, we demonstrate the infrastructure's ability to capture concepts from different representations of an emerging domain (product line architectures). This experience also shows our infrastructure's ability to add tool support for new constructs quickly.

The rest of this paper is organized as follows: Section 2 provides a background on ADL research to date, Section 3 describes our approach and infrastructure in detail, Section 4 shows the benefits of our infrastructure in the context of experiences in several domains, and Section 5 describes related work and how our approach compares to that work. The final section summarizes our conclusions and describes our future work.

2. BACKGROUND

Architecture description languages are formal notations "providing features for modeling a software system's conceptual architecture, distinguished from the system's implementation. ADLs provide both a concrete syntax and a conceptual framework for characterizing architectures" [27]. These notations are typically supported by tools that facilitate understanding, visualizing, analyzing, instantiating, and simulating architecture descriptions. Representative ADLs include Darwin [24], Wright [1], Rapide [23], MetaH [5], C2SADEL [26], and Koala [29]. As Medvidovic and Taylor [27] point out, the minimum requirement for a language to be an ADL is the ability to represent components, connectors, architectural configurations, and interfaces. Each language listed here has this ability. Additionally, each language listed here contains distinguishing features that model aspects of software systems in new or unique ways (see Table 1).

Table 1. Representative ADLs and distinguishing features.

ADL	Distinguishing Features
Darwin	Ability to model distributed, dynamic systems; operation model described in pi-calculus.
Wright	Explicit connectors with checkable formal semantics.
Rapide	Event-based architectures specified using partially-ordered sets of events (POSETS); simulation tools to check interactions of event-based component behaviors.
MetaH	Specification of how software modules interact with hardware, real-time and concurrent state machine aspects.
C2SADL	Multiple, heterogeneous, subtyping mechanisms; varying levels of type conformance. Ability to model architecture changes imperatively.
Koala	Ability to model product line architectures with optional and variant elements.

The wide variety of ADL features corresponds to the equally wide variety of capabilities desired by architects. One way to differentiate ADLs is by what kinds of architectures they can model. Some ADLs are very generic, like Darwin and Wright, while others are better suited to specific domains or architectural styles: Rapide specifically supports event-based architectures; C2SADEL was built to model architectures in the multi-level notify/request C2 style [38]; MetaH was built to model embedded systems, including both hardware and software components. Another way to differentiate ADLs is by what features of software systems they model. Koala, for instance, models product lines by modeling variation points in an architecture. C2SADEL and Darwin are both capable of modeling properties of dynamic architectures—those that change at run-time—although they do so in different ways.

Some ADLs are oriented towards formal analysis—the language and support tools help architects verify certain properties of the software system. For instance, Wright's toolset convert specifications into CSP [13] and check the CSP model for semantic consistency. MetaH's associated tools perform real-time scheduling analysis on systems specified in the MetaH language. Other ADLs are more oriented toward simulation—the ability to understand how architectural elements interact over time. Rapide's tools simulate a specification and provide event traces. Recently, the creators of Darwin have been experimenting with using labeled transition systems (LTS) [25] to simulate the behavior of architectures specified in Darwin.

New research issues continue to emerge from the architecture community. Areas of interest include distributed architectures [24], dynamic architectures. [31], integrating architectures with software deployment and maintenance [14], and product line architectures [9]. Research in areas such as these indicates that it is reasonable to expect new ADLs, or at least ADL features, from the software architecture community. This exemplifies the need for our infrastructure, which reduces the cost of creating and modifying ADLs, experimenting with new features, and building associated tools.

3. APPROACH

To date, many ADLs have been monolithic. Their feature sets and grammar are fixed, and adding new constructs to a monolithic ADL is not possible without modifications to the tool set supporting that ADL. The cost of extending and adapting such an ADL and its tools is significant, and may be comparable to developing a new ADL from scratch.

We approach this problem by developing a modularly extensible architecture description language. In this approach, sets of related

features are defined in individual modules. These modules can define new modeling constructs and extend constructs in other modules. New ADLs are formed by composing modules together.

The idea of an extensible language is not new. Extensible programming languages have been investigated for several decades [8][37]. The results of this research have been adapted to support extensible languages for representing data as well. Examples of such languages are SGML [18], RDF [21], and XML [7]. Among these, XML has several extrinsic benefits not shared by the others: XML has broad support from standards bodies (most notably the W3C), researchers, and practitioners, along with a large and growing set of support tools. This gives any XML-based language a head-start in terms of compatibility and tool support. Additionally, the new XML Schema standard [39], discussed below, provides a meta-language suitable for developing modular and extensible notations. Recognizing these factors, and based on our previous successes building ADLs using XML [19], we have built our infrastructure to support modularly extensible ADLs that are defined in XML schemas and manipulated using tools.

Using XML's ability to create modularly extensible ADLs is only the first part of our approach. A base set of generic schemas and flexible tool support are equally important parts of our infrastructure. It is our strong belief that all three are needed for this infrastructure to be effective. We discuss details of each part of our infrastructure below.

3.1 XML Schemas and Extensiblity

The first part of our infrastructure is its adoption of XML schemas and their extensibility mechanisms to define modularly extensible ADLs. Since its inception, XML has provided capabilities for defining modular, extensible languages. XML's original meta-language, the document type definition (DTD) [7], can be used for this purpose, as is evidenced by the Modularization of XHTML W3C effort [2]. However, using DTDs to create modular languages introduces a specific problem: each combination of modules requires a new "hybrid DTD" to describe the resulting language. A hybrid DTD is a separate document that describes how the modules are connected to create a new language; creating and maintaining this document introduces additional overhead into the process of creating a modular language.

The XML Schema standard, recently ratified as a W3C recommendation, describes a more expressive meta-language for XML that is superior to DTDs in a number of ways. First, whereas DTDs use a style of tags and declarations separate from XML documents, XML schemas resemble XML documents. More importantly, XML schemas add a type system to XML. This includes basic types like integers and strings that define the contents of atomic elements and attributes. Complex types are also allowed, in which elements contain other elements and attributes. In this type system, types can be extended in a manner similar to object-oriented subtyping: types can be extended to add new information, in the form of elements or attributes, without modifying the base type's definition. An additional advantage of XML subtyping over object-oriented subtyping lies in type restrictions: elements and attributes can be removed or restricted in an extended type.

XML schemas provide the meta-language with which modularly extensible architecture description languages are created in our infrastructure. Each schema is one module. Types defined in a schema may describe new first-class constructs, or add/remove elements from a base type defined in another schema. Architects can create a new ADL, then, by choosing an appropriate set of schemas. They may create these schemas on their own, or they may reuse and adapt schemas written by other architects.

XML does not solve all the problems of creating a modularly extensible language. First of all, it does not guarantee syntactic compatibility among modules. This is mostly due to the fact that the current version of the XML schema standard does not support multiple type inheritance. As such, two subtypes that extend the same base type cannot be combined in a single element. This can be resolved by introducing artificial dependencies—making one subtype an extension of the other, even though they may be conceptually orthogonal. We have done this successfully with our own schemas, as described in the next section. The W3C is considering multiple inheritance for inclusion in a future version of XML schemas, which would alleviate this problem.

Second, XML is just syntax. It cannot describe the semantics of individual elements or relationships among elements. Therefore, semantics must be checked and enforced with external semantically-aware tools. This is not different from traditional language development.

3.2 xADL 2.0

The second part of our infrastructure is a set of reusable schemas that can be used as the basis for developing new ADLs, collectively known as xADL 2.0 [10]. To maximize the reusability and applicability of these schemas, we have endeavored to make them as generic as possible. For instance, our schemas define components and connectors, but not their behaviors or how they can be linked together. These aspects are an important part of many ADLs, and we have designed the xADL 2.0 schemas so such aspects can be specified in extension schemas.

xADL 2.0 provides constructs that are useful for describing software architectures in general, as well as three important features that can be used in an ADL derived from xADL 2.0. These are:

1. separation of run-time and design-time models of a software system;
2. implementation mappings that map the ADL specification of an architecture onto executable code; and
3. the ability to model aspects of architectural evolution and product line architectures.

The breakdown of these high-level features into individual schemas is shown in Table 2. We discuss each schema in detail below.

3.2.1 Separation of Run-Time and Design-Time Models

Traditionally, ADLs have focused on design-time aspects of a software system or have combined run-time and design time aspects in a single model. However, research on dynamic software architectures [20][31] has revealed that it is useful to provide a separate architectural model of a software system at run-time. Run-time models capture aspects of a running software system that are different from aspects captured at design-time. For instance, a design-time model of a system might contain information such as: basic metadata about elements (e.g., authors, sizes, textual descriptions), expected behavior of components and connectors, and constraints on the arrangements of components and con-

nectors. In contrast, a run-time model of the same system might contain information such as the current state of a component or connector (e.g., 'not started,' 'running,' 'suspended,' 'blocked,' 'error state'), where a component is running in a distributed system (e.g., which machine, what processor, its process id), and its communication status (e.g., events or calls waiting to be processed, a history of recently processed requests).

In xADL 2.0, two schemas accomplish the separation of run-time and design-time models. Constructs modeling run-time aspects of a system are defined in the INSTANCES schema, also known as "xArch," which we defined in collaboration with researchers at Carnegie Mellon University. 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;
- subarchitectures (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;
- subarchitectures (composite components and connectors with internal architectures);
- general groups;
- component types;
- connector types; and
- interface types.

Table 2. xADL 2.0 schemas and features.

Purpose	Schema	Features
Design-time and Run-time Models	**Instances**	Run-time component, connector, interface, and link instances; subarchitectures; general groups.
	Structure & Types	Design-time components, connectors, interfaces, and links; subarchitectures; general groups; component, connector, and interface types.
Implementation Mappings	**Abstract Implementation**	Placeholder for implementation data for components, connectors, and interfaces.
	Java Implementation	Concrete implementation data for Java-language components, connectors, and interfaces
Architectural 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.

In keeping with the nature of xADL 2.0, the constructs defined in the INSTANCES and STRUCTURE & TYPES schemas are highly generic. 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.

3.2.2 Implementation Mappings

A second important feature of xADL 2.0 is its support for mapping an architecture design onto executable code. Several ADLs such as MetaH support or require a mapping between an architecture specification and its implementation. This is essential if a software system is to be automatically instantiated from its architecture description.

Since xADL 2.0 is not bound to a particular implementation platform or language, it is impossible to know, *a priori*, exactly what kinds of implementations will be mapped to architecture descriptions. Obvious possibilities include Java classes and archives, Windows DLLs, UNIX shared libraries, and CORBA components, but making a comprehensive list is infeasible.

To address this, xADL 2.0 adopts a two-level approach. 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.

3.2.3 Modeling Architectural Evolution and Product Line Architectures

Modeling architectural evolution and product lines are emerging, but important research areas. Initial work in both these areas has focused on applying configuration management concepts to architectures [15][16]. Thus, from a modeling perspective, both areas

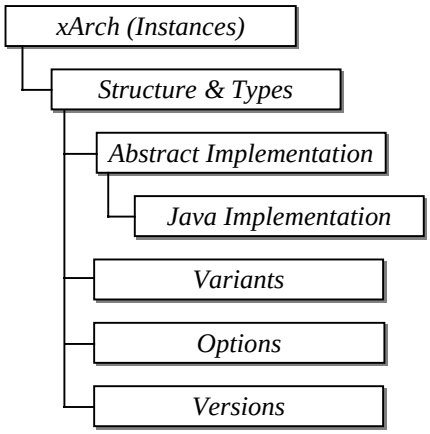

Figure 1. Conceptual dependencies of xADL 2.0 schemas. Child nodes are dependent on their parents.

can be addressed by adding configuration management concepts to an ADL.

The three most important aspects of modeling the evolution of architectures and product lines are *versions*, *options*, and *variants*. 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.

Versions

The VERSIONS schema adds versioning constructs to xADL 2.0. In xADL 2.0, architecture element *types* are the versioned entities [10]. The VERSIONS schema defines:

- version graphs for component types;
- version graphs for connector types; and
- 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.

Options

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

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

Optional elements are accompanied by a "guard condition." This condition, whose format can be specified in an extension, is evaluated when the architecture is instantiated. If the condition is met, then the optional element is included in the architecture; otherwise it is excluded.

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.

3.2.4 Modularity and Incrementality of the xADL 2.0 Schemas

Some of the xADL 2.0 schemas extend constructs defined in other schemas. This introduces dependencies between them. The conceptual dependencies of the various xADL 2.0 schemas are shown in Figure 1. We have attempted to minimize these dependencies when possible. For instance, while the VERSIONS, OPTIONS, and VARIANTS schemas are all dependent on the STRUCTURE & TYPES schema, they are not dependent on one another. Thus, it is possible to have an architecture description that has options and variants, but not versions.

As mentioned earlier, one drawback of using XML schemas is that additional artificial dependencies need to be introduced to resolve conflicts when two extensions are applied to the same base type. We minimized the effect of this in the xADL 2.0 schemas [10].

We have also made the schemas modular and incremental at the level of individual constructs within the schemas. Where possible, we have made individual elements optional. For instance, some ADLs [24][29] do not use explicit connectors. While xADL 2.0 supports them, they are not required in a xADL 2.0-based architecture specification. The xADL 2.0 schemas do not constrain what kinds of elements may be connected, so links can connect components directly if needed. This finer-grained modularity makes the xADL 2.0 schemas even more generic and useful as the basis for a wide range of ADLs.

3.3 Tool Support

The third part of our infrastructure is its extensive set of flexible tools. Tools in our infrastructure provide parsing, syntax checking, and syntax-directed editing based on ADL schemas. These tools are syntax-driven, and provide the basis for tools that perform semantic functions—analysis, simulation, instantiation, testing, etc. Interpreting the elements defined by the XML schemas and enforcing semantic constraints or relationships between those elements is out of the scope of our tools.

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.

Each of the tools in our infrastructure is described in detail here. A diagram showing the various tools and the relationships among them is shown in Figure 2.

3.3.1 COTS and Open-Source XML Tools

One of the key benefits of using XML as the basis for new ADLs is the abundance of commercial-off-the-shelf (COTS) and open-source tools available for manipulating XML documents. These XML tools provide the most basic level of support for editing architecture descriptions. Two such tools that play an important part in our framework are XML Spy [3], and Apache Xerces [4].

XML Spy

XML Spy is an integrated development environment for XML. It has extensive support for both DTDs and XML schemas, and provides an editor and validator for XML documents and schemas. As an XML tool, it exposes many details about XML to its users, limiting its effectiveness as an editor for architecture descriptions. However, it has proven useful as an XML schema editor and validator. We used XML Spy extensively in our development of the xADL 2.0 schemas. We expect that other users of our infrastructure will find this tool, and other XML editors like it, useful in developing schemas as well.

Apache Xerces

Apache Xerces is an open-source programmatic library for parsing and manipulating XML documents. It implements an XML parser and validator, as well as the W3C-defined APIs SAX [28] and DOM [22]. While we do not expect users of our infrastructure to interact with Xerces directly, it is an important part of many of the tools in our infrastructure because it allows our tools to interact with XML documents programmatically. Without it, we would have had to create our own XML parser and APIs, increasing our effort substantially.

3.3.2 Data Binding Library

Our infrastructure includes a library of Java-to-XML data bindings [6] 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. Manipulating the objects causes corresponding changes in the underlying XML document. Whereas a generic XML API like DOM exposes functions like `addElement(...)` and `getChildElements(...)`, our data binding library exposes

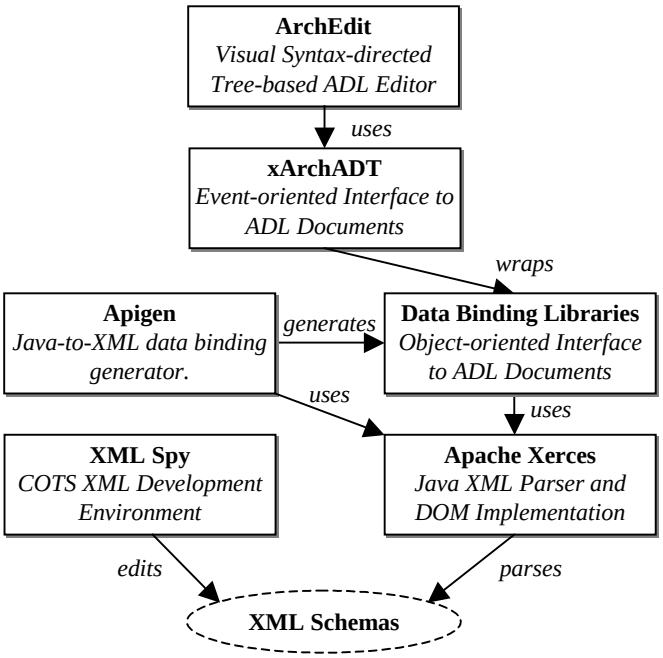

Figure 2. Infrastructure tools and their relationships.

functions like `addComponentInstance(...)` and `getAllGroups(...)`. Internally, the library uses the DOM implementation provided with Apache Xerces to manipulate the underlying XML document.

Consider the following XML definition of a component, from the xADL 2.0 STRUCTURE & TYPES Schema (namespace information omitted for clarity):

```
<complexType name="Component">
  <sequence>
    <element name="description"
      type="Description"/>
    <element name="interface"
      type="Interface"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="type"
       type="XMLLink"
       minOccurs="0" maxOccurs="1"/>
  </sequence>
  <attribute name="id" type="Identifier"/>
<complexType>
```

For this type, the data binding library includes a Java class that exposes the following interface:

```
void setDescription(IDescription value);
void clearDescription();
IDescription getDescription();
void addInterface(IInterface newInterface);
void addInterfaces(Collection interfaces);
void clearInterfaces();
IInterface getInterface(String id);
Collection getInterfaces(Collection ids);
Collection getAllInterfaces();
void removeInterface(IInterface interface);
void setType(IXMLLink link);
void clearType();
IXMLLink getType();
void setId(String id);
String getId();
void clearId();
```

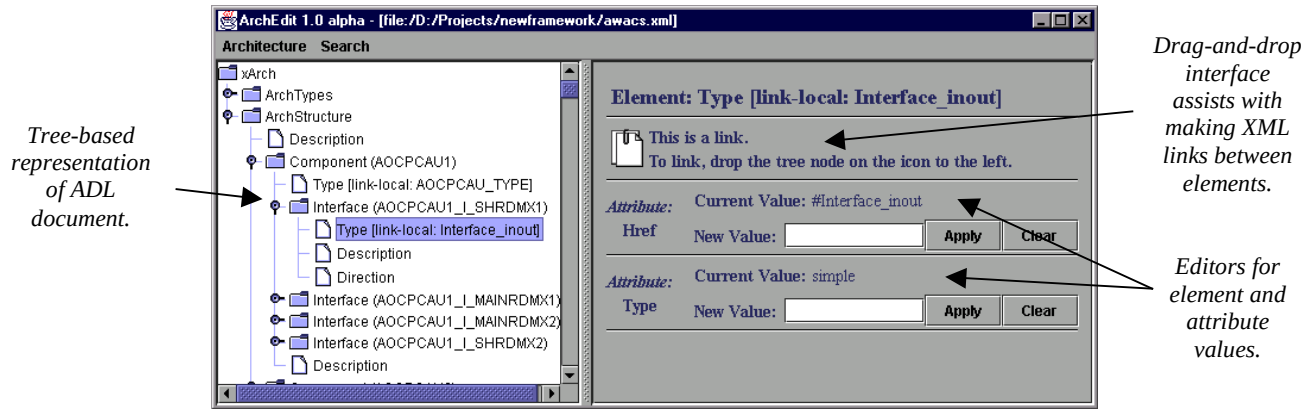

Figure 3. ArchEdit Screenshot.

This demonstrates that, despite having no knowledge of the semantics of the ADL, the data binding library exposes functions that are closer (in terms of their level of abstraction) to the concepts relevant to a software architect. This makes building architecture tools with the data binding library more intuitive than building them with an XML tool like Xerces.

3.3.3 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.

Apigen is not a generic data-binding generator; it does not support the full XML schema language. However, it supports a large set of schema features, and so far has been sufficient to generate data bindings for all the xADL 2.0 schemas as well as schemas written by third parties.

3.3.4 xArchADT

The data binding library provides a traditional object-oriented interface to edit architecture descriptions. This requires the library's callers to maintain many direct object references. In general, distributed and event-based systems assume that components do not share an address space, and therefore cannot contain object references across components. Because of this, using such a library as an independent component in a distributed or event-based system is difficult.

To address this, we have built a wrapper, called 'xArchADT,' for the data-binding library that provides an event-based interface instead of an object-oriented one. Instead of procedure calls, xArchADT is accessed via asynchronous events. It uses reified first-class object references, rather than direct pointers, to refer to elements in xADL 2.0 documents. When the underlying architecture description is modified by one tool, xArchADT emits an event informing all listening tools of the change. This gives the data binding library the added property of loose coupling.

Like the data binding library itself, this component is highly flexible. xArchADT uses Java's reflection capabilities to adapt to changes in the data binding library automatically. That is, if the library is regenerated by apigen, xArchADT will work without modification.

3.3.5 ArchEdit

The data binding library and xArchADT expose different programmatic interfaces for manipulating architecture descriptions. Our infrastructure also includes a user interface-based tool called ArchEdit. A screenshot of this tool is shown in Figure 3. ArchEdit depicts an architecture description graphically in a tree format, where each node can be expanded, collapsed, or edited. This is similar to many visual XML editors, except ArchEdit hides the XML details of the document from the user. The ADL's XML schemas direct the structure of the displayed tree view, making the structure of the XML document and the structure of the displayed tree are identical. This gives architects direct access to architecture descriptions without abstracting away details of the architecture.

ArchEdit is syntax-driven—it does not understand the semantics of the displayed elements. It does not enforce stylistic constraints or other rules on the architecture description that cannot be specified in XML. The advantage of having such a tool is that it builds its view and interface dynamically from the XML schemas used to define the ADL. Therefore, it does not need to be modified when schemas are added, modified, or removed. This flexibility is valuable because it gives architects a simple graphical editor for ADL documents automatically, even if the new ADL features have recently been added.

ArchEdit is an event-based software component and accesses architecture descriptions through xArchADT. Changes to the architecture description made via xArchADT by ArchEdit or other tools are immediately reflected in the ArchEdit user interface.

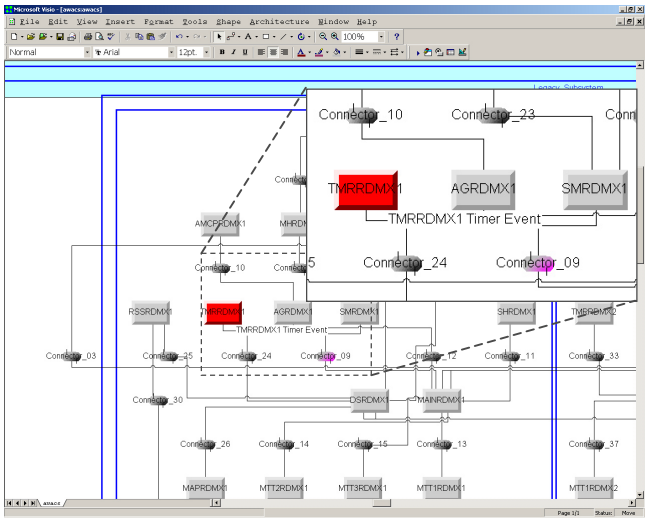

Figure 4. AWACS simulator screenshot with detail callout.

4. EXPERIENCES

Our infrastructure has demonstrated its effectiveness in a number of problem domains. In this section, we highlight three experiences that each demonstrate a different strength of the infrastructure. First, we show how our infrastructure supported the modeling and simulation of the architecture of a large military system—demonstrating the scalability of the infrastructure. Second, we show how our infrastructure supported the development of an ADL now used in architectural modeling experiments for spacecraft systems—demonstrating the adaptability of the infrastructure to new architectural domains with unique modeling requirements. Third, we show how our infrastructure supported the development of ADLs for Koala [29] and Mae [16], two representations used to model product line architectures—demonstrating the extensibility of the XML schemas and tools in our infrastructure. These three experiences demonstrate how the different aspects of our infrastructure (XML-based extensibility, a base set of schemas, and flexible tool support) contribute to its effective use.

4.1 AWACS

To demonstrate the scalability of our infrastructure, we used it to model the AWACS aircraft's [41] 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. We then used this model as the basis for an architecture-driven simulation of the AWACS software system.

Because of the large size of the architecture, and its high internal regularity (similar elements repeated over and over), we built the initial AWACS description programmatically with a small (1000-line) program that calls our infrastructure's data binding library. The result was an architecture document consisting of approximately 10,000 lines of XML. Obviously, creating such a large description by hand or in a GUI-based editor would have been infeasible, further demonstrating the value of the data binding library. The AWACS description describes the components, connectors, interfaces, and links in the architecture, along with component, connector, and interface types. We validated this description against the xADL 2.0 schemas using XML Spy and visualized it with ArchEdit. We used ArchEdit to inspect the architecture and make further improvements until the model was accurate.

We also built an AWACS simulator with our infrastructure to visualize the interactions among the components. We created implementations for each component type and connector type in Java. Using xArchADT to read the architecture description, a short bootstrap program instantiates and connects the elements. To visualize the simulation, we added an extension to Microsoft Visio that allows Visio to display events on a graphical architecture diagram. A screenshot of this simulator is shown in Figure 4. This shows that our infrastructure tools are useful in high-value semantically oriented tools like simulators.

In addition to scalability, this experience demonstrates several additional benefits of our infrastructure. First, it shows that our base schemas have effective modeling capabilities by themselves, as no extensions were needed to model the AWACS architecture. Second, it shows the flexibility of the infrastructure's tool support, as we were able to use xArchADT (and, by extension, our data binding library) to create the architecture description and use it as the basis for our simulator. Finally, it shows the value of the user-interface based tools in our infrastructure, as we were able to use XML Spy to validate our description and ArchEdit to refine it.

4.2 JPL

As a demonstration of the adaptability of the infrastructure to a new domain with its own unique architectural requirements, we describe our experience with the Jet Propulsion Laboratory (JPL). JPL has adopted our infrastructure 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.

This experience verifies that our infrastructure's genericity and its extensibility mechanisms are useful in as-yet unexplored domains, especially with regard to the xADL 2.0 base schemas. JPL was able to reuse the xADL 2.0 base schemas, creating small new schemas to model domain-specific details of spacecraft software. JPL's use of apigen and the associated data binding library to manipulate architecture descriptions further shows the value of our infrastructure's tool support. Finally, their mapping of architecture descriptions to other representations shows the adaptability of our approach to other, unforeseen situations.

4.3 Mappings to Koala & Mae

To demonstrate our infrastructure's ability to capture concepts from an emerging research area and add tool support for those concepts efficiently, we have created schemas that add the unique modeling constructs of Koala [29] and Mae [16] to our base xADL 2.0 schemas [12]. Koala and Mae are two representation formats for capturing product line architectures. Each product line consists of multiple variants of a software architecture. These

variants may be different configurations of the system for use in different environments, or may represent different stages of the evolution of an architecture over time. The xADL 2.0 schemas already provide basic support for product line architectures with the VERSIONS, OPTIONS, and VARIANTS schemas. However, both Koala and Mae have unique modeling characteristics that differ from those in the xADL 2.0 schemas and from each other.

There are several differences between Koala and xADL 2.0. First, Koala does not support the notion of explicit connectors or versioning. Next, Koala has two constructs not present in xADL 2.0: *diversity interfaces* and *switches*. A diversity interface, representing a point of variation in an architecture, is required to be present on variant components and must be an 'out' interface. A switch is an explicit construct applied at a variation point that creates the connection to the variant component that will be used.

The Mae representation is somewhat closer to xADL 2.0. Two key differences exist between xADL 2.0 and Mae. First, component types in Mae are augmented with a string describing their architectural style. Second, component types in Mae also have a subtype relation and a reference to their supertype.

We addressed these differences using several aspects of our infrastructure. xADL 2.0's flexibility at the level of individual elements was useful several times. For instance, Koala lacks support for explicit connectors, so we simply excluded connectors from our mapping since xADL 2.0 does not require them, demonstrating the modularity of the xADL 2.0 schemas at the level of individual elements. When new constructs were required, like Koala's diversity interfaces and Mae's subtypes, we created simple schemas that added these entities to xADL 2.0.

This experience further demonstrates the effectiveness of the XML-based extensibility mechanism we have chosen for our infrastructure. Consider the following schema, used to add Koala-style diversity interfaces to xADL 2.0 (some tags and namespace information omitted for clarity):

```
<schema xmlns="diversity.xsd">
 <complexType name="DiversityInterface">
  <complexContent>
   <restriction base="Interface">
    <sequence>
     ...
     <!--This is the only element
        that changes-->
     <element name="direction"
              type="Direction"
              minOccurs="0" maxOccurs="1"
              fixed="out"/>
     ...
    </sequence>
    <attribute name="id"
               type="Identifier"/>
   </restriction>
  </complexContent>
 </complexType>

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

This schema subtypes the definition of an interface to create a diversity interface, and extends the definition of a component to add such an interface. Note the relative simplicity of this schema; other schemas (shown in full in [12]) are also simple and straightforward. For the Koala and Mae mappings, we successfully exercised the full gamut of XML schema-based extensibility techniques (creation of new elements, extension of existing elements, restriction of elements, etc.)

This experience also reinforces the effectiveness of the infrastructure tools. We used XML Spy to verify and create our extension schemas and apigen to create a new data binding library that supports them. ArchEdit was able to support these schemas automatically. As such, after writing the short extension schemas required to map Koala and Mae into our infrastructure, our tools provided parsing, syntax checking, and GUI editing abilities automatically.

5. RELATED WORK

Our infrastructure has its roots in several key areas of research and practice. First, research on traditional and domain-specific ADLs indicates the need for new representations and features and the ability to create them efficiently. Second, early research on XML-based ADLs showed that XML could be effectively used to develop ADLs. Finally, UML is a heavyweight, extensible design notation that represents a different way of modeling software systems. Our infrastructure is described in these contexts here.

5.1 Traditional and Domain-Specific ADLs

As noted in Section 2, traditional ADLs (and the wide proliferation thereof) are the inspiration for this work. Additional motivation comes from domain-specific software architecture (DSSA) research, which has shown that architecture description languages tailored to specific domains can increase automation and reduce effort in the software development process. The amount of reuse and abstraction possible in a single, well understood domain, such as avionics [40], far exceeds that possible in the general case.

5.2 XML-based ADLs

XML-based ADLs have been investigated in a limited fashion over the past few years [19][30]. These ADLs are able to take advantage of XML's extensive off-the-shelf tool support., but their reliance on DTDs prevents easy modular extension. The creation of a new ADL requires the development of a "hybrid DTD" that describes how to compose the component DTDs that define the language.

xAcme [36] is a more recent XML-based ADL that represents Acme concepts in a set of XML schemas, based on the xArch core. This ADL is another example of a successful use of XML to develop an ADL. However, it is simply another example of an ADL, and is not part of an infrastructure for experimenting with new ADLs or ADL features—its associated tool set is not meant to adapt to non-xAcme ADLs.

5.3 UML

Work done with the Unified Modeling Language (UML) [35] to model systems is closely related to, but distinct from, the contributions of our infrastructure. First, and most importantly, the intents of our infrastructure and UML (plus its associated tools) are quite different. UML is a rather heavyweight design notation, modeling the full structure and semantics of a software system in seven separate views. In contrast, our infrastructure is geared toward lightweight experimentation and rapid extension,

toward lightweight experimentation and rapid extension, allowing architects to choose and develop constructs that fit a particular need or interest with generic tools and base schemas to support them.

A second distinction can be made between the extensibility mechanisms supported by UML and our infrastructure. Several papers have analyzed UML's suitability as an ADL [33][34][17]. These papers have revealed that raw UML is well-suited to modeling some aspects of software architectures, but fails in modeling others. Therefore, approaches to representing architectures in UML require extensions to UML. An early approach [34] extends UML via changes to the UML meta-model, the model in which UML itself is defined. This approach allows creation of new elements and subtyping of existing ones, but renders UML tools incompatible with the resulting language. A later approach by Robbins et. al. [33] extends existing UML elements with UML's built-in extensibility mechanisms, namely *stereotypes, tagged values*, and *constraints*. These extension mechanisms are part of UML and are well-supported by UML tools. Robbins and Medvidovic found, however, that this is a less than ideal approach because it cannot fully represent all aspects of ADLs. In contrast, our infrastructure leverages mechanisms from XML that provide extensibility comparable to editing the UML meta-model. Most UML tools focus on supporting basic UML system design activities. In contrast, our tools focus on supporting extension and serving as the basis for high-value design tools.

6. CONCLUSIONS

This paper contributes an infrastructure for creating and extending architecture description languages. The infrastructure drastically reduces the amount of effort infolved in experimenting with and developing new architectural concepts. This reduction results from the three parts of our infrastructure: an XML-based extensibility mechanism, a set of generic base schemas, and a set of flexible tools. It eliminates the need to build tools like parsers, syntax checkers, and data bindings for ADLs, allowing researchers to spend more time on building high-value tools that focus on addressing open research issues.

Our infrastructure has demonstrated its effectiveness in a number of projects. We have demonstrated the scalability of our infrastructure and the flexibility of our tools by modeling and simulating the AWACS software architecture. The adaptability of our infrastructure to a new, unexplored domain (spacecraft software) and the effectiveness of our generic xADL 2.0 schemas have been demonstrated by work done at JPL. We have demonstrated that our infrastructure can be used to capture aspects of an emerging research area (product line architectures) with Koala and Mae. Our tools provided a parser, data bindings, and a GUI editor for t those aspects automatically.

In our future research, we plan to expand the xADL 2.0 schemas to include new modeling constructs, particularly those that will support the specification of distributed and dynamic architectures. We are also adding new tools to our infrastructure, including ArchDiff, a tool for determining the difference between two architecture descriptions, and a selector that can select an architectural variant in a product line architecture description. The ultimate goal of our work with this infrastructure is to investigate issues related to distributed, dynamic software architectures, as well as applying architecture-based development to areas of the lifecycle such as deployment and maintenance.

7. URL

More information about our infrastructure can be found here:

http://www.isr.uci.edu/projects/xarchuci/

8. ACKNOWLEDGMENTS

The authors would like to acknowledge and thank our industrial contacts, particularly Will Tracz at Lockheed Martin and Nicolas Rouquette and Vanessa D. Johnson at JPL. We would also like to like to acknowledge the contributions to this work of David Rosenblum, Yuzo Kanomata, Jie Ren, Girish Suryanarayana, Joachim Feise, Kari Nies, Christopher van der Westhuizen, and Ping Chen at UC Irvine. Finally, we would like to thank David Garlan and Bradley Schmerl at Carnegie Mellon University for their collaboration on the development of xArch.

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. Effort also partially funded by the National Science Foundation under grant number CCR-0093489. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.

9. REFERENCES

[1] R. Allen and D. Garlan. *A Formal Basis for Architectural Connection*. ACM Trans. Software Eng. and Methodology, vol. 6, no. 3, pp. 213-249, July 1997.

[2] M. Altheim, F. Boumphrey, S. Dooley, S. McCarron, S. Schnitzenbaumer, and T. Wugofski., eds. Modularization of XHTML. URL: http://www.w3.org/TR/xhtml-modularization/.

[3] Altova GmbH. *XML Spy*. URL: http://www.xmlspy.com/.

[4] Apache Group. Xerces Java Parser Readme. URL: http://xml.apache.org/xerces-j/index.html.

[5] P. Binns, M. Engelhart, M. Jackson, and S. Vestal. *Domain-Specific Software Architectures for Guidance, Navigation, and Control*. International Journal of Software Engineering and Knowledge Engineering, vol. 6, no. 2, 1996.

[6] R. Bourret. *XML Data Binding Resources*. URL: http://www.rpbourret.com/xml/XMLDataBinding.htm.

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation 6 October 2000.

[8] C. Christensen and C. Shaw, eds. *Proc. of the Extensible Languages Symposium*, Boston, May 13, 1969.

[9] P. Clements and L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley 2001.

[10] E. Dashofy, A. van der Hoek, and R. N. Taylor. *A Highly-Extensible, XML-Based Architecture Description Language*. In Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001), Amsterdam, Netherlands.

[11] E. Dashofy. *Issues in Generating Data Bindings for an XML Schema-Based Language.* In Proceedings of the Workshop on XML Technologies and Software Engineering (XSE2001), Toronto, ONT, Canada.

[12] E. Dashofy and A. van der Hoek. *Representing Product Family Architectures in an Extensible Architecture Description Language*. In Proc. of the Int'l Workshop on Product Family Engineering (PFE-4), Bilbao, Spain, October 2001.

[13] C.A.R. Hoare, Communicating Sequential Processes. Prentice Hall, 1985.

[14] A. van der Hoek, D. Heimbigner, and A. L. Wolf. *Investigating the Applicability of Architecture Description in Configuration Management and Software Deployment.* Technical Report CU-CS-862-98, Department of Computer Science, University of Colorado, November 1998.

[15] A. van der Hoek, D. Heimbigner, and A. L. Wolf. *Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage a Trois*. Technical Report CU-CS-862-98, Department of Computer Science, University of Colorado, 1998.

[16] A. van der Hoek, M. Mikic-Rakic, R. Roshandel, and N. Medvidovic. *Taming Architectural Evolution.* In Proceedings of the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), Vienna, Austria, September 2001.

[17] C. Hofmeister, R. L. Nord, and D. Soni. *Describing Software Architecture with UML*. In Proceedings of Working IFIP Conference on Software Architecture, pages 145-160. Kluwer Academic Publishers, February 1999.

[18] International Organization for Standardization. *Information processing—Text and office systems—Standard Generalized Markup Language (SGML)*. ISO 8879:1986. 154pp.

[19] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic and R. Taylor. *xADL: Enabling Architecture-Centric Tool Integration with XML*. Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34), 2001.

[20] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. IEEE Trans. on Software Eng., SE-16, 11 (1990), pp 1293-1306.

[21] O. Lassila and R. Swick. *Resource Description Framework (RDF) Model and Syntax Specification*. W3C Recommendation 22 February 1999.

[22] Le Hors, A., ed. Document Object Model (DOM) Level 3 Core Specification. URL: htttp://www.w3.org/TR/2001/WDDOM-Level-3-Core-20010126/.

[23] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. *Specification and Analysis of System Architecture Using Rapide.* IEEE Trans. Software Eng., vol. 21, no. 4, pp. 336-355, Apr. 1995.

[24] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, *Specifying Distributed Software Architectures*, Proc. Fifth European Software Eng. Conf. (ESEC '95), Sept. 1995.

[25] J. Magee and J. Kramer. Concurrency: State Models & Java Programs. Wiley, 1999.

[26] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor. *A Language and Environment for Architecture-Based Software Development and Evolution*. Proc. 21st Int'l Conf. Software Eng. (ICSE '99), pp. 44-53, May 1999.

[27] N. Medvidovic and R. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 70-93, January 2000.

[28] Megginson Technologies. SAX 2.0: The Simple API for XML. URL: http://www.megginson.com/SAX/.

[29] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. *The Koala Component Model for Consumer Electronics Software*. IEEE Computer 33(3): 78-85 (2000).

[30] Open Group. *Architecture Description Markup Language (ADML), Version 1*. URL: http://www.opengroup.org/onlinepubs/009009899/.

[31] P. Oreizy, N. Medvidovic, and R. Taylor. *Architecture-Based Runtime Software Evolution*. In Proceedings of the International Conference on Software Engineering 1998 (ICSE'98), pages 177-186, Kyoto, Japan, April 19-25, 1998.

[32] D. E. Perry and A. L. Wolf. *Foundations for the Study of Software Architectures*. ACM SIGSOFT Software Engineering Notes, pages 40-52, October 1992.

[33] J. Robbins, N. Medvidovic, D. Redmiles, D. Rosenblum. *Integrating Architecture Description Languages with a Standard Design Method*. In Proc. 20th International Conference on Software Engineering (ICSE'98), Kyoto, Japan.

[34] J. Robbins, D. Redmiles, D. Rosenblum. *Modeling C2 in the Unified Modeling Language*. In Proc. California Software Symposium 1997.

[35] J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1998.

[36] B. Schmerl. *xAcme: CMU Acme Extensions to xArch.* URL: http://www-2.cs.cmu.edu/~acme/pub/xAcme/guide.pdf.

[37] Stephen A. Schuman, ed. *Proceedings of the International Symposium on Extensible Languages*, Grenoble, France, September 6-8, 1971.

[38] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. Nies, P. Oreizy and D. Dubrow. *A Component- and Message-Based Architectural Style for GUI Software*. IEEE Transactions on Software Engineering, June 1996.

[39] H. Thompson, D. Beech, M. Maloney and N. Mendelsohn, eds. XML Schema Part 1: Structures. URL: http://www.w3.org/TR/xmlschema-1/.

[40] W. Tracz and L. Coglianese. *An Avionics Domain-Specific Software Architecture.* Crosstalk, October 1992, pp. 22-25.

[41] U.S. Air Force, *AWACS E-3 Sentry Fact Sheet*. URL: http://www.af.mil/news/factsheets/E_3_Sentry__AWACS_.html.