A Comprehensive Approach for the Development of Modular Software Architecture Description Languages

ERIC M. DASHOFY, ANDRÉ VAN DER HOEK, and RICHARD N. TAYLOR
University of California, Irvine

Research over the past decade has revealed that modeling software architecture at the level of components and connectors is useful in a growing variety of contexts. This has led to the development of a plethora of notations for representing software architectures, each focusing on different aspects of the systems being modeled. In general, these notations have been developed without regard to reuse or extension. This makes the effort in adapting an existing notation to a new purpose commensurate with developing a new notation from scratch. To address this problem, we have developed an approach that allows for the rapid construction of new architecture description languages (ADLs). Our approach is unique because it encapsulates ADL features in modules that are composed to form ADLs. We achieve this by leveraging the extension mechanisms provided by XML and XML schemas. We have defined a set of generic, reusable ADL modules called xADL 2.0, useful as an ADL by itself, but also extensible to support new applications and domains. To support this extensibility, we have developed a set of reflective syntax-based tools that adapt to language changes automatically, as well as several semantically-aware tools that provide support for advanced features of xADL 2.0. We demonstrate the effectiveness, scalability, and flexibility of our approach through a diverse set of experiences. First, our approach has been applied in industrial contexts, modeling software architectures for aircraft software and spacecraft systems. Second, we show how xADL 2.0 can be extended to support the modeling features found in two different representations for modeling product-line architectures. Finally, we show how our infrastructure has been used to support its own development. The technical contribution of our infrastructure is augmented by several research contributions: the first decomposition of an architecture description language into modules, insights about how to develop new language modules and a process for integrating them, and insights about the roles of different kinds of tools in a modular ADL-based infrastructure.

This work was 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. The work was also partially funded by the National Science Foundation under Grant Nos. CCR-0093489 and IIS-0205724. 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 National Science Foundation, the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.
Authors' address: Institute for Software Research, University of California, Irvine, CA. 92697; email: {edashofy,andre,taylor}@ics.uci.edu.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms: Design, Languages

Additional Key Words and Phrases: Architecture description languages, XML, xADL 2.0, ArchStudio 3

1. INTRODUCTION

Software architecture-based development addresses software at a higher-level of abstraction than objects or lines of code [Perry and Wolf 1992]. At a minimum, software architecture deals with software systems consisting of components (loci of computation), connectors (loci of communication), and configurations (arrangements of components and connectors, and properties of that arrangement). At this level of abstraction, many aspects of a software system can be modeled: its topology, behavior, usage scenarios, deployment profile, and so on. Each of these aspects may also be modeled at different levels of detail or in different ways. Each way of modeling can bring its own advantages, perhaps providing a new way to analyze or simulate the system or to explain the system's structure to other developers.

Despite these advantages, software architecture research has yet to make a significant impact on software engineering in practice. We believe that this can be partially attributed to the fact that current techniques and tools for modeling software architectures (and leveraging the information contained in those models) have failed to meet the needs of software engineering practitioners. Making software architecture cost-effective requires that the notations, tools, and techniques used model the important aspects of a system at the right level of detail.

The result of a decade of research has been a plethora of notations for representing software architectures, usually in the form of architecture description languages (ADLs) [Medvidovic and Taylor 2000]. In general, the notations developed to date each support a single research goal. However, to be useful in a real-world context, software architecture research must move beyond single-purpose notations. Architectures must capture many aspects of a software system simultaneously. Furthermore, the set of aspects that are important enough to model varies from domain to domain. For example, embedded systems projects may require extensive, detailed modeling capabilities for timing and power consumption, while distributed systems may require modeling aspects of fault tolerance and bandwidth usage. Despite this wide spectrum of needs across domains, however, many projects will also share some modeling needs. For example, both embedded and distributed system architects may be concerned with tracking and managing the evolution of their architectures across product releases. Projects within a domain will typically exhibit even larger sets of common concerns.

The wide differences in modeling priorities and needs across domains indicates that a satisfactory "one-size-fits-all" ADL is unlikely to emerge. However, the significant commonalities that exist, even among disparate domains,

indicate that developing new notations for each domain (or project) from scratch means consistently reinventing the wheel. Therefore, a middle ground is necessary: one that allows architects to choose from the common architecture-level aspects of the software system those aspects they want to model and to easily add to those their own concerns.

We hypothesize that such a middle ground exists in the form of an infrastructure in which ADLs can be quickly constructed, combining compatible modeling features and allowing explorative integration of new features as necessary. These ADLs would be modular rather than monolithic: modeling features would be encapsulated in modules and modules would be composed into ADLs. This will reduce the amount of effort it takes for architects to obtain a notation that helps them to model the "right" aspects of their systems at the "right" level of detail.

This raises many interesting research questions. What is an effective set of technologies to create a modular notation? How can ADL features be effectively modularized and later composed? How can feature modules be developed so as to maximize reusability and compatibility with other modules that may be independently developed? What is the process by which architects can develop new features (or new ADLs), and how can tools support this process? How can tools be constructed in this context to maximize their reusability and applicability?

To answer these questions, we have taken an empirical approach. First, we have created an infrastructure for the creation and use of modular ADLs. This infrastructure provides:

(1) an XML-based modular extension mechanism for defining ADLs;
(2) a base set of features that can be reused in ADL development, supporting design-time and run-time modeling, implementation mappings, and product lines; and
(3) a flexible set of tools to support ADL development and use.

Second, we followed up the creation of this infrastructure with four evaluation cases in different domains, both to test the infrastructure and to give us additional insights regarding the research questions at hand. The results of these experiences indicate that our infrastructure is generic, extensible, scalable, capable of modeling advanced features from other ADLs, and capable of supporting its own development.

The primary research contributions of our approach are the first decomposition of an architecture description language into modules, insights about how to develop new language modules and a process for integrating them, and insights about the roles of different kinds of syntactic and semantic tools in a modular infrastructure. In addition, the infrastructure itself represents a technical contribution in the form of modeling features and tools that can be used in practice. Use of this infrastructure results in a significant overall reduction in effort compared to developing ADLs and supporting tools from scratch. Tool support for extensible ADLs is especially important. While it may be relatively easy to change the *definition* of a language, adapting tools to support a modified language has been costly. We take this into account by using a two-tiered approach.

A first layer of tools is primarily syntactic: these tools generate their behaviors and interfaces from XML schemas directly, adapting to language changes and new modules automatically. A second layer of tools is primarily semantic: these tools leverage the syntactic tools, but also make assumptions about the meanings of elements specified in language modules. Because of this, the semantic tools are generally more capable and user-friendly than the syntactic tools at the cost of changes and updates when the underlying language changes.

A research question we chose not to address is how to ensure semantic compatibility among ADL features. Our approach primarily addresses syntactic compatibility and extensibility. Semantics are a more complex issue. The feature interaction problem [Zave 1999] has been described as the primary problem in extensible language development [Berners-Lee and Connolly 1998; Peake and Salzman 1997]. In our approach, ensuring semantic compatibility among language features is the responsibility of the ADL developer. This is not unique to our approach; developers who create new ADLs from scratch must also address feature interactions.

The remainder of this article is organized as follows. Section 2 provides background material about software architectures, ADLs in general, and XML. Section 3 provides a high-level introduction to our approach. Section 4 discusses our set of reusable ADL features. Section 5 details the tools we have developed to support ADL development. Section 6 discusses how our infrastructure has been applied in several domains. Section 7 summarizes our main research results and describes the insights we developed while building and using our infrastructure. Section 8 compares our work to related projects. Finally, Section 9 concludes the article and discusses future work.

2. BACKGROUND

Understanding our approach requires a background in software architecture research, the large body of previous work in developing software architecture description languages (ADLs), and some information on XML itself.

2.1 Software Architecture

Since the development of abstraction and modularization in software development, abstract models (formal or otherwise) of a system's structure have played a key role in the development of large software systems. As the complexity of systems has increased, support for more and more abstract concepts in programming languages and environments has also increased. Object- oriented development, for example, provides a useful abstraction above the level of program statements by encapsulating related functionality and state in a single construct: the object. Software architecture research has resulted in a level of abstraction above that of simple objects, modules, or lines of code [Perry and Wolf 1992]. Architecture descriptions of software systems are generally composed of at least three key entities: components, connectors, and configurations [Medvidovic and Taylor 2000].

Components are the loci of computation in an architecture, and may be stateful. In this way, they resemble objects, but components and objects differ in

several ways. Components tend to be larger than objects, often encapsulating an entire off-the-shelf library or tool. Components are rarely created and destroyed in the life of a software system, objects are created and destroyed constantly [Szyperski 1997].

Connectors are the loci of communication in an architecture. Connectors may be implicit, like a procedure call, or explicit, resembling a component. Simple connectors may be implemented as a basic message queue or API entrypoint, complex connectors may encapsulate middleware or cross process boundaries. This can blur the distinction between components and connectors; a good rule of thumb is that connectors may change the form or syntax of the data they convey but generally do not affect application-level semantics; that is, they do not change the meaning of the data [Mehta et al. 2000].

Configurations describe how components and connectors are arranged, as well as the properties of that particular arrangement. Most commonly, this is expressed through a set of explicit component-to-connector links that define an architectural topology. Alternatively, it may include constraints or patterns on arrangements of components and connectors, or how they behave.

2.2 Architecture Description Languages

Beyond components, connectors, and configurations, a description of a software system at the architectural level can contain almost any kind of information. The kinds of information present in an architecture description are influenced by the domain in which the software is being developed and how the architecture description is being used.

The proliferation of specialized architecture description languages over the past decade confirms this. Wright [Allen and Garlan 1997] extends component and connector specifications with behavioral information in the language of communicating sequential processes (CSP) [Hoare 1978] so that architectures can be analyzed. Rapide [Luckham et al. 1995] describes components with partially ordered event sets called POSETs so that their behavior can be simulated. Darwin [Magee et al. 1995] describes configurations of systems that are distributed across multiple processes and machines.

Despite the fact that these notations share many common conceptual elements (components, connectors, interfaces, links, and so on), they do not share even a syntactic basis. This renders each notation's tools incompatible with the others. Other ADLs such as Weaves [Gorlick and Razouk 1991] and MetaH [Binns et al. 1996] are similarly incompatible. To compound the problem, all these languages lack support for extensibility, so adding features to any of them would require significant changes to all its supporting tools.

The most notable exception to this from the architecture community is Acme [Garlan et al. 2000] which was developed as an architectural interchange language. The Acme core consists of basic types of constructs that occur in practically every architecture notation: components, connectors, interfaces (called ports and roles), links (called connections), and so on. Each of these core entities is decorated with a set of arbitrary name-value pair properties. This strategy allows some extensibility, but its usefulness is hampered by several constraints.

```
<name lang="en-us">
  <first>John</first>
  <last>Doe</last>
</name>
```

Fig. 1. Sample marked-up text in XML.

(1) It is not possible to extend the core set of elements (components, connectors, ports, roles) to add new first-class entities. This prohibits architectural constructs that are not naturally represented as decorations on one of the core elements such as component version trees.
(2) It is difficult to encode and manage complex data structures as sets of name-value pair properties.
(3) Acme is not supported by a metalanguage that allows description of allowable properties and their format.

Thus, Acme's ability to serve as a general-purpose interchange language or extensible ADL is diminished.

2.3 XML as a Basis for an Extensible Modeling Language

Constructing an extensible modeling language for software systems requires a way to define allowable constructs in the language and a way to extend or change constructs in the language to add or modify features. Many metalanguages and environments exist in which an extensible, modular language can be created, among them XML [Bray et al. 1998], Lisp [Steele 1990], GME [Ledeczi et al. 2000], and DOME [Honeywell Inc. 1999]. From a technical and theoretical perspective, we could have used any of these environments to develop our infrastructure (although they are not all equivalent; technical differences between these approaches are discussed further in Section 8.2). From a practical perspective, however, XML has far more support from standards committees, tool vendors, and practitioners than any other alternative. XML is platform-neutral, unbound to any particular hardware or network architecture. Many ancillary standards support XML, such as XLink [DeRose et al. 2001], which allows linking within and between XML documents, and XPath [Clark and DeRose 1999], which allows indexing of specific elements and attributes within a document. A plethora of off-the-shelf tools for constructing, visualizing, storing, and manipulating XML documents are available. This is not to say that XML is by any means perfect even for our application. Specific drawbacks of using XML are detailed in Section 7.3. We believe, however, that XML's support for modular extensibility and extrinsic benefits far outweigh its technical drawbacks.

XML documents are text documents in which some of the text is marked up using specially formatted tags that define the beginning and end of a segment of marked-up text. Data delimited by a start and end tag is known as an *element*; start tags can contain additional annotations called *attributes*. Data marked up with elements and attributes is shown in Figure 1.

It is possible to introduce a grammar of elements and attributes over XML documents using one of several available metalanguages. Several

metalanguages for defining XML grammars are available, two of which have been standardized by the World Wide Web Consortium (W3C): DTDs and XML schemas [Fallside 2001]. DTDs offer limited mechanisms for supporting modular extensibility as evidenced in the *Modularization of XHTML* standard [Altheim et al. 2001]. However, XML schemas are much more capable in this regard, offering an easy way to define constructs in one schema and extend them in another through a mechanism very similar to object-oriented subtyping. A subtype can add new elements and attributes to the content model of its supertype. Unlike in object-oriented types, however, XML schema subtypes can also be more restrictive than their supertypes in certain ways. Specifically, they can restrict the cardinality of elements in their supertype. For example, if an element in the supertype has cardinality 0-*n*, the cardinality of the element in a restricted subtype could be 0-0, eliminating the element entirely.

3. APPROACH

Existing architecture description languages are either too specific (e.g., single-purpose ADLs like Rapide, Darwin, or Koala) or too general (e.g., Acme). Furthermore, the cost to adapt a notation and its tools to support a new feature—even when that feature only slightly extends the capabilities of the target ADL—is prohibitively expensive. To remedy this, we have developed an infrastructure supporting modular extensibility. Our objectives in building this infrastructure were:

—It must place as few limits on what can be expressed at the architecture level as possible;
—It should allow new modeling features to be added and existing features to be modified over time;
—It should allow experimentation with new features and combinations of features;
—It should provide a library of generically useful modules applicable to a wide variety of domains;
—It should allow modeling features, once defined, to be reused in other projects; and
—It should provide tool-builders with support for creating and manipulating architecture models, even when the underlying notation can and will change.

The core elements of our infrastructure are shown in Figure 2. At the base of the infrastructure is a collection of language modules, realized in our implementation as XML schemas. These schemas define modeling constructs and extend modeling constructs from other schemas. Some of these schemas are relatively generic, containing modeling features applicable to many domains. In our infrastructure, these generic schemas are collectively called xADL 2.0 and are described in Section 4. Other schemas may be more domain-specific such as those described in Section 6.2. Decomposing features into individual schemas is nontrivial and is (we believe) the key contributor to maintaining the flexibility of this infrastructure. Above this language layer is a layer of syntax-oriented tools. These tools leverage the syntax defined in the schemas to automatically

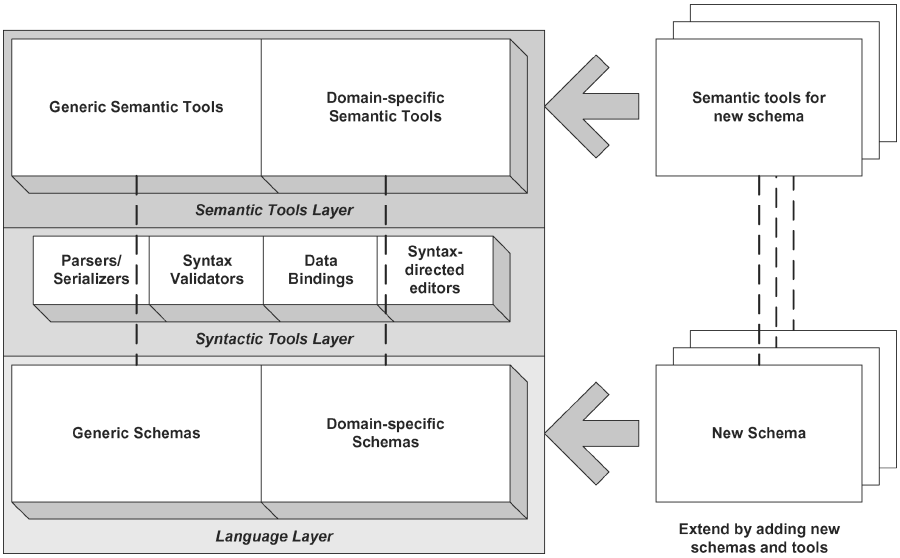


Fig. 2. Approach overview depicting the relationship between language modules as realized in XML schemas, syntactic tools that work on any schemas, and schema-specific semantic tools.

provide services that are generically useful regardless of the contents of any particular schema—these include parsers, serializers, data bindings, and syntax-directed editors. The primary advantage of these tools is that they do not need to be changed when new schemas are added to the language layer. Finally, the infrastructure contains a layer of semantic tools. These tools are built to support features of specific schemas in the language layer and generally use the syntax-oriented tools to read and write architecture descriptions. These tools may include visual editors, analysis tools, code generators, and so on. Because these tools are bound to specific schemas and schema features, they may need to be changed when the schemas in the language layer change.

This infrastructure promotes a straightforward process for software architecture-based development. First, the architect looks for an ADL (in this case, a composition of schemas) that has already been developed that meets the needs of the current project. If no such ADL exists, then the closest ADL (the one with the most features/modules applicable to the current project) is chosen. Unneeded modules can be removed. Additional features required can come from one of three sources: by reusing modules from other ADLs, by extending existing modules, or by developing new modules if no suitable basis exists. Obviously, reusing or extending existing modules is preferable to developing new ones because of the potential to also reuse the accompanying engineering knowledge and, more importantly, semantic tools that were developed along with the original modules.

4. xADL 2.0: A GENERIC BASE FEATURE SET

Architecture description languages must minimally provide methods for describing components, connectors, interfaces, and links. Beyond this core, each

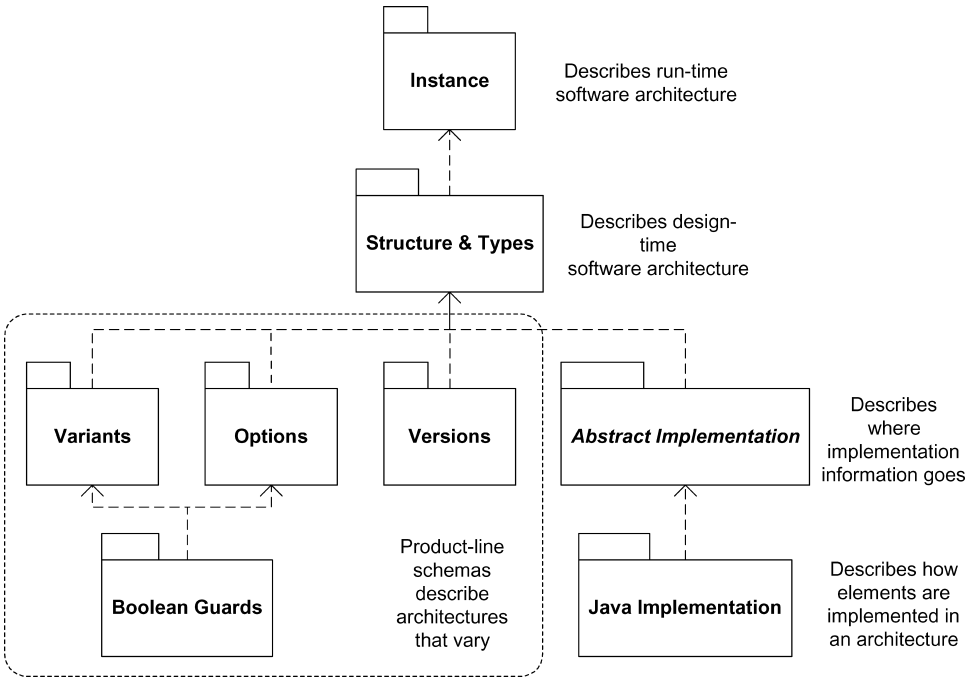


Fig. 3. xADL 2.0 XML schemas and their relationships.

domain will have unique modeling needs. However, there are some ADL features that are generally applicable and desirable across many diverse domains. We believe that well-designed implementations of these features that are syntactically and semantically compatible with each other and the core constructs are a key part of a successful ADL development infrastructure.

Accordingly, we have used XML schemas to define a set of generic constructs that are useful for modeling software architectures across many domains. These constructs can be used as is, or they can be extended to support additional modeling features. Instead of defining all these generic constructs in one large, monolithic XML schema, we have elected to group them in schemas based on their intended purpose. The result is a set of reusable "modules" that together comprise a generic ADL, known as xADL 2.0 [Dashofy et al. 2001]. The current set of xADL 2.0 schemas and their dependency relationships are shown in Figure 3[1].

Cognizant that these schemas will be used in many domains, we have attempted to keep them as generic as possible. For example, xADL 2.0 defines basic constructs like components and connectors but does not dictate how they must behave or how they can be linked together. These characteristics vary

[1]Figure 3 shows the conceptual dependencies between xADL 2.0 schemas. Because the current XML schema standard does not support multiple inheritance of elements, we have occasionally needed to introduce some syntactic dependencies between schemas; however, we have attempted to minimize or eliminate the semantic impact of these artificial dependencies. This is discussed in detail in Section 7.3.

Table I. xADL 2.0 Schemas and Features

Purpose	Schema	Features
Design-time and Run-time Modeling	INSTANCES	Run-time component, connector, interface, and link instances; subarchitectures, general groups.
	STRUCTURE & TYPES	Design-time components, connectors, interfaces, links, subarchitectures, general groups.
Implementation Mappings	ABSTRACT IMPLEMENTATION	Placeholder for implementation data on components, connectors, and interfaces.
	JAVA IMPLEMENTATION	Concrete implementation data for Java components, connectors, and interfaces.
Architecture Evolution Management and Product-Line Architectures	VERSIONS	Version graphs for component, connector, and interface types.
	OPTIONS	Optional design-time components, connectors, interfaces, and links.
	VARIANTS	Variant design-time component and connector types.

from ADL to ADL so they can be specified in extension schemas. Depending on the situation and the requirements of the model, however, these generic features can be sufficient on their own as is evidenced in our experiences with them (see Section 6).

xADL 2.0's core, generic features 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 I. We discuss each schema in detail in this section. To illustrate how these schemas are applied to describe a software system's architecture, we will use xADL 2.0 to describe a hypothetical software system for a consumer electronics product.

4.1 Separation of Design-Time and Run-Time Views

Traditional ADLs have viewed software architecture as a design-time artifact, focusing their modeling capabilities on the design phase of development. However, research into run-time software evolution and dynamism has shown that it is useful to maintain an architectural model of the system at run-time as well [Oreizy et al. 1999; Schmerl and Garlan 2002]. The design-time and run-time models of a system will be similar but not identical. At design time, a system model may contain basic metadata about elements (author, size, textual descriptions), specification of intended (not observed) behavior, and constraints

on the arrangements of elements. In contrast, run-time aspects of a system that might be captured in an ADL include the physical distribution of the software system across machines, the process ID in which each component runs, or the current state of a particular connector (e.g., running, processing messages, blocked).

Two schemas support the design-time/run-time separation in xADL 2.0. These are called the Structure & Types and INSTANCES schemas, respectively. Both schemas support the following features.

—*Components*: Components are the loci of computation in the architecture. In these schemas, components are defined generically; that is, they have only a unique identifier and a short textual description, along with a set of interfaces (described in the following).

—*Connectors*: Connectors are the loci of communication in the architecture. Similar to components, connectors also have only a unique identifier, a textual description, and a set of interfaces.

—*Interfaces*: Interfaces are components' and connectors' portals to the outside world; what we term "interfaces" are also known as "ports" in other ADLs. For example, if a component implements a particular API, it would likely contain an interface indicating that other components can make calls to that API on the target component. In these schemas, interfaces have a unique identifier, a textual description, and a direction (an indicator of whether the interface is provided, required, or both). Specific interface semantics are not specified in xADL 2.0 but in extensions, see Section 6.2 for an example of this.

—*Links*: Links are connections between elements that define the topology of the architecture. In most architecture notations, links connect interfaces, but this constraint is not mandated.

—*Subarchitectures*: Components and connectors may be atomic (not broken down further at the architectural level) or composite. Composite components and connectors have internal architectures, called subarchitectures.

—*General Groups*: Groups are simply collections of pointers to elements in the architecture description. In these schemas, a group has no semantics. Groups with specific meanings (e.g., common authorship, common platform, similar functionality) can be specified in extension schemas.

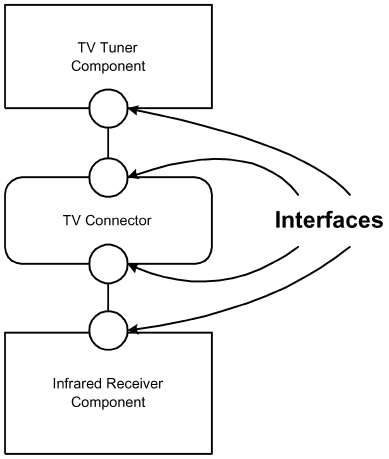
Because the INSTANCES and STRUCTURE & TYPES schemas have definitions for the basic architectural constructs (components, connectors, etc.), they are the core of xADL 2.0. By maintaining definitions of these elements in both schemas, they can be extended separately: the INSTANCES schema should be extended to provide additional run-time data, and the STRUCTURE & TYPES schema should be extended to provide additional design-time data. To provide traceability, XLinks link run-time elements to their design-time counterparts.

To see how these schemas can be used to model the core of a software architecture, consider a software system that might be used to drive a low-end television set. Such a device might have only two software components, one to interface with the TV tuner, and one to drive the infrared detector used to pick up signals from the remote control. These two components are connected by a

```
<xArch>
  <archStructure>
    <component id="tuner">
      <description>
        TV Tuner Component
      </description>
      <interface id="tuner.channel">
        <description>
          ChangeChannel Interface (in)
        </description>
        <direction>in</direction>
      </interface>
    </component>

    <component id="ir">
      <description>
        Infrared Receiver Component
      </description>
      <interface id="ir.channel">
        <description>
          ChangeChannel Interface (out)
        </description>
        <direction>out</direction>
      </interface>
    </component>

    <connector id="tvconn">
      <description>
        TV Connector
      </description>
      <interface id="tvconn.in">
        <description>
          ChangeChannel Interface (in)
        </description>
        <direction>in</direction>
      </interface>
      <interface id="tvconn.out">
        <description>
          ChangeChannel Interface (out)
        </description>
        <direction>out</direction>
      </interface>
    </connector>
```




```
    <link id="link1">
      <description>
        Tuner to Connector
      </description>
      <point>
        <anchor href="#tuner.channel"/>
      </point>
      <point>
        <anchor href="#tvconn.out"/>
      </point>
    </link>

    <link id="link2">
      <description>
        Connector to IR
      </description>
      <point>
        <anchor href="#tvconn.in"/>
      </point>
      <point>
        <anchor href="#ir.channel"/>
      </point>
    </link>
  </archStructure>
</xArch>
```

Fig. 4. Diagram of an example architecture for a television accompanied by its xADL 2.0 design-time structural description (in abbreviated, non-XML notation).

software connector that allows the infrared receiver component to send signals to the TV tuner to change the channel. A graphical depiction of this architecture, accompanied by its xADL 2.0 description (with ancillary mark up elided) is shown in Figure 4.

4.2 Design-Time Type System

Typing is an important construct in most ADLs. xADL 2.0 incorporates the base structures of a typing system that supports type equality and composition. In xADL 2.0, the use of types is optional. Each component, connector, or interface

can optionally contain an XML link to a type; multiple elements can share a type. The type serves as a construct where common properties of elements can be specified once (in the type) instead of in each element declaration. In the STRUCTURE & TYPES schema, these types consist of a unique identifier and a textual description along with a set of signatures. Signatures are prescribed interfaces; two components or connectors of the same type should have the same types of interfaces.

To demonstrate this, we will add types to our television example. We will also add another component to our television, a picture-in-picture tuner. Televisions with picture-in-picture require two tuners to support the display of both channels simultaneously, but since the main tuner and the picture-in- picture tuner are basically identical, they will share a type. The additions to our existing architecture and its description are shown in Figure 5.

Types are also used in xADL 2.0 to support design-time subarchitectures (that is, components or connectors that have internal architectures also specified in xADL 2.0). Thus, two components or connectors that share a type also share an internal architecture. Specifically, types can have an optional XLink to another ArchStructure element describing the internal architecture, as well as a set of mappings between signatures on the type and interfaces on components and connectors in the subarchitecture. These mappings serve to link the outer architecture with the inner one. A full xADL 2.0 depiction of subarchitectures is beyond the scope of this paper, but several are available on our websites, see Section 10 for the URLs.

Some of our intended semantic constraints for types (for example, that a component's type link point to a component type or that interfaces match their prescriptions as expressed in signatures) cannot be expressed directly in the XML schemas. However, we have built checking tools and environments that help to guide users into following and maintaining these constraints in their own architectures. These are discussed further in Section 5.2.

4.3 Implementation Mappings

A second important feature of xADL 2.0 is its support for mapping design-time architectural elements onto executable code. Several ADLs such as MetaH [Binns et al. 1996] 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 and can help to manage the transition from system design to implementation.

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 used. Obvious possibilities include Java classes and archives, Windows DLLs, UNIX shared libraries, and CORBA components [Object Management Group 2001], 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. This indicates to future developers how to extend the xADL 2.0 schemas

```
<xArch>
  <archStructure>
    <!--Contents from Figure 4 above
        included here, but omitted for
        space. Also, each component,
        connector & interface would have a
        type link, as shown below.-->

    <component id="piptuner">
      <description>Pic in Pic Tuner
        </description>
      <interface id="pinptuner.channel">
        <description>ChangeChannel
          Interface (in)</description>
        <direction>in</direction>
        <type href="#changechannel_type"/>
      </interface>
      <type href="#tuner_type"/>
    </component>

    <link id="link3">
      <description>PinP Tuner to Conn
        </description>
      <point>
        <anchor href="#piptuner.channel"/>
      </point>
      <point>
        <anchor href="#tvconn.out"/>
      </point>
    </link>
  </archStructure>

  <archTypes>
    <componentType id="tuner_type">
      <description>TV Tuner Type
        </description>
      <signature id="tuner_type.channel">
       <description>ChangeChannel Sig (in)
          </description>
        <direction>in</direction>
        <type href="changechannel_type"/>
      </signature>
    </componentType>

    <componentType id="ir_type">
      <description>Infrared Receiver Type
        </description>
      <signature id="ir_type.channel">
        <description>ChangeChannel Sig
          (out)</description>
        <direction>out</direction>
        <type href="changechannel_type"/>
      </signature>
    </componentType>

    <connectorType id="tvconn_type">
      <description>TV Connector Type
        </description>
      <signature id="tvconn_type.in">
        <description>ChangeChannel Sig (in)
          </description>
        <direction>in</direction>
        <type href="changechannel_type"/>
      </signature>

      <signature id="tvconn_type.out">
        <description>ChangeChannel
          Sig (out)</description>
        <direction>out</direction>
        <type href="changechannel_type"/>
      </signature>
    </connectorType>

    <interfaceType id="changechannel_type">
      <description>Change Channel
        Interface</description>
    </interfaceType>
  <archTypes>
</xArch>
```

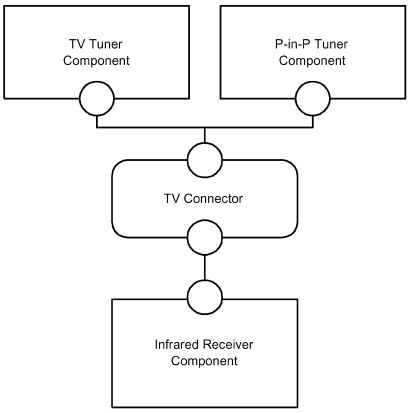


Fig. 5. Diagram for an expanded version of the example television architecture, adding a picture-in picture component, accompanied by additions to our earlier xADL 2.0 description.

to add data for a new implementation platform. 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

```
<xArch>
  <archStructure>
    ...
  </archStructure>

  <archTypes>
    <componentType id="tuner_type">
      <description>TV Tuner Type</description>
      ...
      <implementation>
        <mainClass>
          <javaClassName>edu.uci.isr.tv.TelevisionTuner</javaClassName>
          <url>http://www.isr.uci.edu/classes/tuner.jar</url>
          <initializationParameter>
            <name>tv_model</name>
            <value>Astro 5000</value>
          </initializationParameter>
        </mainClass>
        <auxClass>
          <javaClassName>edu.uci.isr.tv.TelevisionUtils</javaClassName>
          <url>http://www.isr.uci.edu/classes/tuner.jar</url>
        </auxClass>
      </implementation>
    </componentType>
  </archTypes>
</xArch>
```

Fig. 6. Expanded description of one of the television component types, adding implementation details.

schema. xADL 2.0 includes a JAVA IMPLEMENTATION schema that concretely defines a mapping from components, connectors, and interface types to Java classes; we are currently working on additional implementation schemas for mapping to source code as well.

With this setup, determining the concrete implementation for a given element is straight-forward. For a design-time element like a component or a connector, the user simply follows the element's type XLink and gets the implementation data from the type. Run-time elements like component instances and connector instances require an additional step, following the XLink from the run-time element to the design-time element, and then following the design-time element's XLink to its type.

We can add implementation data to component and connector types in our television example, as shown in Figure 6. Here, we show that the television tuner component is implemented by two Java classes residing in the same JAR archive. The main class takes an initialization parameter of the television's model, which is useful if the component's implementation is multipurpose or reusable in multiple contexts (e.g., for many television models).

4.4 Modeling Architecture Evolution and Product Lines

Many first-generation ADLs focused on modeling the architecture of a single software system or product. This is inadequate for two reasons. First,

architectures evolve over time. As a product evolves, so does its architecture. Second, software products are often part of a larger product line. A product line is a family of related software products that share significant portions of their architectures with specific points of variation [Bosch 1999; Ommering 2002; Tracz and Coglianese 1993]. A single product line may contain products that are localized for specific regions or represent different feature sets for marketing purposes (e.g., Standard, Professional, Enterprise).

From a modeling perspective, both architectural evolution and product lines can be addressed by applying traditional concepts from configuration management to software architectures. Inspired by previous research in modeling architectural product lines [Bosch 2000; Clements and Northrop 2001; Ommering et al. 2000], xADL 2.0 integrates these concepts in the form of three schemas: the VERSIONS, OPTIONS, and VARIANTS schemas. 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 a group of elements (components, connectors, links, and so on). Variants indicate points in an architecture where one of several alternatives may be substituted for an element or group of elements. We developed these three schemas to be compatible such that modelers can choose to use only the VERSIONS, OPTIONS, or VARIANTS schema, or any combination of the three.

4.4.1 *Versions.* The VERSIONS schema adds versioning constructs to xADL 2.0. It defines version graphs for component, connector, and interface types. In xADL 2.0, architecture element types are the versioned entities. The decision to version types rather than elements such as components and connectors was made because we feel that it best matches the semantics of the type system. For example, by versioning types, it is possible to have multiple instances of the same version of a component, connector, or interface by simply creating multiple instances of a type. Different versions of a component, connector, or interface can coexist in an architecture as well simply by creating instances of type that share a version tree. We believe this makes sense because, as architectures evolve, newer versions of element (types) may have different characteristics than older versions (e.g., additional signatures).

The relationship between concrete elements (e.g., components), their types, and version graphs is depicted in Figure 7. Here, the TV Tuner Type and the HDTV Tuner Type share a common lineage. The HDTV Tuner Type is a later version of the TV Tuner Type, but both versions may be included in the same architecture if necessary.

In this example, the version graph describes the evolution of a single element. However, using the type-based subarchitecture mechanism defined in the STRUCTURE & TYPES schema, a version graph can capture the evolution of groups of elements or whole architectures. This is another reason why we chose to version types.

In keeping with the generic nature of xADL 2.0 schemas, version graphs do not constrain the relationship between different versions of individual elements, for instance, that they must share some behavioral characteristics or

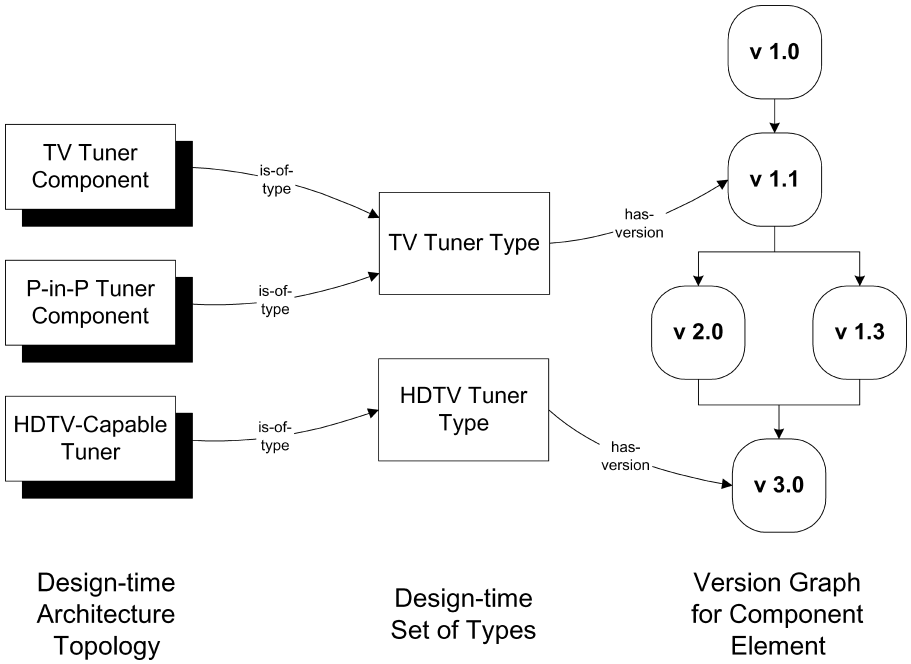


Fig. 7. Relationships between concrete elements, their types, and version graphs.

interfaces. Such constraints may be specified in extension schemas and checked with external tools.

4.4.2 *Options.* The OPTIONS schema allows design-time components, connectors, and links to be labeled as optional in an architecture. Optional elements are accompanied by a "guard condition" whose format can be specified in an extension. xADL 2.0 provides a default schema for guards, the BOOLEAN GUARD schema, that allows guards to be specified as Boolean expressions similar to those found in modern programming languages. If the guard condition is satisfied when evaluated, then the optional element is included in the architecture; otherwise it is excluded.

In our television example, we may wish to turn our single product architecture into a product line by adding optionality. By making the picture-in-picture tuner optional along with its link to the TV connector, we can create a product line that describes two products: a television with picture-in-picture and a television without it. The changes that we must make to our existing product description are shown in Figure 8. Here, we add an `<optional>` element to both the picture-in-picture tuner and its link to the connector. The guard condition for both `<optional>` elements is `hasP_in_P == true`, so these elements will only be included in the architecture if the target product has picture-in-picture (i.e., the variable `hasP_in_P` is bound to the value `true`). Variable-value bindings are established in a selector tool, see Section 5.2.3.

Note that it is possible to express architectures with guards that could result in incomplete or incorrect architectures (e.g., if the link's guard in the example

```
<xArch>
  <archStructure>
    ...
    <component id="pinptuner">
      <description>Pic in Pic Tuner
        </description>
      ...
      <optional>
        <guard>
          <booleanExp>
            <equals>
              <symbol>hasP_in_P</symbol>
              <value>true</value>
            </equals>
          </booleanExp>
        </guard>
      </optional>
    </component>
    <!--P-in-P tuner-to-connector link
        is also optional with same guard
        condition-->
  </archStructure>
</xArch>
```

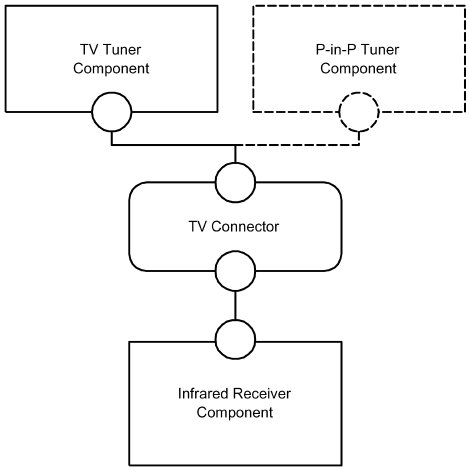


Fig. 8. Diagram for a television product-line where the picture-in-picture elements are optional, accompanied by xADL 2.0 description.

above were missing or different than the guard on the P-in-P tuner). When the product line is sculpted down to a single product via use of our selector tool, these inconsistencies will become apparent and many of them (such as a dangling link) will be caught by our design critics (see Section 5.2.1).

4.4.3 *Variants.* The VARIANTS schema allows the types of certain design-time constructs to vary in an architecture. In particular, it defines variant component and 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 must be mutually exclusive. When a guard condition is met, its associated component or connector type is used in place of the variant type.

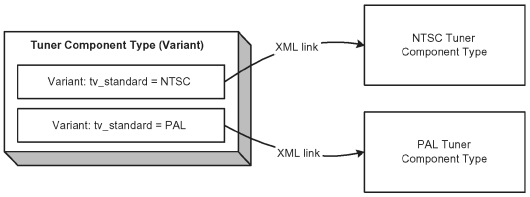
Let us diversify our television product line more by expanding it to multiple international markets. Televisions distributed to North America or Japan will need NTSC tuners, while televisions distributed to most of Europe will need PAL tuners. We can express this by replacing the existing tuner type with a variant type that links to two possible concrete types: an NTSC tuner type and a PAL tuner type. This is shown in Figure 9. In this example, the structural description of the architecture does not change at all: both the main and picture-in-picture tuners retain the link to the same type (tuner_type). Now, however, their type has become variant: they may be an NTSC or PAL tuner, depending on the situation.

5. TOOL SUPPORT

An ADL's usefulness is closely tied to the amount of tool support available for that ADL. Tools are needed to create architecture descriptions, to edit them, to analyze them, to map them to system implementations, and so on. Tools can

```
<xArch>
  ...

  <archTypes>
    <componentType id="tuner_type">
      <description>TV Tuner Type
        </description>
      <signature
        id="tuner_type.channel">
        <description>
          ChangeChannel Sig
        </description>
        <direction>in</direction>
        <type href="#changechannel_type"/>
      </signature>
      <variant>
        <guard>
          <booleanExp>
            <equals>
              <symbol>tv_standard</symbol>
              <value>NTSC</value>
            </equals>
          </booleanExp>
        </guard>
        <variantType
          href="#NTSC_tuner_type"/>
      </variant>
      <variant>
        <guard>
          <booleanExp>
            <equals>
              <symbol>tv_standard</symbol>
              <value>PAL</value>
            </equals>
          <booleanExp>
        </guard>
        <variantType href="#PAL_type"/>
      </variant>
    </componentType>
```




```
    <componentType id="NTSC_type">
      <description>
        NTSC Tuner Component Type
      </description>
      <signature id="NTSC_type.channel">
        <description>
          ChangeChannel Sig
        </description>
        <direction>in</direction>
        <type href="#changechannel_type"/>
      </signature>
    </componentType>

    <componentType id="PAL_type">
      <description>TV Tuner Type
        </description>
      <signature id="PAL_type.channel">
        <description>
          ChangeChannel Sig
        </description>
        <direction>in</direction>
        <type href="#changechannel_type"/>
      </signature>
    </componentType>
  </archTypes>
</xArch>
```

Fig. 9. Expansion of a formerly concrete type (tuner_type) into a variant type.

also help to insulate the architect from unpleasant syntactic details (such as XML mark up and namespaces).

We partition the tools available in our infrastructure into two groups: syntax-based and semantics-based tools. Syntax-based tools are generic, and their interfaces (graphical, textual, or programmatic) are based on the syntax of the ADL as specified in XML schemas. Because ADLs created in our infrastructure are extensible and easy to change, syntax-based tools are important since they adapt themselves as the language evolves and changes. Semantic-based tools make some assumptions about the meanings of particular constructs in the constructed ADLs and may require or support the use of specific schemas. These tools provide additional value for users of our infrastructure who adopt some or all of the xADL 2.0 schemas and for whom the semantic assumptions are valid. The relationships among the tools in our infrastructure are depicted in Figure 10. Each tool in our infrastructure is described in this section.

5.1 Syntax-Based Tools

One of the advantages of using XML is its support of off-the-shelf syntax- based tools for manipulating documents and schemas. Additionally, we have leveraged

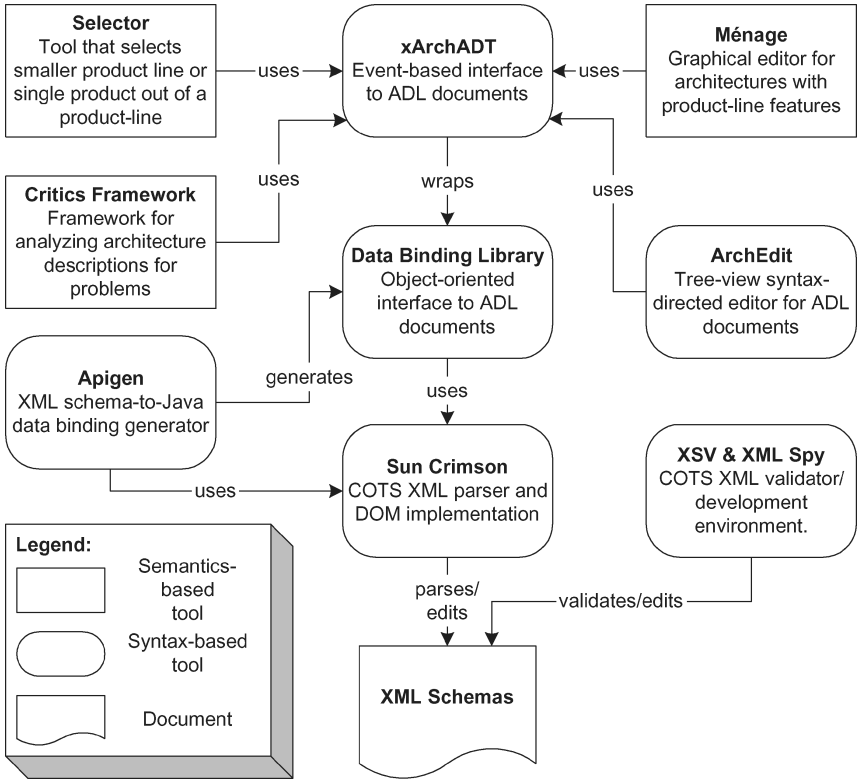


Fig. 10. Relationships among tools in our infrastructure.

these off-the-shelf tools to create additional syntax-based tools specific to our infrastructure.

5.1.1 *Off-the-Shelf XML Tools: Sun's Crimson Parser, XSV, and XML Spy.* Because ADLs created in our infrastructure are defined in XML schemas, and because architecture descriptions are XML documents that conform to these schemas, many off-the-shelf tools for working with XML in general are available to users. Three such tools are particularly important as they form the basis for all our other tools.

Sun's Crimson parser [Apache Group 2003] is a Java implementation of the DOM [Le Hors et al. 2003] and SAX [SAX Project 2003] APIs for parsing and manipulating XML documents. Crimson provides the ability to programmatically access the structures in XML documents (both architecture descriptions and schemas) at the level of XML syntactic constructs (elements, tags, attributes, comments, etc.) This API forms the foundation for more powerful syntax-based tools as will be described later in this section.

XSV [Thompson and Tobin 2003] is an open-source XML schema and instance document validator. It provides two main functions: to verify the syntactic correctness of XML schemas that are part of an ADL and to verify that an architecture description conforms to a set of schemas. While XSV cannot

verify complex architectural properties such as type consistency, it can provide some rudimentary syntactic checking of consistency at the level of XML constructs.

Altova's XML Spy [Altova GmbH 2003] is a commercial development environment for XML. XML Spy can be used to design and validate new schemas. Most importantly, it provides a way of exploring and designing schema constructs graphically which is a valuable tool for schema designers to inspect and communicate the modeling features in their schemas without having to read the raw schemas.

The reuse of off-the-shelf XML tools is a subtle but important benefit to our approach. While these tools do not work at the level of architectural constructs (e.g., components and connectors), they do allow us to avoid mundane tasks such as the creation of parsers, serializers, syntax checkers, graphical schema editors, and so on. Our approach also benefits automatically from improvements in these technologies: performance improvements in the underlying DOM implementation translate into faster applications.

5.1.2 *Data Binding Library.* Off-the-shelf XML tools tend to support manipulation of documents in terms of low-level XML concepts like elements, attributes, and text segments. Building tools that work with these low-level constructs directly is cumbersome and error prone as it requires the tools themselves to manage elements like namespaces directly and to ensure that documents conform to XML schemas.

When schemas are available, a more friendly programmatic interface to XML documents can be created based on the language prescribed by the schemas. In our infrastructure, this interface is provided through a data binding library [Sun Microsystems 2003a]. Our data binding library provides a set of Java classes corresponding to elements and attributes specified in the xADL 2.0 schemas. These classes hide XML details such as namespaces, header tags, sequence ordering, and so forth. Whereas a generic XML API like DOM exposes functions like `addElement(...)` and `getChildElements(...)`, classes in our data binding library expose functions like `addComponentInstance(...)` and `getAllInterfaces(...)`. Internally, the library uses the DOM implementation provided with Crimson to manipulate the underlying XML document. The Data Binding Library is automatically generated by another tool in our infrastructure, Apigen, described in the next section.

Consider the following XML definition of a component, excerpted from the xADL 2.0 STRUCTURE & TYPES Schema:

```
<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();
```

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 Crimson. Furthermore, it reduces the number of lines of code necessary to manipulate an XML-based architecture description significantly, by a ratio of approximately 5:1. That is, each call to the data binding library (e.g., `addInterface`) encapsulates about 5 lines of DOM code.

5.1.3 *Apigen.* If the data binding library must be rewritten every time a schema is added to, changed, or removed from an ADL, then the benefit of having it is negated. Fortunately, the syntax information present in the ADL schemas is enough to generate the data binding library automatically. We built a tool called "Apigen" (short for API generator) [Dashofy 2001] that can automatically generate the Java data binding library, described previously, for xADL 2.0 and extension schemas.[2] When an ADL's schemas are changed, tool builders simply rerun Apigen over the modified set of schemas to generate a new data binding library. Of course, bindings for elements that did not change will remain the same, minimizing the impact on existing tools that use the library. Because of the complexity of the XML schema language, Apigen is not a generic data binding generator it does not support the full gamut of constructs available in XML schemas (to our knowledge, no XML schema binding

[2]When we built Apigen, no adequate data binding generator existed that supported XML scemas. Several promising alternatives have since emerged' (e.g., Sun Microsystems' JAXB [Sun Microsystems 2003a] and XML Spy 5's proprietary generator [Altova GmbH 2003]).

Data Binding Library Call:

```
IComponent component;
IInterface interface;
...
component.addInterface(interface);
```

xArchADT Message:

```
ObjRef componentRef;
ObjRef interfaceRef;

message{
  name = "add";
  source = componentRef;
  type = "interface";
  target = interfaceRef;
}
```

Fig. 11. Example of a data binding library call and an equivalent xArchADT message.

generator currently does). 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. A comprehensive list of supported and nonsupported constructs in Apigen is beyond the scope of this article but is included with the Apigen tool's documentation.

5.1.4 *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 maintain 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. An example of how a data binding library call is expressed as an event data structure is shown in Figure 11. It uses first-class indirect object references rather than direct pointers to refer to elements in xADL 2.0 documents. That is, xArchADT assigns identifiers to xADL 2.0 elements, and those identifiers are used to refer to the elements. 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.

xArchADT, like Apigen and the data binding library, is reflective. It uses Java's built-in 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. xArchADT's biggest contribution to our tool suite, however, is that it increases the range of contexts in which the data binding library can be used. The library alone is well-suited for use in tightly-coupled object-oriented system development, but the xArchADT wrapper gives it an interface suitable for remote access across process boundaries (facilitated by middleware) and inclusion in an event-based environment.

5.1.5 *ArchEdit.* The data binding library and xArchADT expose different programmatic interfaces for manipulating architecture descriptions. Our

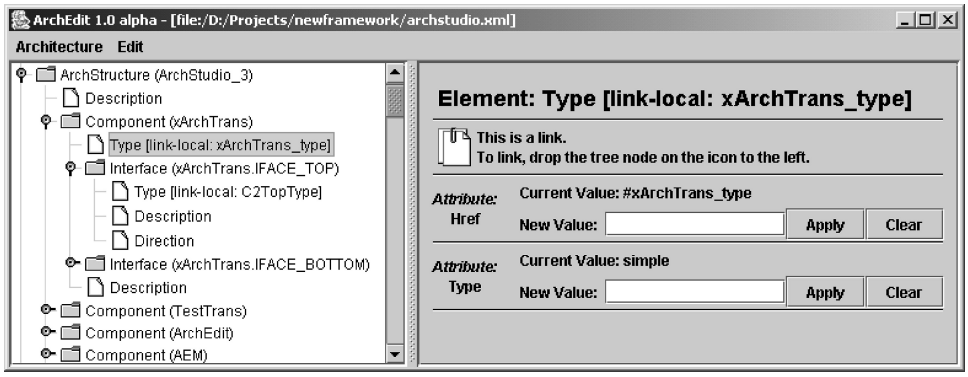


Fig. 12. ArchEdit Screenshot.

infrastructure also includes a syntax-driven tool with a graphical user interface called "ArchEdit". A screenshot of this tool is shown in Figure 12.

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 identical. This gives architects direct access to architecture descriptions without abstracting away details. 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.

ArchEdit is another reflective tool in our infrastructure. ArchEdit is syntax-driven: it does not understand the semantics of the displayed elements. 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.

There are two disadvantages to ArchEdit's reflectiveness. First, it does not enforce stylistic constraints or other rules on the architecture description that cannot be specified in XML. Second, ArchEdit cannot display the structure of the software architecture in an intuitive way—as a box-and-arrow diagram, for instance. These disadvantages are inherent in any syntax-based tool, but the benefits of having a free editor for new ADL features outweighs these disadvantages. More semantically-aware editors can be built to augment ArchEdit and integrated using xArchADT as we will show in the next section.

5.2 Semantics-Based Tools

Semantics-based tools differ from syntax-based tools in that they carry with them some notion of the semantics of the underlying notation. This necessarily

means that the use of certain semantics-based tools will be predicated on the inclusion of certain specific XML schemas in their target ADLs, as well as some assumptions about semantics associated with those schemas. Unlike syntax-based tools, semantics-based tools cannot necessarily update their interfaces and behavior automatically to support new schemas. This makes extensibility a paramount concern when developing new semantics-based tools. We have, therefore, endeavored to make our existing tools as tolerant of new schemas as possible, for example, by ignoring unknown information rather than failing.

In this section, we will profile three mature semantic tools that we developed for our infrastructure: the critics framework, used for developing and integrating analysis components; Ménage, a graphical editor for architecture descriptions; and the Selector, a tool for sculpting product lines down to smaller product lines or individual products. Additionally, we will provide a brief overview at the end of the section of current, in-progress projects we are undertaking to build more semantic tools.

5.2.1 *Critics Framework.* Consistency checking and analysis tools are important parts of any ADL's tool suite. In fact, many other ADLs were constructed specifically for the purpose of experimenting with new analysis techniques. In an extensible ADL, a single analysis tool is impractical. Having a modular critic framework [Robbins and Redmiles 1998] is a preferable alternative. Critics are software components that check properties of architecture descriptions, identify faults and potential flaws, and report them to other components. Critics differ from traditional analysis tools because they can update their analysis as the document changes, allowing continual revalidation to occur. The architecture of the critics framework, expressed in the C2 architectural style [Taylor et al. 1996], is shown in Figure 13. In the C2 style, components can make requests of components above them, but are not allowed to make assumptions about the components below them. As such, events requesting service are emitted upwards, and state changes are emitted downwards. For example, whenever the CriticADT's internal data store of open issues changes, it emits a notification to all components below it.

Design critics are independent components that can be included in the framework and activated if their analysis is relevant to the underlying ADL. Critics in our framework either perform fine-grained analysis, checking only a single property or a set of related properties, or build upon other critics to perform more complex analyses. For example, the *Link Critic* checks that all link endpoints are anchored on interfaces, while the *Architecture Evolution Manager* critic uses the *Link Critic* (and others) to decide whether the architecture description is complete enough to instantiate the specified system.

The CriticManager allows users to choose which critics to activate and deactivate, and several artist and GUI components render open issues for examination by end users. Users can use GUI elements (in our current framework, a "Focus Open Editors" button) to notify open architecture editors to focus on the particular element(s) that are involved in an issue. Screenshots of sample

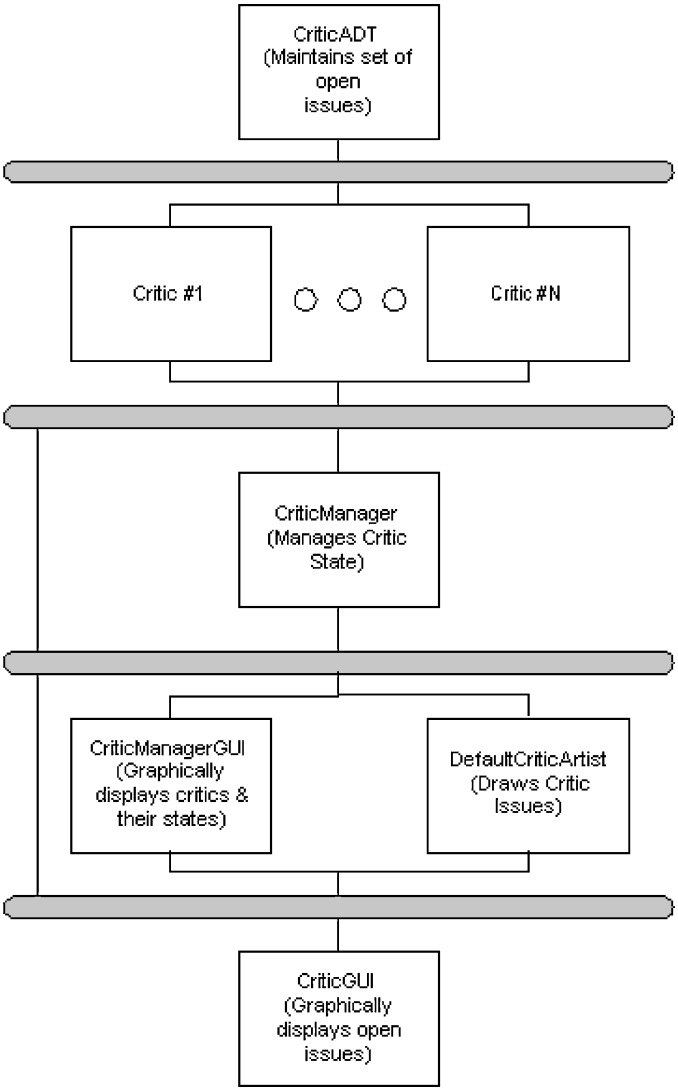


Fig. 13. Architecture of critics framework.

outputs from these critics and their visualization in the critic manager are shown in Figure 14.

Our framework includes a growing set of critics for basic consistency checking of xADL 2.0 documents. Current critics in our framework are listed in Table II. The particular set of critics we have constructed to date check some of the most common errors that can occur when defining a xADL 2.0 architecture. While our focus thus far has been on the creation of the framework, we intend to use this as a jumping-off point for future investigation of techniques for building and composing critics, for example, leveraging off-the-shelf XML-based analysis tools like xlinkit [Nentwich et al. 2002].

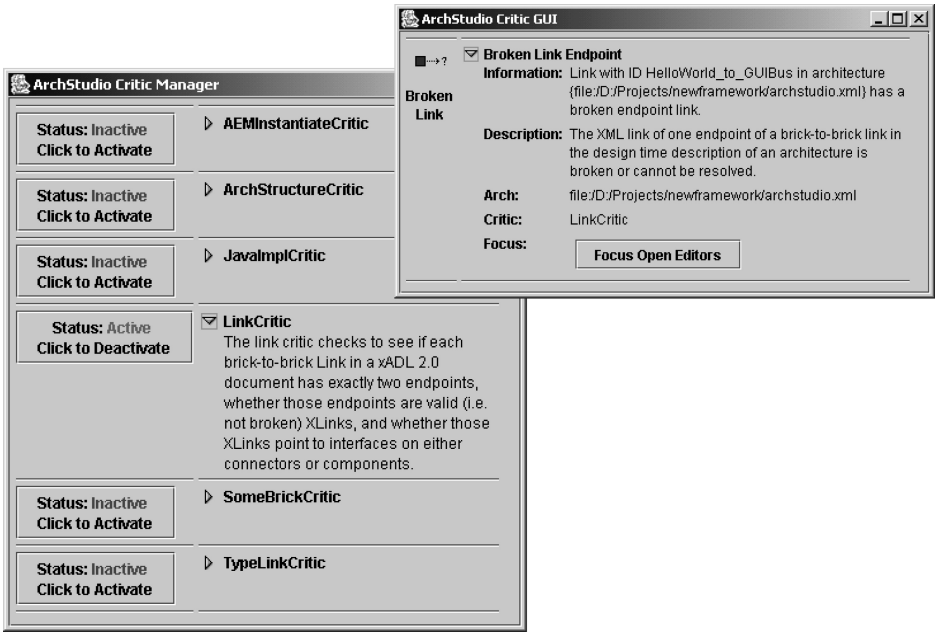


Fig. 14. Screenshots of critic manager and critic issues.

5.2.2 *Ménage.* Ménage [Garg et al. 2003] is a graphical design environment for single-product and product-line architectures. Ménage requires its target ADL to include the STRUCTURE & TYPES, VERSIONS, OPTIONS, VARIANTS, and BOOLEAN GUARD schemas from xADL 2.0. Ménage allows architectures to be visualized and expressed in a graph structure. Optional and variant constructs are depicted distinctively. Optional constructs have a dotted border. Variant components and connectors are tagged as variants; double-clicking on a variant element shows the set of possible variants. Each variant or optional element is associated with an editable Boolean guard condition to determine whether it is included in the architecture or not. A screenshot of Ménage is shown in Figure 15.

Ménage works well in combination with ArchEdit and our other syntax-directed tools. Architecture descriptions created in Ménage can be finetuned in ArchEdit if necessary, or ArchEdit can be used to access schema elements not supported directly by Ménage.

5.2.3 *Architecture Selector.* A product-line architecture represents a set of possible product architectures. As design parameters become fixed, the number of points of variation in a product-line architecture are reduced. When all points of variation have been removed, the result is a single product. Our infrastructure includes a tool called the "Selector" that allows the reduction of a product line to be done automatically for product-line architecture descriptions that use the STRUCTURE & TYPES, OPTIONS, VARIANTS, and BOOLEAN GUARD schemas.

Recall that each optional or variant element is accompanied by a Boolean guard expression that defines when it is included in the architecture. This

Table II. Critics and Their Descriptions

Critic	Description
ArchStructure Critic	Checks validity of ArchStructure elements.
Java Implementation Critic	Checks validity of Java implementation annotations on component, connector, and interface types.
Link Critic	Checks validity of architectural links—that they have two valid endpoints, etc.
Type Link Critic	Checks to ensure that each component, connector, and interface has a valid link to a type.
XML Link Critic	Checks that XML links found in architecture descriptions are valid.
Version Graph Critic	Checks that all component, connector, and interface types have valid links to a node in an appropriate version graph.
Signature Critic	Checks that components and connectors have the correct number and types of interfaces as specified by their types' signatures.
Guard Critic	Checks that all optional and variant elements are accompanied by a guard.
Signature-Interface Mapping Critic	Checks that components and connectors with sub-architectures are properly connected to internal components and connectors.
Interface Direction Critic	Checks that the directions of connected interfaces are compatible ('in' interfaces connect only to 'out' interfaces, etc.)
Variant Type Critic	Checks that all possible variants are consistent with their variant type.
Floating Elements Critic	Checks for components and connectors not connected to the rest of the architecture.
Duplicates Critic	Checks for elements with duplicate identifiers.
Architecture Evolution Manager Critic	Determines whether the architecture description contains enough information for it to be instantiated.

Boolean expression includes variables, just as Boolean expressions do in a programming language, but these variables are not bound to values in the architecture description. The Selector tool presents users with a graphical user interface in which they can specify a set of values that will be bound to variables in the Boolean expressions in the product-line architecture. Then, the user clicks a button to start the selection process. The selector evaluates all Boolean guards for optional and variant elements in the architecture using variable values provided by the user. For expressions that can be resolved, optional components are either made permanent (included) or removed (excluded) and chosen variants are made permanent. For expressions that cannot be fully resolved (due to an unbound variable, for example), the expression is reduced as far as possible, and the element remains optional or variant. When all options and variants have been resolved, the result is a single product architecture. If some options and variants remain, the result is a subset of the original product line.

5.2.4 *Additional Projects.* In addition to those listed in this section, we are constantly developing new semantic tools to augment our framework and

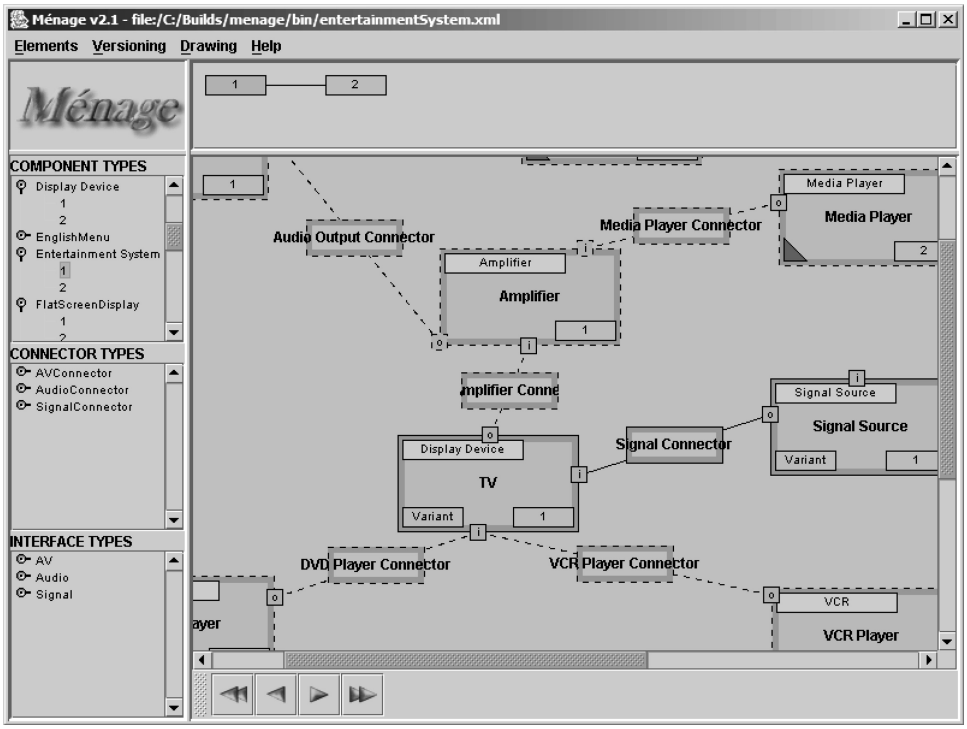


Fig. 15. Ménage screenshot.

provide more comprehensive support for the xADL 2.0 schemas. These projects are in various states of development, and more information on all of them can be found on our websites (see Section 10). A partial list follows:

Architecture Differencing and Merging. Tools to automatically generate documents describing the difference between two architectures, and merge the difference documents into existing architectures [Westhuizen and Hoek 2002]. This is useful for evolution management and feature propagation.

Product-Line Differencing and Merging. The application of differencing and merging techniques to product-line members, allowing differencing of individual members of a product line and propagation of features from one member product to another.

Eclipse Integration. An effort to provide and maintain mappings from xADL 2.0 components to their source-code implementations in the Eclipse [Eclipse Foundation 2004] Java development environment.

Archipelago. A second-generation graphical xADL 2.0 editor focusing on supporting exploratory design and new user-interface metaphors for architecture design. Archipelago is built on a highly-flexible plug-in framework that makes it a good fit with an extensible language infrastructure.

Type Wrangler. A tool to assist users in making sure components and connectors are matched properly to their types (e.g. interfaces match signatures, interface types are consistent, and so on).

6. EXPERIENCES

Our infrastructure has been applied to problems in a number of domains. In this section, we highlight four 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 and Mae, two representations used to model product-line architectures, demonstrating the extensibility of the XML schemas and tools in our infrastructure. Finally, we show how the infrastructure has been used to develop its own software development environment—demonstrating its ability to describe and support itself. Two of these examples demonstrate how the xADL 2.0 schemas can already support architecture-based development on their own, and two of them describe experiences where a new ADL was created in our infrastructure, leveraging our base schemas and tools. Overall, these 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.

6.1 AWACS

The U.S. Airborne Warning and Control System (AWACS) aircraft [Air Combat Command Public Affairs Office 2000] is supported by a large, distributed message-passing software architecture [Milligan 2000]. However, no formal architectural model of this software exists. To remedy this, and to help evaluate the scalability of our infrastructure, we modeled the architecture of the AWACS aircraft's software systems in xADL 2.0 and used this model as the basis for an architecture-based simulation of AWACS. The model of AWACS was created based on available documentation and a proprietary simulator of AWACS component behavior provided to us by an industrial partner.

The AWACS description consists of more than 10,000 lines (almost one megabyte) of XML. The AWACS description describes 125 components and 206 connectors, distributed across 28 processors, along with component, connector, and interface types. The description was validated against the xADL 2.0 schemas using XSV and visualized with ArchEdit. We used ArchEdit to inspect the architecture and make further improvements until the model was accurate and ran critics against the architecture to verify various aspects of it.

The various tools we used were relatively efficient given AWACS' large size. XSV can analyze an XML document of AWACS's size in a few seconds. ArchEdit amortizes the amount of time it takes to visualize an architecture by only reading data from the DOM tree when it is viewed; expanding the most populous node in the AWACS description for the first time takes about 5–8 seconds on a state-of-the-art PC. The performance of critics largely depended on how much data the critics were required to analyze. Critics that check a large portion of

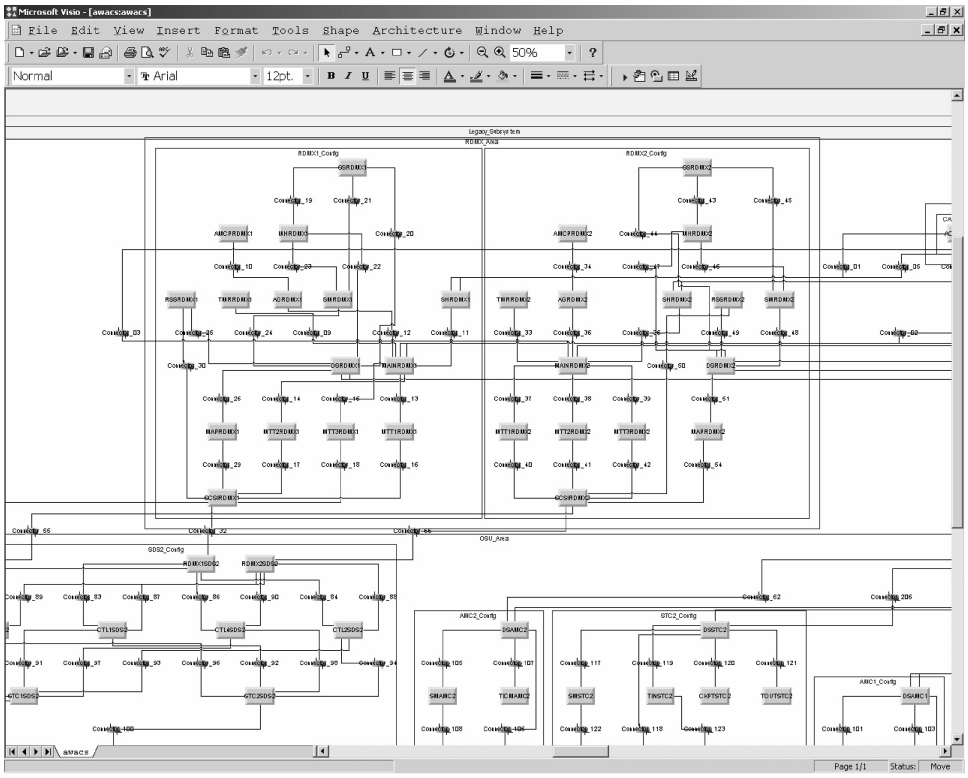


Fig. 16. AWACS simulator screenshot.

the architecture (for example, the Link Critic) took about 2 seconds to check AWACS on the PC. We believe that these times are reasonable given the size of the architecture being checked.

Along with the initial architecture description, we developed an architecture-based simulation of the communication among the components on the aircraft. The simulator consists of implementations of each component and connector type in the architecture in Java. Implementation mappings between types in the architecture description and these Java classes were added to the architecture description. A bootstrapping program reads the architecture description into xArchADT and instantiates and links the components and connectors automatically. To visualize the simulation, a separate project created an extension to Microsoft Visio that can render a xADL 2.0 architecture as a graph of components, connectors, and links [Ren and Taylor 2003]. Messages sent among AWACS components and connectors are animated on this graph. A screenshot of this simulator is shown in Figure 16.

Beyond scalability, our experience with AWACS demonstrates several additional positive characteristics 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 COTS and user-interface based tools in our infrastructure as we were able to use XSV to validate our description and ArchEdit to refine it.

6.2 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 NASA's Jet Propulsion Laboratory (JPL). JPL develops software systems for space probes and ground systems. This domain induces unique modeling needs; specifically, these architectures are modeled as state-based systems. To accommodate these modeling constraints, JPL's Mission Data Systems group [Rouquette and Reinholtz 2002] and their research partners at the University of Southern California (USC) created extensions to xADL 2.0 [Roshandel et al. 2004]. While the specific contents of these schemas cannot be shared due to their proprietary and confidential nature, they are roughly characterized as follows:

Static Behavior Schema. This schema extends the core xADL 2.0 schemas to capture static behavioral properties of the system. These behaviors add preconditions, postconditions, and invariants to xADL 2.0 component types, using a set of variables also defined in the extension. Additionally, xADL 2.0 signatures are extended with input and output parameters, allowing them to specify programming-language-like functions.

MDS Types Schema. This schema extends the static behavior schema to capture namespaces and complex inheritance information for components.

MDS Implementation Schema. This schema links architectural artifacts to implementation-level counterparts as expressed in the JPL-proprietary MDS framework. It is used primarily for code generation.

JPL integrated their extended version of xADL 2.0 into their development process by creating translators to and from existing notations such as UML and proprietary text-based notations. They also adopted our tools to visualize and manipulate architecture descriptions. Additionally, researchers at USC used the extended version of xADL 2.0 to model the architecture of the SCrover, a mobile robot based on JPL's MDS framework. They modified an existing architecture analysis tool, DRADEL [Medvidovic et al. 1999], to use JPL-extended xADL descriptions rather than its own proprietary models and used DRADEL to analyze the SCrover architecture. The DRADEL analysis found 21 errors in the SCrover architecture, 6 of which were not also detected by a peer review process. The ultimate aim of these efforts is to use xADL 2.0 as part of a large-scale effort to foster model-driven code generation. Additionally, the success of this effort recently prompted JPL to enter into a second project leveraging our infrastructure, this time focusing on space mission modeling and systems architecture.

This experience verifies that our infrastructure's genericity and its extensibility mechanisms are useful in previously unexplored domains, especially with regard to the xADL 2.0 base schemas. JPL was able to reuse the xADL 2.0

base schemas, creating relatively 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, nonpreplanned situations.

6.3 Mappings to Koala and Mae

To demonstrate our infrastructure's ability to capture concepts from an emerging research area and add tool support for those concepts efficiently, we created schemas that add the unique modeling constructs of Koala and Mae to our base xADL 2.0 schemas. Koala [Ommering et al. 2000] and Mae [Hoek et al. 2001] are two representation formats for capturing product-line architectures. As described earlier in Section 4.4, 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 by extending the definition of a component type with a new schema.

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 ADL 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 schema shown in Figure 17 used to add Koala-style diversity interfaces to xADL 2.0. 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 [Dashofy and Hoek 2001], 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

```
<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>
```

Fig. 17. xADL 2.0 extension schema for Koala-style diversity interfaces.

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.

6.4 ArchStudio 3

Finally, we have used our infrastructure to support its own development and evolution. As discussed above, our infrastructure includes a number of interacting tools that manipulate, analyze, and otherwise work with architecture descriptions. Most of these tools rely on a basic set of services for interacting with architecture descriptions (parse, read, modify, save) that are provided by the xArchADT component. Some tools rely on the services provided by other tools; for example, Ménage relies on design critics to perform consistency checking of architecture descriptions before it manipulates them. We have created

an architecture-based development environment called ArchStudio 3 that integrates most of our tools as components [Institute for Software Research]. Like the AWACS simulator, ArchStudio's architecture is described in xADL 2.0. ArchStudio's bootstrapper reads the xADL 2.0 description of ArchStudio and instantiates and connects all the components and connectors.

ArchStudio 3's toolset currently includes xArchADT, the critics framework (and all critics), ArchEdit, Ménage, Selector, and many other tools. It is comprised of approximately 80,000 lines of Java code, not including code generated by Apigen. Our experience in building, maintaining, and evolving ArchStudio 3 demonstrates that our infrastructure can support a nontrivial product's development and design and is capable of describing and supporting itself.

7. LESSONS LEARNED

In building and evolving our infrastructure, we learned several important lessons about modular architecture description language development, both in general and in the specific context of our XML-based infrastructure.

7.1 Guidance for Developing Modular ADL Features

As we have stated, our infrastructure does not attempt to automatically resolve the feature interaction problem [Zave 1999] in ADL development. That is, it does not provide tool support for ensuring semantic compatibility among ADL features. Nonetheless, in an infrastructure such as ours, feature interactions must be resolved in creating new ADL modules. We gained significant experience in solving feature interactions and preventing future difficulties when we developed the xADL 2.0 schemas and worked with our partners to extend the language further for specific domains. Our insights about developing new ADL features, some of which reflects conventional wisdom, are summarized here.

Separate core concepts from details. In developing ADL modules, concepts are more important than the specific implementation of those concepts. For example, our ABSTRACT IMPLEMENTATION schema captures the concept of an implementation mapping and indicates where such data should go but does not specify the details of what data should be included. Concrete schemas like the JAVA IMPLEMENTATION schema perform that function. Similarly, the STRUCTURE & TYPES schema defines the core architecture elements (components, connectors, interfaces, and links) but leaves out details to be specified in extensions.

Keep the dependency tree shallow. Every time a semantic dependency is introduced between schemas, it means that adopters of the dependent schemas must also implicitly adopt the parent schemas. Sometimes this is desirable: the xADL 2.0 type system provides a convenient place to specify variants. However, by maintaining compatibility without dependencies among the OPTIONS, VARIANTS, and VERSIONS schemas, we ease the adoption of these schemas individually.

Build domain-specific features atop generic ones. Each additional schema must be carefully compared semantically with all other schemas in the target ADL to determine compatibility. To maximize reusability of schemas, it is better to build and incorporate domain-generic schemas (schemas whose features are useful across many domains) first, and then build domain-specific schemas.

For example, we built features like product-line support and implementation mappings into xADL 2.0 first. Because the domain-generic schemas are not dependent on domain-specific ones, they can be more easily ported to other projects or ADLs because they will not require the porting or adoption of more specific schemas.

Eliminate redundancy where possible. Data that must be specified or copied in multiple places puts an undue burden on the tools that support the ADL to maintain consistency between the copies. Eliminating or minimizing redundancy can reduce this burden. In the xADL 2.0 schemas, types are provided as a way to keep common data about similar components, connectors, and interfaces in a single place.

Use first-class data elements instead of decorating existing elements. While it is natural to extend the definition of existing elements with additional data, it is often better in terms of extensibility to model the data as a set of first-class entities and provide links from the existing elements to the data. For example, rather than including version data and links among related versions inside component and connector types, we instead created first-class version trees and linked types to nodes in the tree. This allows for partial specifications (indicating that a version exists without necessarily having a fully-defined type to represent it). This also helps to disentangle version data and information from other elements that *do* decorate types such as information about variants. Finally, using first class elements allows many entities to link to the same information, helping to minimize redundancy.

Beyond these general guidelines, it is difficult to offer insight into the best way to integrate any given new feature. In the end, only experience can bear out whether a particular feature was integrated well or poorly. For example, we believe that our modularization of xADL 2.0 is a relatively good and manageable separation of concerns. We cannot credibly assert that our breakdown of features is best, or that there is no refactoring of the schemas that would lead to a more elegant, extensible, or modular notation. Nonetheless, our experiences using the notation increase our confidence that our modularization is an effective one.

Our infrastructure offers some means to alleviate this difficulty. First, it encourages experimentation with new refactorings and changes to existing notations more easily than a monolithic language would. So, while it cannot guide users in how to best integrate a new feature, it can at least help them to experiment with different alternatives. Second, it encourages the dissemination and reuse of existing feature integrations as we have done with xADL 2.0. xADL 2.0 carries with it the engineering knowledge and experiences we (and others) have had in integrating many useful features; the ability to reuse this particular well-worn modularization of features is far preferable to reinventing it.

7.2 The Usefulness of Syntax-Based and Other Reflective Tools

In the continual development of an extensible language, or any language that changes over time, the importance of reflective and syntax-based tools cannot be underestimated. Experience has shown that, in the absence of a reasonable

method to extend a language and its tool support, extensions will usually appear in the form of hacks [Luckham et al. 1987; Sun Microsystems 2003b]. For programming languages, extensions usually come in the form of specially-formatted comments, misusing this feature of the language simply because it is ignored by existing language tools like compilers and environments. For other design notations such as UML [Booch et al. 1998], existing concepts are overloaded to support new features to avoid extending the language definition and tools. Sometimes, when new features are too different from an existing language to be integrated, users will attempt to use multiple notations in tandem, or worse, develop an entirely new, competing notation. Reflective tools allow language elements to be extended and new first-class elements to be added to a language without resorting to hacks or excessive overloading.

Our experiences with our infrastructure, described in Section 6, have demonstrated that the availability of syntax-based and reflective tools like Apigen, xArchADT, and ArchEdit help make extending an existing notation and its tools tolerable. They provide the ability to experiment with new modeling features before taking on an extended investment in building and adapting semantics-based tools. They encourage users to extend the notation in a rational, principled way instead of hacking it. When extensions are eventually supported by more powerful, semantics-based tools, the syntax-based tools can assume a secondary role and augment the semantics-based tools.

7.3 XML Schemas as a Basis for Extensible Language Development

We have been relatively pleased with XML schemas, as they provide an expressive but not overly-complex metalanguage supported by a significant set of off-the-shelf tools that freed us from having to write parsers, serializers, and syntax checkers. For us, the most problematic restriction of XML schemas is that they only allow single inheritance of data types for elements and attributes, that is, a subtype can have only a single supertype. Therefore, subtypes that mix the content model of several supertypes are not permitted in XML schema-based languages. Several alternative metalanguages exist that alleviate this difficulty such as XInterfaces [Nölle 2002] and RDF [Lassila and Swick 1999], but none has the support from available tools and standards committees of XML schemas. In our experience, the benefits of XML schemas outweigh the severity of the single-inheritance limitation.

To examine how single inheritance affects the development of modeling languages for software, consider a base type "Component" that describes a software component. Now, consider two independent extensions to Component. Feature 1 adds implementation information to the component's description to describe how it is implemented in a particular programming language. Feature 2 adds administrative data to the component's description to describe who is responsible for the component, and who has worked on it in the past. This situation is depicted in Figure 18. Semantically, neither feature depends on the other. Therefore, it should ideally be possible to model a plain component, a component with implementation details, a component with administrative details, and a component with both implementation and administrative details. In XML

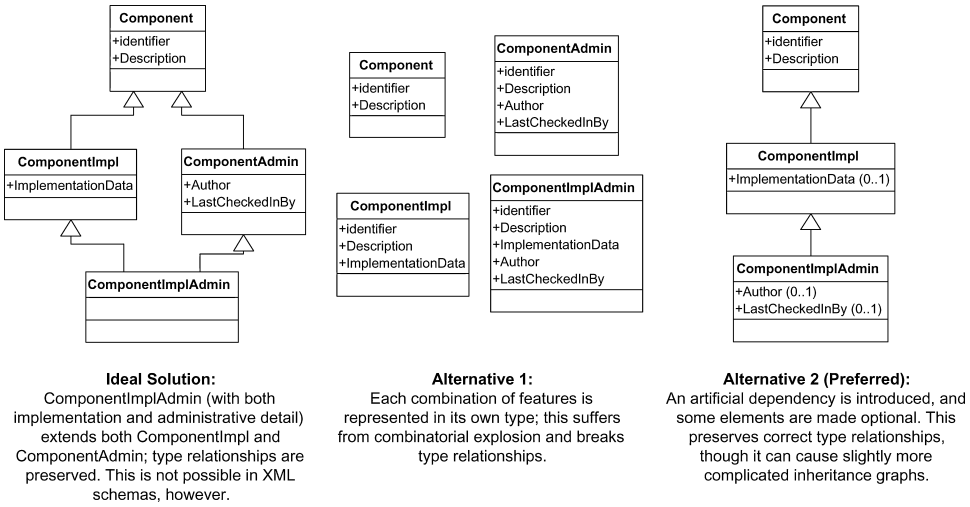


Fig. 18. Depiction of multiple inheritance issue in XML schemas and potential solutions.

schemas, there are several ways to accomplish this. First, it would be possible to create individual, independent types for each of the four variants of a component. This works, but it causes a combinatorial explosion of types as new, orthogonal features proliferate. Also, it breaks expected type relationships (a component-with-administrative-details is not a subtype of component).

A preferable solution is to introduce an artificial dependency between the two extensions. For example, component-with-administrative-and-implementation-details extends component-with-implementation-details which, in turn, extends component. To accommodate components that have implementation details only, administrative details are made optional in the extension. This preserves expected type relationships (a component-with-administrative-and-implementation-details is-a component) and is able to model all four kinds of components.

A final alternative is to use substitution groups and abstract base types for extensions. This is analogous to adding a list of `void*` to each object class in C++ or a vector of objects to the end of each object class in Java. This does not adequately preserve type relationships and weakens the ability to validate architecture documents via XML type checking.

Artificial dependencies are an imperfect solution but are workable and do not suffer from combinatorial explosion. If and when the W3C decides to add multiple-inheritance support to XML schemas in the future, schemas and instance documents that use artificial dependencies should be fairly easy to refactor.

8. RELATED WORK

The notion of an extensible architecture description language is not new, nor is our infrastructure the first to support XML-based architecture descriptions. However, our approach provides the first modular architecture description

language with useful generic features broken down into modules and accompanied by a process and tools that explicitly support new language extensions.

Our approach has been inspired by several other research areas. First, there is a large body of work on extensible and modular languages spanning three decades which continues today in projects like ArchJava, AspectJ, and LXWB. Second, many meta and generic modeling environments have emerged that technically could support many of the syntactic capabilities of our XML-based infrastructure. Third, previous ADL research has yielded XML-based ADLs such as ADML and xADL 1.0, two first-generation ADLs with limited extensibility. Finally, it is important to compare our approach with that of the Unified Modeling Language (UML) which has been touted as an architecture description language.

8.1 Extensible and Modular Language Research

The 1970s spawned significant research into extensible programming languages [Christensen and Shaw 1969]. These programming languages allowed additions to their syntax, usually through new BNF production rules, and to their semantics, through semantic modules at run-time.

Programming languages continue to be extended today. ArchJava [Aldrich et al. 2002] extends a programming language with architectural constructs and semantics such as components and connectors. AspectJ [Lopes et al. 1997] allows cross-cutting concerns to be specified for Java programs that are "woven" into existing Java classes. In these cases, the semantic capabilities provided by the tools could be achieved in the target programming language (Java) without language extensions. However, the amount of (largely manual) work to do so would be prohibitively difficult. ArchJava provides a compact way to specify and check constraints on Java programs that would be nearly impossible to check by hand for a system of any significant size. AspectJ adds significant value to Java by providing a single place where widely distributed concerns can be specified; previously, implementing such cross-cutting concerns would be done by copy-and-paste coding or ad hoc preprocessors. The difference between these tools and simple preprocessors is that tools like ArchJava and AspectJ add (and check) semantics beyond those available in the target language (here, Java) but conveniently output Java bytecode for compatibility with the existing run-time environment. More complex semantics (e.g., the addition of concurrency to a programming language that lacks it) may require more significant modification to the language and tools.

Other research has focused on modular domain-specific languages. The Language eXtension Work Bench (LXWB) [Peake et al. 2000] was developed in the mid-1990s to allow for modular language description using BNF-like production rules. The LXWB allows language modules to be defined and automatically generates parsers and other tools for compositions of modules much like our infrastructure. In fact, the extensibility aspects of the metalanguage supported by LXWB are very similar to those found in XML schemas. LXWB's metalanguage defines a type system over language elements and uses inheritance to extend existing constructs.

8.2 Meta and Generic Modeling Environments

As an alternative implementation route for our research contributions, we could have chosen any one of a number of alternative metamodeling languages and environments. Examples of these environments include Lisp [Steele 1990], DOME [Honeywell Inc. 1999] and GME [Ledeczi et al. 2000].

Lisp has been proposed as an alternative to using XML since XML's inception. In particular, the argument has been made that Lisp s-expressions are a better way of encoding hierarchical data than XML [Cunningham & Cunningham Inc. 2004] and that incorporation of additional Lisp elements could mean building semantics and semantic checks directly into the language itself.

DOME, the Domain Modeling Environment, is a tool for creating environments for model-based software engineering. DOME allows users to define visual grammars and then develop models which are instances of those grammars. The packaged version of DOME has special support for modeling in several popular modeling notations such as Coad-Yourdon Object Oriented Analysis and Data Flow diagrams. The ability for DOME users to create their own graphical grammars is roughly on par with editing the MOF specification for UML. Code can be added in DOME-specific languages (called Projector and Alter) to analyze models in user-defined grammars.

GME, the Generic Modeling Environment, is similar to DOME. Like DOME, users can edit a graphical metamodel and later create instances of that model. Like XML schemas, GME uses a types-and-instances paradigm: metamodels are types, and models are instances of those types. Both GME and DOME support notions of inheritance at the metamodel level, DOME supporting single inheritance of nodes and GME supporting virtual multiple inheritance.

All these environments are effectively viable alternatives to XML and XML schemas, offering different advantages and disadvantages, particularly in terms of the capabilities the off-the-shelf tools can provide such as vendor support, and so on. We do not assert that our specific choice and use of XML as the metamodeling environment for our infrastructure is a research contribution per se (though we did learn several valuable lessons about creating modular languages in XML such as those described in Section 7.3); rather, our research contributions come in the form of the particular decomposition of ADL features found in xADL 2.0, insights about how to create and compose ADL modules, and the roles of different kinds of tools in supporting modular ADL development and use.

8.3 Domain-Specific Software Architectures

Domain-specific ADLs and Domain Specific Software Architectures (DSSAs) [Coglianese 1992; Tracz and Coglianese 1992; Tracz 1996] are tangentially related to our infrastructure. Because of the amount of domain knowledge infused in a DSSA and its associated modeling language, a measure of compactness and reuse is possible that is not present in traditional, all-purpose ADLs. Because of the modularity of our infrastructure and the ability to make arbitrary extensions to ADLs, our infrastructure is ideal for constructing and experimenting

with domain-specific architecture description languages. Additionally, product-line architecture modeling techniques supported in our infrastructure provide intuitive ways of specifying points of variation in a reference architecture.

8.4 Other XML-Based ADLs

Early XML-based ADLs explored the possibility of encoding software architecture descriptions in XML but did not leverage its extensibility. Examples include xADL 1.0 [Khare 2001] and ADML [Spencer 2000]. xADL 1.0 is an intellectual predecessor of xADL 2.0 and included novel features for an ADL such as the ability to specify run-time change and detailed interface specifications. xADL 1.0's syntax was defined in a DTD and, therefore, it was extensible via XML namespaces, but its tools did not support extensibility. Moreover, its set of supported features was much smaller than xADL 2.0's.

ADML is basically an XML translation of the Acme core, also defined in a DTD. ADML improved marginally on Acme by introducing a notion of metaproperties and property types but also failed to leverage XML's extensibility. In fact, ADML still uses name-value pair properties as its extensibility mechanism with the names and values simply encoded in XML. Our approach is the first to leverage XML's extensibility mechanisms fully and does not require the creation or use of a proprietary metalanguage such as ADML's metaproperties. Beyond better extensibility, our existing support for features such as product-line architectures and implementation mappings exceeds ADML's capabilities.

8.5 Relationship to UML

The Unified Modeling Language [Booch et al. 1998] is an effort to create a standard, generic, graphical modeling language for software systems. There is some amount of disagreement on whether UML is an ADL or not [Booch et al. 1999; Robbins et al. 1997]. From our perspective, UML 1.x can be used as an ADL but it must be extended to some degree to model the core constructs of software architecture such as components and connectors. To address this deficiency, UML 2.0 added a new diagram type called a composite structure diagram. Composite structure diagrams include notions of parts, connectors, ports, and links which map onto xADL 2.0 components, connectors, interfaces, and links, respectively.

Our approach improves on the UML approach in two significant ways: features and extensibility. With respect to features, xADL 2.0's type system and product-line support represent abilities not present in UML, including UML 2.0.

More importantly, our infrastructure's extensibility mechanisms and ability to add features like these are far more flexible than UML's. UML has two levels of extensibility: built-in extensibility mechanisms (stereotypes, tagged values, and constraints) and MOF editing. UML's built-in mechanisms allow UML to be extended through overloading of existing constructs. That is, an existing construct like a UML class or part can be stereotyped to serve as a software architecture component or component type. This approach has been used in

several efforts to adapt UML into an ADL [Hofmeister et al. 1999; Robbins et al. 1997]. Overloading can become problematic if constructs are needed that do not naturally fit one of the existing UML diagram types. For example, consider a construct like xADL 2.0's version graphs. It would be possible to create a graph of nodes using an existing UML diagram type like a class diagram, but the intent of the class diagram is to show relationships between software artifacts in the design of a system rather than track the evolution of a component over time. An alternative to using the built-in extensibility mechanisms is to redefine UML in its own metalanguage, the metaobject framework (MOF). This allows arbitrary extensibility, but the resulting language is not UML, and, therefore, UML tools will generally not support this new language. In contrast, our approach provides extensibility analogous to editing at the MOF level but provides tools that expect and support this level of extensibility.

UML's relationship to XML is maintained through the XMI format, an XML-based interchange format in which UML diagrams can be saved and loaded. XMI eases interoperability among UML tools by providing them a common text-based *lingua franca* (UML's graphical format does not have a standard method of serialization), but does not contribute to UML's modeling capabilities or extensibility.

9. CONCLUSIONS AND FUTURE WORK

This article describes both research and technical contributions. Research contributions include the first decomposition of an architecture description language into modules, insights about how to develop new language modules and a process for integrating them, and insights about the roles of different kinds of tools in a modular ADL-based infrastructure. This is accompanied by the technical contribution of a viable infrastructure for creating and extending XML-based architecture description languages. The infrastructure dramatically reduces the amount of effort involved 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 provides critical tools like parsers, editors, syntax checkers, and data bindings for ADLs, allowing developers to spend more time on building high-value tools that focus on addressing open issues.

We believe that our infrastructure has fulfilled all the goals set forth in Section 3. Table III recalls these goals and describes the specific infrastructure mechanisms used to meet the goals.

Additionally, the experiences that we and others have had in applying our approach have indicated many positive qualitative aspects of the approach. 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, previously 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 [Ommering et al. 2000] and Mae [Hoek

Table III. Goals Achieved by Our Infrastructure

Goal	Infrastructure Characteristics
Place as few limits on what can be expressed at the architecture level as possible.	Our XML-based extension mechanism allows users to define arbitrary new constructs and extend any existing construct. Our experiences with JPL and the Koala/MAE extensions confirm the effectiveness of this mechanism.
Allow new modeling features to be added and existing features to be modified over time.	
Allow experimentation with new features and combinations of features.	New ADLs can be created simply by composing a set of schemas.
Provide tool-builders with support for creating and manipulating architecture models, even when the underlying notation can and will change.	Our syntax-based tools provide instant parsers, data bindings, and editors for new schemas automatically with no new code; our semantic tools and environments are componentized and relatively easy to extend, as confirmed by the integration of DRADEL into our environment.
Provide a library of generically useful modules applicable to a wide variety of domains.	The xADL 2.0 schemas are these modules. Some or all of these schemas were reused in each of our evaluation efforts.
Allow modeling features, once defined, to be reused in other projects.	Along with the reuse of the xADL 2.0 schemas, the MDS extension schemas developed by JPL have been used in both code-generation and architecture analysis efforts.

et al. 2001]. Our infrastructure supports its own development and evolution within the ArchStudio 3 design environment.

In the future, we plan to experiment with integrating more tools into our infrastructure via the ArchStudio 3 environment to add useful features. Some of these are described in Section 5.2.4. Beyond these, we believe the xlinkit tool [Nentwich et al. 2002] can potentially be used to express and check constraints on XML links in xADL 2.0 documents. Also, the SmartTools toolset [Attali et al. 2001] can potentially provide an XML-based language like xADL 2.0 with alternative editors and semantic analysis tools.

For our longterm research goals, 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 also want to investigate the application of our existing tools to new problems. For example, we believe that our product-line tools can be applied to support architectural trade-off specification and analysis, where product variants represent points in the space of available trade-offs instead of actual product descriptions.

10. ONLINE RESOURCES

For more information about xADL 2.0, please see http://www.isr.uci.edu/projects/xarchuci/.

For more information about the ArchStudio 3 suite of tools, please see http://www.isr.uci.edu/projects/archstudio/.

ACKNOWLEDGMENTS

The authors would first like to acknowledge and thank the anonymous reviewers for their invaluable insights about this article which greatly assisted us in shaping and scoping this work and its presentation. We would also like to thank Ping Chen, Tina Cheng, Matthew Critchlow, Rob Egelink, Justin Erenkrantz, Joachim "Joe" Feise, Akash Garg, John Georgas, Scott Hendrickson, Fei Hoffman, Jesse Hsia, Arick Ma, Roshni Malani, Kari Nies, Jie Ren, Jane Tran, and Christopher Van der Westhuizen for their valuable contributions to ArchStudio 3 and the other xADL 2.0 tools. Additionally, we would like to thank the researchers and practitioners who contributed significantly to our evaluation efforts, especially Will Tracz, Nicolas Rouquette, Vanessa Carson, Peter Shames, Neno Medvidovic and Roshanak Roshandel.

REFERENCES

APACHE GROUP. 2003. *Crimson*. Available at <http://xml.apache.org/crimson/>.

AIR COMBAT COMMAND PUBLIC AFFAIRS OFFICE. 2000. *Fact Sheet: E-3 Sentry (AWACS)*. U.S. Air Force (July). Available at <http://www.af.mil/news/factsheets/E_3_Sentry_AWACS_.html>.

ALDRICH, J., CHAMBERS, C., AND NOTKIN, D. 2002. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*. (Orlando, FL.) ACM, 187–197.

ALLEN, R. AND GARLAN, D. 1997. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Method. 6*, 3 (July), 213–249. Available at <http://doi.acm.org/10.1145/258077.258078>.

ALTHEIM, M., BOUMPHREY, F., DOOLEY, S., MCCARRON, S., SCHNITZENBAUMER, S., AND WUGOFSKI, T. 2001. Modularization of XHTML. World Wide Web Consortium, W3C Recommendation Report (April). Available at <http://www.w3.org/TR/ xhtml-modularization/>.

Altova GmbH. 2003. XML spy website. Available at <http://www.xmlspy.com/>.

ATTALI, I., COURBIS, C., DEGENNE, P., FAU, A., PARIGOT, D., AND PASQUIER, C. 2001. SmartTools: A generator of interactive environments tools. In *Proceedings of the the International Conference on Compiler Construction (CC'01)* (April). Genova, Italy.

BERNERS-LEE, T. AND CONNOLLY, D. 1998. Web architecture: Extensible languages. W3C Note Report (Feb.) 10. Available at <http://www.w3.org/TR/NOTE-webarch-extlang>.

BINNS, P., ENGLEHART, M., JACKSON, M., AND VESTAL, S. 1996. Domain-specific software architectures for guidance, navigation and control. *Int. J. Softw. Eng. Knowl. Eng. 6*, 2 (June), 201–227.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 1998. *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley, Reading, MA.

BOOCH, G., GARLAN, D., IYENGAR, S., KOBRYN, C., AND STAVRIDOU, V. 1999. *Is UML an architectural description language?* Available at <http://www.acm.org/sigplan/oopsla/oopsla99/2_ap/tech/2d1a_uml.html>. (*OOPSLA '99*).

BOSCH, J. 1999. Product-line architectures in industry: A case study. In *Proceedings of the 21st International Conference on Software Engineering*. IEEE Computer Society Press. Los Angeles, CA. 544–554.

BOSCH, J. 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Wesley, A. Ed. ACM Press.

BRAY, T., PAOLI, J., AND SPERBERG-MCQUEEN, C. M. 1998. Extensible markup language (XML): *Part I. Syntax*. World Wide Web Consortium, Recommendation Report (Feb.). Available at <http://www.w3.org/TR/1998/REC-xml>.

CHRISTENSEN, C. AND SHAW, C. 1969. *Proceedings of the Extensible Languages Symposium*. Boston, MA.

CLARK, J. AND DEROSE, S. 1999. XML path language (XPath) version 1.0. World Wide Web Consortium, W3C Recommendation Report REC-xpath-19991116 (Nov.). Available at <http://www.w3.org/TR/xpath>.

CLEMENTS, P. AND NORTHROP, L. 2001. *Software Product Lines—Practices and Patterns*. Pearson Education, (Addison-Wesley).

COGLIANESE, L., SMITH, R., AND TRACZ, W. 1992. DSSA case study: Navigation, guidance, and flight director design and development. In *Proceedings of the IEEE Symposium on Computer-Aided Control System Design*. (March), 102–109.

CUNNINGHAM & CUNNINGHAM, INS. 2004. Xml isa poor copy of ess expressions. Available at <http://c2.com/cgi/wiki?XmlIsaPoorCopyOfEssExpressions>.

DASHOFY, E. M. 2001. Issues in generating data bindings for an XML schema-based language. In *Proceedings of the Workshop on XML Technologies in Software Engineering (XSE'01)* (May). Toronto, Canada.

DASHOFY, E. M. AND HOEK, A. V. D. 2001. Representing product family architectures in an extensible architecture description language. In *Proceedings of the International Workshop on Product Family Engineering (PFE-4)* (Oct.), 330–341.

DASHOFY, E. M., HOEK, A. V. D., AND TAYLOR, R. N. 2001. A highly-extensible, XML-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)* (Aug.). Amsterdam, The Netherlands.

DEROSE, S., MALER, E., AND ORCHARD, D. 2001. XML linking language (XLink) version 1.0. World Wide Web Consortium, W3C Recommendation Report (June). Available at <http://www.w3.org/TR/xlink/>.

Eclipse Foundation. 2004. *Eclipse*. Available at <http://www.eclipse.org/>.

FALLSIDE, D. C. 2001. XML schema part 0: Primer. World Wide Web Consortium, W3C Recommendation Report (May). Available at <http://www.w3.org/TR/xmlschema-0/>.

GARG, A., CRITCHLOW, M., CHEN, P., VAN DER WESTHUIZEN, C., AND HOEK, A. V .D. 2003. An environment for managing evolving product line architectures. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'03)* (Sept.). Amsterdam, The Netherlands.

GARLAN, D., MONROE, R. T., AND WILE, D. 2000. ACME: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, Leavens, G.T. and Sitaraman, M. Eds. Cambridge University Press, 47–48.

GORLICK, M. M. AND RAZOUK, R. R. 1991. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering*. (May), 23–34.

HOARE, C. A. R. 1978. Communicating sequential processes. *Comm. ACM. 21*, 8 (Aug.), 666–677.

HOEK, A. V. D., MIKIC-RAKIC, M., ROSHANDEL, R., AND MEDVIDOVIC, N. 2001. Taming architectural evolution. In *Proceedings of the 6th European Software Engineering Conference (ESEC) and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)* (Sept.). Vienna, Austria. 10–14.

HOFMEISTER, C., NORD, R. L., AND SONI, D. 1999. Describing software architecture with UML. In *Proceedings of the 1st IFIP Working Conference on Software Architecture*. (Feb.). San Antonio, TX. Available at <http://citeseer.nj.nec.com/cache/papers/cs/15435/;http:zSzzSzwww.scr.siemens.comzSzpdfzSzUsingUML-unix.pdf/;hofmeister99describing.pdf>.

HONEYWELL INC. 1999. DOME Guide.

INSTITUTE FOR SOFTWARE RESEARCH. ArchStudio, an architecture-based development environment. University of California, Irvine. Available at <http://www.isr.uci.edu/projects/archstudio/>.

KHARE, R., GUNTERSDORFER, M., OREIZY, P., MEDVIDOVIC, N., AND TAYLOR, R. N. 2001. xADL: Enabling architecture-centric tool integration with XML. In *Proceedings of the 34th International Conference on System Sciences (HICSS-34), Software mini-track* (Jan.). Maui, Hawaii.

LASSILA, O. AND SWICK, R. 1999. *Resource Description Framework (RDF) Model and Syntax Specification*. World Wide Web Consortium, W3C Recommendation Report (Feb.). Available at <http://www.w3.org/TR/REC-rdf-syntax/>.

LE HORS, A., LE HÉGARET, P., WOOD, L., NICOL, G., ROBIE, J., CHAMPION, M., AND BYRNE, S. 2003. Document object model (DOM) level 3 core specification. World Wide Web Consortium, W3C Working Draft Report (June). Available at <http://www.w3.org/TR/2003/WD-DOM-Level-3-Core-20030609/>.

LEDECZI, A., MAROTI, M., BAKAY, A., KARSAI, G., GARRETT, J., THOMASON, C., NORDSTROM, G., SPRINKLE, J., AND VOLGYESI, P. 2000. The generic modeling environment. Vanderbilt University, 6. Tech. Rep. Available at <http://www.isis.vanderbilt.edu/Projects/gme/GME2000Overview.pdf>.

LOPES, C. V., KICZALES, G., MENDHEKAR, A., MAEDA, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*. Finland.

LUCKHAM, D. C., HENKE, F. W. V., KRIEG-BRÜCKNER, B., AND OWE, O. 1987. *Anna—A Language for Annotating Ada Programs*. Springer-Verlag, 260.

LUCKHAM, D. C., KENNEY, J. J., AUGUSTIN, L. M., VERA, J., BRYAN, D., AND MANN, W. 1995. Specification and analysis of system architecture using rapide. *IEEE Trans. Softw. Eng. 21*, 4 (April), 336–355. Available at <http://citeseer.nj.nec.com/luckham95specification.html>.

MAGEE, J., DULAY, N., EISENBACH, S., AND KRAMER, J. 1995. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*. Springer-Verlag, Berlin. 137–153. Available at <http://citeseer.nj.nec.com/rd/0ftp:zSzzSzdse.doc.ic.ac.ukzSzdse-paperszSzdarwinzSzesec.pdf/magee94specifying.pdf>.

MEDVIDOVIC, N., ROSENBLUM, D. S., AND TAYLOR, R. N. 1999. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)* (May). IEEE Computer Society. Los Angeles, CA. 44–53. Available at <http:// www.ics.uci.edu/~dsr/old-home-page/icse99-dradel.pdf>.

MEDVIDOVIC, N. AND TAYLOR, R. N. 2000. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng. 26*, 1 (Jan.), 70–93. *Reprinted in Rational Developer Network: Seminal Papers on Software Architecture*. Rational Software Corporation. Available at <http://www.rational.net/>.

MEHTA, N. R., MEDVIDOVIC, N., AND PHADKE, S. 2000. Towards a taxonomy of software connectors. In *Proceedings of the 2000 International Conference on Software Engineering* (June). Limerick, Ireland. 178–187. Available at <http://sunset.usc.edu/classes/cs599_2000/Conn-ICSE2000.pdf>.

MILLIGAN, M. K. J. 2000. Implementing COTS open systems technology on AWACS. *CrossTalk: J. Defense Softw. Eng.* (Sept.). Available at <http://www.stsc.hill.af.mil/crosstalk/2000/09/milligan.html>.

NENTWICH, C., CAPRA, L., EMMERICH, W., AND FINKELSTEIN, A. 2002. Xlinkit: A consistency checking and smart link generation service. *ACM Trans. Internet Tech. 2*, 2 (May), 151–185. Available at <http://www.systemwire.com/whitepapers/xlinkit.pdf>.

NÖLLE, O. 2002. XInterfaces: A new schema language for XML. BS Thesis. Institute for Computer Science, University of Freiburg. Available at <http://www.onoelle.de/xinterfaces/thesishtml/index.html>.

OBJECT MANAGEMENT GROUP. 2001. The Common Object Request Broker: Architecture and Specification.

OMMERING, R. V., LINDEN, F. V. D., KRAMER, J., AND MAGEE, J. 2000. The Koala component model for consumer electronics software. *IEEE Comput. 33*, 3 (March), 78–85.

OMMERING, R. V. 2002. Building product populations with software components. In *Proceedings of the 24th International Conference on Software Engineering*. 255–265.

OREIZY, P., GORLICK, M. M., TAYLOR, R. N., HEIMBIGNER, D., JOHNSON, G., MEDVIDOVIC, N., QUILICI, A., ROSENBLUM, D. S., AND WOLF, A. L. 1999. An architecture-based approach to self-adaptive software. *IEEE Intell. Syst. 14*, 3 (May-June), 54–62.

PEAKE, I. AND SALZMAN, E. 1997. Support for modular parsing in software reengineering. In *Proceedings of the Conference on Software Technology and Engineering Practice'97* (July) London, UK. 58–66.

PEAKE, I. 2000. LXWB User's Guide. Centre for Software Maintenance. Queensland.

PERRY, D. E. AND WOLF, A. L. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Soft. Eng. Notes. 17*, 4 (Oct.), 40–52. Available at <http://citeseer.nj.nec.com/perry92foundation.html>.

REN, J. AND TAYLOR, R. N. 2003. Visualizing software architecture with off-the-shelf components. In *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering* (July) San Francisco, CA. 132–141.

ROBBINS, J., REDMILES, D., AND ROSENBLUM, D. 1997. Integrating C2 with the unified modeling language. In *Proceedings of the California Software Symposium (CSS'97)* (Nov.). Irvine, CA. 11–18. Available at <ftp.ics.uci.edu/pub/eden/papers/conferences/1997/css/CSS97.pdf>.

ROBBINS, J. AND REDMILES, D. 1998. Software architecture critics in the Argo design environment. In *Proceedings of the International Conference on Intelligent User Interfaces (UIST'98)* (Jan.) San Francisco, CA. 47–60.

ROSHANDEL, R., SCHMERL, B., MEDVIDOVIC, N., GARLAN, D., AND ZHANG, D. 2004. Understanding tradeoffs among different architectural modeling approaches. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04)* (June) Oslo, Norway.

ROUQUETTE, N. AND REINHOLTZ, K. 2002. The mission data system's software architecture framework. *International Conference on Software Engineering (ICSE'02)*. Orlando, FL. Available at <http://www-scf.usc.edu/~csci577/teams/team12a/MDS/mds_sw_arch_framework.ppt>.

SAX PROJECT. 2003. *SAX: Simple API for XML*. Available at <http://www.saxproject.org/>.

SCHMERL, B. AND GARLAN, D. 2002. Exploiting architectural design knowledge to support self-repairing systems. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*. (July), 241–248. Ischia, Italy. Available at <http://portal.acm.org/citation.cfm?id=568804>.

SPENCER, J. 2000. Architecture description markup language (ADML): Creating an open market for IT architecture tools. The Open Group, White Paper Report (Sept.). Available at <http://www.opengroup.org/tech/architecture/adml/ background.htm>.

STEELE, G. 1990. *Common Lisp: the Language*. 2nd Ed., Digital Press, Woburn, MA.

SUN MICROSYSTEMS. 2003a. Java architecture for XML binding (JAXB). Available at <http://java.sun.com/xml/jaxb/>.

SUN MICROSYSTEMS. 2003. Javadoc tool home page. Available at <http://java.sun.com/j2se/javadoc/>.

SZYPERSKI, C. 1997. *Component Software: Beyond Object-Oriented Programming*. ACM Press, New York, NY.

TAYLOR, R. N., MEDVIDOVIC, N., ANDERSON, K. M. E., JAMES WHITEHEAD, J., ROBBINS, J. E., NIES, K. A., OREIZY, P., AND DUBROW, D. L. 1996. A component- and message-based architectural style for GUI software. *IEEE Trans. Softw. Eng. 22*, 6 (June), 390–406.

THOMPSON, H. S. AND TOBIN, R. 2003. Current status of XSV. Tech. Rep. University of Edinburgh (July). Available at <http://www.ltg.ed.ac.uk/~ht/xsv-status.html>.

TRACZ, W. AND COGLIANESE, L. 1992. A case for domain-specific software architectures. *In Proceedings of the WISR-5*.

TRACZ, W. AND COGLIANESE, L. 1993. An adaptable software architecture for integrated avionics. In *Proceedings of the IEEE National Aerospace and Electronics Conference*. (May), 1161–1168.

TRACZ, W. 1996. Domain-specific software architectures, frequently asked questions. Tech. Rep. Loral Federal Systems Company.

WESTHUIZEN, C. V. D. AND HOEK, A. V. D. 2002. Understanding and propagating architectural change. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 3)* (Aug.). Montreal, Canada.

ZAVE, P. 1999. FAQ sheet on feature interaction. Available at <http://www.research.att.com/~pamela/faq.html>, AT&T, HTML.

Received August 2003; revised July 2004; accepted October 2004