A Highly-Extensible, XML-Based Architecture Description Language

Eric M. Dashofy edashofy@ics.uci.edu
André van der Hoek andre@ics.uci.edu
Richard N. Taylor taylor@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.

ABSTRACT

Software architecture research focuses on models of software architectures as specified in architecture description languages (ADLs). As research progresses in specific areas of software architectures, more and more architectural information is created. Ideally, this information can be stored in the model. An extensible modeling language is crucial to experimenting with and building tools for novel modeling constructs that arise from evolving research. Traditional ADLs typically support a small set of modeling constructs very well, but adapt to others poorly. XML provides an ideal platform upon which to develop an extensible modeling language for software architectures. Previous XML-based ADLs successfully leveraged XML's large base of off-the-shelf tool support, but did not take advantage of its extensibility. To give software architecture researchers more freedom to explore new possibilities and modeling techniques while maximizing reuse of tools and modeling constructs, we have developed xADL 2.0, a highly extensible XML-based ADL. xADL 2.0 supports run-time and design time modeling, architecture configuration management and model-based system instantiation. Additionally, xADL 2.0 has a set of extensible infrastructure tools that support the creation, manipulation, and sharing of xADL 2.0 documents.

1. Introduction

One of the key goals of software architecture research is understanding and manipulating a system at a higher level of granularity than modules or lines-of code. Generally, software architectures are composed of *components*, the loci of computation, *connectors*, the loci of communication, and *configurations*, constraints on the arrangement and behavior of components and connectors [10,21]. The architecture of a software system is a model, or abstraction, of that system. Software architecture researchers need extensible, flexible architecture description languages (ADLs) and equally flexible and extensible tools to represent these models and experiment with new models or modeling techniques.

To these ends, we have developed xADL 2.0, a highly-extensible XML-based ADL. xADL 2.0 is based on xArch, a core representation for basic architectural elements that uses the XML schema extension method (see Section 2.2) for extending this core. xArch was jointly developed by the University of California, Irvine and Carnegie Mellon University to provide a basis for creating new ADLs and ADL features easily. xADL 2.0 was developed in parallel with xArch. In addition to the core xArch elements, xADL 2.0 provides support for architectural prescription (the "recipe" for how an architecture should be instantiated into a running system), a types-and-instances model, architecture-level configuration management concepts such as versions, options, and variants, and a mapping from types onto implementations of those types. We plan to further extend xADL 2.0 in the future as our research on software architecture progresses. While xADL 2.0 is not the first XML-based ADL, it offers some novel features.

First, xADL 2.0 is highly extensible. Since xADL 2.0 inherits XML's schema-based extensibility mechanism, users can effectively and independently extend it with features that support their particular needs. Specifically, users can write extensions that modify and add to the existing elements of xADL 2.0 in a modular fashion. Moreover, since xADL 2.0 is defined as a set of extensions to xArch, undesired or unused features of xADL 2.0 can be left out of architectural descriptions.

Second, xADL 2.0 is supported by tools that provide infrastructure for storing, manipulating, and sharing xADL 2.0 specifications. This set of tools completely insulates end-users from the peculiarities of XML and the internal structure of xADL 2.0 documents. The only users who need to be aware of the XML-nature of xADL 2.0 are those who extend it with their own schemas.

Third, xADL makes a clear distinction between architectural *prescription* (the design-time template that is used for instantiating the architecture) and architectural *description* (which describes the run-time state of the system). Whereas existing ADLs typically assume that their models are applicable both at design-time and run-time, xADL 2.0 logically separates the two. Extensions that make sense at design-time may not make sense at run-time and vice-versa.

Effort partially sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

Fourth, xADL 2.0 provides a general set of extensions that users may or may not adopt. For example, it incorporates support for architecture-level configuration management and mappings from components and connectors onto Java implementations. These features are generally not supported by architecture description languages.

Combined, all of these features provide a solid foundation for tool-building and research experimentation in software architectures. xADL 2.0 and its associated tools can be used to incrementally develop new or variant ADLs instead of "reinventing the wheel" by building a new ADL from scratch. In particular, xADL 2.0 can represent architectures in multiple architectural styles, provided appropriate extensions are made. This extensibility makes it seem as if xADL 2.0 is just a representation format. However, it satisfies all the requirements to be an ADL, as outlined in [18]. Therefore, xADL 2.0 can be considered both a representation format and an ADL; however, we will discuss it as an ADL for the remainder of this paper.

2. Background and Related Work

2.1. First-Generation ADLs

Architecture description languages such as Wright [1], Rapide [15] and Darwin [16] represent the first generation of ADLs. A more complete review of ADLs is provided in [18]. This group of ADLs is characterized by proprietary language syntax, supported by programming-language style tools (compilers, static and dynamic analysis tools, and virtual machines or simulators).

Wright is an architecture description language (ADL) whose particular focus is on formally specifying protocols of interaction among components in an architecture. To this end, it employs a subset of communicating sequential processes (CSP) [11]. Given an architectural specification, Wright is able to determine the interaction characteristics of components communicating through any given connector, e.g., whether they will deadlock.

Rapide is an ADL whose accompanying toolset provides extensive modeling, analysis, simulation, and code generation capabilities. Rapide's strengths include the ability to create executable specifications that can be dynamically analyzed and a strong notion of event-based communication.

Darwin is an ADL with precise semantics based on the pi calculus [16]. Darwin's strengths include the ability to model hierarchically constructed systems and systems that are distributed across many machines. Darwin also has limited support for dynamism, via the definition of structures that can be dynamically instantiated.

2.2. XML, DTDs, and XML Schemas

Before continuing, it is useful to have a short discussion of XML and XML schemas because of their relative youth in the world of software architecture research. Since 1996, the World Wide Web Consortium (W3C) has been working on XML, the eXtensible Markup Language [6]. "XML" refers to a family of technologies surrounding the XML 1.0 specification. This specification describes a method for marking up text documents with information about various parts of the text delimited by "tags".

In XML, fragments of text can be delimited with specially formatted start and end tags. The start tag of an element may have additional attributes that provide further information about the tagged text. The original XML 1.0 specification defines a special part of the XML language called the document type definition (DTD). By writing DTDs, XML authors can define the syntax of a document in terms of what elements and attributes are allowed. An extension to XML called XML namespaces [7] allows XML document authors to import elements and attributes from many DTDs.

While XML was initially intended to be used as a text-markup method, it has quickly grown into a method for encoding data, serializing objects, and managing metadata. The limited expressiveness of DTDs makes them insufficient for many of these purposes, so the W3C began work on XML Schema [8], a more expressive DTD replacement. The primary contribution of XML schemas is a type system for XML that supports simple and compound types, plus a method for type inheritance. Type inheritance is absolutely crucial to creating extensible XML syntaxes. It allows users to define XML types that are composed of attributes and elements, and later define extensions to those types that add (or remove) attributes and elements from those base types, similar to subtyping in an object-oriented programming language. The type extension can be defined in a separate schema from the base type, allowing developers to extend other developers' schemas without directly interacting.

2.3. XML-Based ADLs

ADLs like xADL 1.1 [13] and ADML [23] are based on XML, and are defined in XML DTDs. Defining an ADL in XML greatly increases the tool support for that ADL, as there are many COTS tools available for creating, editing, parsing, validating, storing, and manipulating XML documents. This eliminates or reduces the need for custom syntax checkers and compilers. Moreover, programmatic support for XML is provided in the form of APIs like DOM, SAX, and JDOM. More thorough rationale for building ADLs in XML is given in [22].

xADL 1.1, the predecessor of xADL 2.0 developed at the University of California, Irvine, is an architecture description language and interchange format. Its strengths include support for architectural dynamism, an architecture-based type system, and mappings from element types to implementations. A xADL 1.1 description can be used to describe how a software system should be instantiated.

ADML, developed by MCC and submitted as a standard to the Open Group, is a minimal extension to the Acme [9] representation for software architectures. ADML is currently defined in a DTD available from the Open Group [19]. ADML elements can only be extended through the addition of properties—the set of core elements in ADML is fixed. Properties are typed, and can have compound values. Meta-properties in the ADML description of a system serve as the "metalanguage" defining allowable extensions.

2.4. xArch

xArch, jointly developed by Carnegie Mellon University and the University of California, Irvine, is an XML-based representation for building ADLs. It consists of a core of basic architectural elements, defined in an XML schema called the "instances" schema. The xArch instances schema provides definitions for the following elements typically found in an ADL:

- Component, connector, interface, and link (connection between interfaces) instances;
- Subarchitectures, for specifying hierarchically composed component and connector instances; and
- Groups, allowing the combination of basic elements into logical aggregations such that they can be identified as a single entity by architectural tools.

The definitions of these elements are semantically neutral: no particular behavior is attached. Constraints on the arrangement and workings of the various architectural elements can be specified in xArch extensions (as xADL 2.0 has done).

xArch can be extended by writing new XML schemas that augment the core xArch schema with additional information about the architecture by modifying existing tags and attributes or adding new ones. This allows architecture researchers to add their own modeling constructs to xArch. In addition to the many XML schema-aware tools available off-the-shelf, several tools exist specifically to assist xArch users in creating their own extensions; these are described in Section 9.

3. Motivation

Our development of xADL 2.0 was spurred by our desire to carry out new software architecture research in configuration management, dynamism, distribution, and events. In doing so, we wished to adopt an existing ADL as the basis for our experimentation with features suitable for rapid change and research exploration. This continuous evolution requires an evolvable ADL and toolset.

We surveyed the landscape of existing ADLs, but were unable to find one with the necessary flexibility and tool-supported extension mechanism. To our surprise, this included existing XML-based ADLs, which failed to take advantage of XML's extensibility mechanism.

Extensible languages, particularly extensible programming languages, are not a new concept, having been implemented from the earliest work on Forth and Lisp to modern work on domain-specific language creation [5]. A key problem in this domain is that of *feature interaction*—determining how (and if) an arbitrary combination of extensions will work together. However, our goal in building xADL 2.0 was to foster experimentation with new extensions, not to create an ADL containing every possible feature. Furthermore, we have found that many ADL features are orthogonal (including the ones included in xADL 2.0), and the cost to resolve feature interaction problems among such extensions is relatively low.

3.1. Extending a First-Generation ADL

We first considered building xADL 2.0 by modifying an existing first-generation ADL. However, more often than not, first-generation ADLs like Darwin, Rapide and Wright share several salient features. First, they are usually custom-tailored to support only one "killer" feature that distinguishes them from other ADLs (e.g. Darwin's support for distribution, Rapide's executable specifications, Wright's protocol specifications). Second, they are supported by a proprietary toolset, usually authored entirely by the original ADL developers.

These problems arise because first-generation ADLs and their tools are mostly geared to support monolithic languages. That is, the entire ADL and all its features are defined inextricably in one place. This has several disadvantages. First, it reduces understandability because architects must become familiar with the whole ADL instead of just the necessary parts. Second, it can increase the model size greatly. Our previous work with architectures has shown promise in an approach where the architectural model of a system is maintained alongside the system itself [13]. This becomes problematic on resource-constrained devices. The ability to pick-and-choose only essential modeling constructs is key in maintaining architectural models on resource-constrained platforms. For these reasons, we wanted xADL 2.0 to be a more modular core-plus-extensions model, eventually facilitated by XML schema's extensibility mechanism.

A final problem with first-generation ADLs is that neither the language nor the tools are designed to be directly extensible. Because no modularly extensible meta-language is provided in these ADLs, their tools require significant changes to support language extensions.

3.2. Extending an XML-based ADL

XML-based ADLs like xADL 1.1 and ADML showed much more promise on the extensibility front. However, these languages offer limited possibilities for independent extension because they are based on XML DTDs. These languages have essentially inherited the extensibility problems present in first-generation ADLs.

ADML has a well-defined, but limited extension mechanism. Borrowed from Acme [9], this mechanism is based on typed name-value pair properties. The core set of elements in ADML cannot be changed. This property-based mechanism circumvents the extensibility problems caused by DTDs at the cost of adding a new metalanguage and associated tools. While XML tools can check the syntax of an ADML document, a separate set of tools must be used to check names, types, and values of the properties. Because of this, users extending ADML lose the use of existing tools that work with XML's powerful meta-language elements.

We considered converting either ADML or xADL 1.1 into an XML schema and using that as a core. But, in doing so, we would have removed non-essential elements. In the case of ADML, we would have removed the property-based extension mechanism in favor of an XML-based one. Either option would have rendered the resulting core incompatible with xADL 1.1 or ADML, sacrificing any existing tool support in the process.

3.3. Why we chose (to develop) xArch

Instead of modifying one of the existing XML-based ADLs beyond recognition, we decided to build xADL 2.0 as an xArch application. In fact, our participation in the

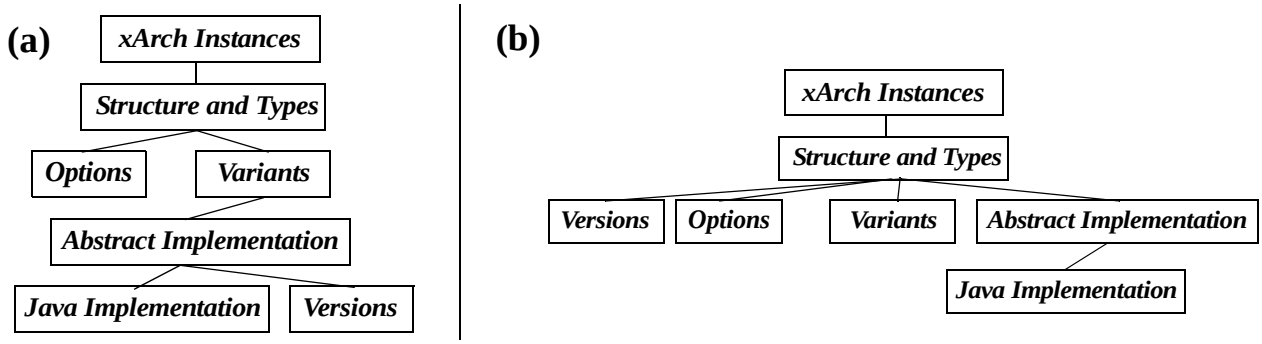

Fig. 1. Actual and conceptual dependencies of the xADL 2.0 XML schemas. Child nodes are dependent on their parent node.

development of xArch (alongside Carnegie Mellon University) was largely driven by our desire to use it as a basis for xADL 2.0.

Rather than defining its own meta-language for extension, which would lead to the problems outlined above, xArch leverages the tools and techniques developed for XML schemas. This allows us to extend the xArch core directly, adding our own first-class elements and types. For example, using XML extensions, we were able to add type systems, architectural configuration management, mappings to component/connector implementations, and more, drawing from and expanding on the research results and capabilities of previous ADLs. Furthermore, we were able to use existing XML tools to rapidly develop our extensions, giving us more time to concentrate on building xADL 2.0 tools that focused specifically on xADL 2.0 semantics.

4. Approach

4.1. Organization of the xADL 2.0 Schemas

A xADL 2.0 document is an XML document that conforms to the syntax of the xArch core and xADL 2.0 extensions to that core. It is useful to understand how the various XML schemas that comprise xADL 2.0 are organized in terms of their dependencies. One of the limitations of using XML schemas as a meta-language is that, at this time, they only support a single-inheritance model of type extension. Thus, it is not possible for two independent extensions of the same XML tag to coexist in an XML document. To avoid this problem, we have introduced some artificial dependencies among the xADL 2.0 schemas (see Fig. 1a). The XML community is considering adding support for multiple inheritance type extension in a future version of XML schemas. If this change occurs, we will rework the xADL 2.0 schemas to remove these artificial dependencies, keeping only the necessary conceptual dependencies (see Fig. 1b).

Perhaps a more useful organization of the xADL 2.0 schemas breaks them into three groups according to their purposes. One set of schemas establishes the architecture description, or run-time model, and architecture prescription, or design-time model. A second set of schemas provides mappings from components and connectors to their implementations. A third set of schemas provides support for architectural configuration management. The features provided by each schema are shown in Table 1. Each schema and feature is discussed in detail in Sections 5, 6, and 7.

Table 1. xADL 2.0 schemas and features.

Purpose	Schema	Features Provided
Architecture Modeling - Description and Prescription	xArch Instance	Component, connector, interface, and link instances; arbitrary groups; hierarchical construction.
	Structure & Types	Design-time architectural prescription; architectural structure (components, connectors, interfaces, and links), programming-language style types-and-instances model; hierarchical construction via types.
Instantiatable Architectures	Implementation	Abstract placeholder for implementation information for components and connectors.
	Java Implementation	Java-specific implementation information for components and connectors.
Architecture Configuration Management / Product Family Architectures	Options	Optional components, connectors, and links.
	Variants	Variant component and connector types.
	Versions	Version graphs for components, connectors, and interfaces.

4.1.1. Architecture Description and Prescription. The xADL 2.0 architecture model is defined in the xArch instances schema and the xADL 2.0 structure & types schema. xADL 2.0 maintains a separation between the run-time and design-time models of an architecture. In most cases, design-time information about an architecture (architectural prescription) differs greatly from run-time information (architectural description). For instance, a design-time model could indicate that a group of components are optional, whereas a run-time model would indicate whether those components were actually instantiated or not.

xADL 2.0 also maintains programming language-style type information about components, connectors, and

interfaces. A type system is valuable because it allows tools and users to make inferences about entities of the same type. Depending on the semantics associated with types, one may infer, for example, that two components exhibit similar behavior or share an implementation.

4.1.2. Implementation Mappings. Some of our previous work with software architecture models showed that, if elements in the architectural model are associated with implementations of those elements, tools can instantiate and manipulate a running system directly from the model [17]. xADL 2.0 has a schema, the abstract implementation schema, that can be extended in a straightforward manner to describe component, connector, or interface implementations in different languages.

4.1.3. Architecture Configuration Management. One issue in building adaptable and flexible architectures is the ability to manage an evolving software architecture. To compound the problem, individual elements in the architecture may evolve independently. Architecture-based configuration management is one way we are attempting to address these issues [12]. Architecture-based configuration management applies traditional configuration management concepts such as versions, options, and variants to architectural elements and architectures as a whole.

As an additional benefit, the combination of versions, variants and options allows architects to define a *product family architecture* [4]. Product family architectures are similar architectures that vary depending on their intended purpose. The use of options and variants gives an architect the freedom to specify an entire product family in a single design-time xADL 2.0 document, and then instantiate any one of the members of the product family. Versioned types allow various members of the family, or the family as a whole, to evolve over time. Some examples of product family architectures can be found in [14, 20].

Three xADL 2.0 schemas, the versions, options, and variants schemas, provide the modeling constructs to represent these CM concepts.

4.2. Tools

One of the cornerstones of our approach is the intimate tie between xADL 2.0 and xADL 2.0-aware tools. Specifically, we do not anticipate that any user will write a xADL 2.0 document directly. As noted in the introduction, the only users who should be aware of xADL 2.0's internal representation are those who are writing extension schemas. Other users are shielded from the peculiarities of XML and the xADL 2.0 structure by tools that provide users with structured APIs to a xADL 2.0 document. These tools are described, in detail, in Section 9.

5. Architecture Modeling Schemas

5.1. Instances Schema

xArch does not make a distinction between the run-time and design-time views of the system, as described by the instances schema. xADL 2.0, however, explicitly separates design-time elements from run-time elements. xADL 2.0 uses the instances schema exclusively to represent run-time instances. Design-time elements are represented in the structure & types schema, described in detail in Section 5.2. The run-time instances are the architecture *description*, described in Section 4.1.1 above. Elements from the instance schema represent actual, running counterparts in an executing software system.

In xADL 2.0's use of the xArch instances schema, component, connector, interface, and link instances are semantically neutral; that is, their behaviors are not formally specified. This is in keeping with xArch's semantically neutral nature. As such, the instance schema is basically structural, describing the topological organization of component and connector instances. In this topology, conventions from typical ADLs are employed: component and connector instances may have zero or more interface instances connected by link instances, and component and connector instances can be hierarchically constructed out of smaller-grain instances. This set of architectural elements is similar to that found in the Acme core [9]. A survey of other ADLs [18] reveals that most ADLs contain similar notions of components, connectors, interfaces and links.

xADL 2.0 leverages the general-purpose grouping mechanism provided by xArch. This provides the ability to define identifiable groups of elements that may be used by specific tools built with xADL 2.0. A group may, for example, represent elements that exist on the same machine (in a distributed system) or elements that were created by the same author. Groups in the instance schema are semantically neutral, but semantics can be added through an extension to the group structure.

5.2. Structure & Types Schema

xADL 2.0 uses the structure & types schema to model design-time elements. This is the architecture *prescription*, described in Section 4.1.1 above. We view the prescription of an architecture as a template - a recipe that can be used to instantiate a running system. Along with *structural* prescription, this schema provides a type system that allows architects to reason about design-time elements that share a common type. We view types as the ingredients that go into the recipe.

The structure and types schema builds on the instance schema and adds several modeling constructs:

- Structural prescription: the design-time template for components, connectors, interfaces, and links that can be instantiated into a running architecture;
- A type system that captures type information about these structural elements;
- A type-based hierarchical construction method;
- Groups; and
- Links between component, connector, and interface run-time instances and their design-time counterparts.

Each of these is described in detail below.

5.2.1. Structural Prescription. Whereas elements in the instance schema represent real, running instances, the components, connectors, interfaces, and links in the structure & types extension represent their design-time counterparts. These components, connectors, interfaces,

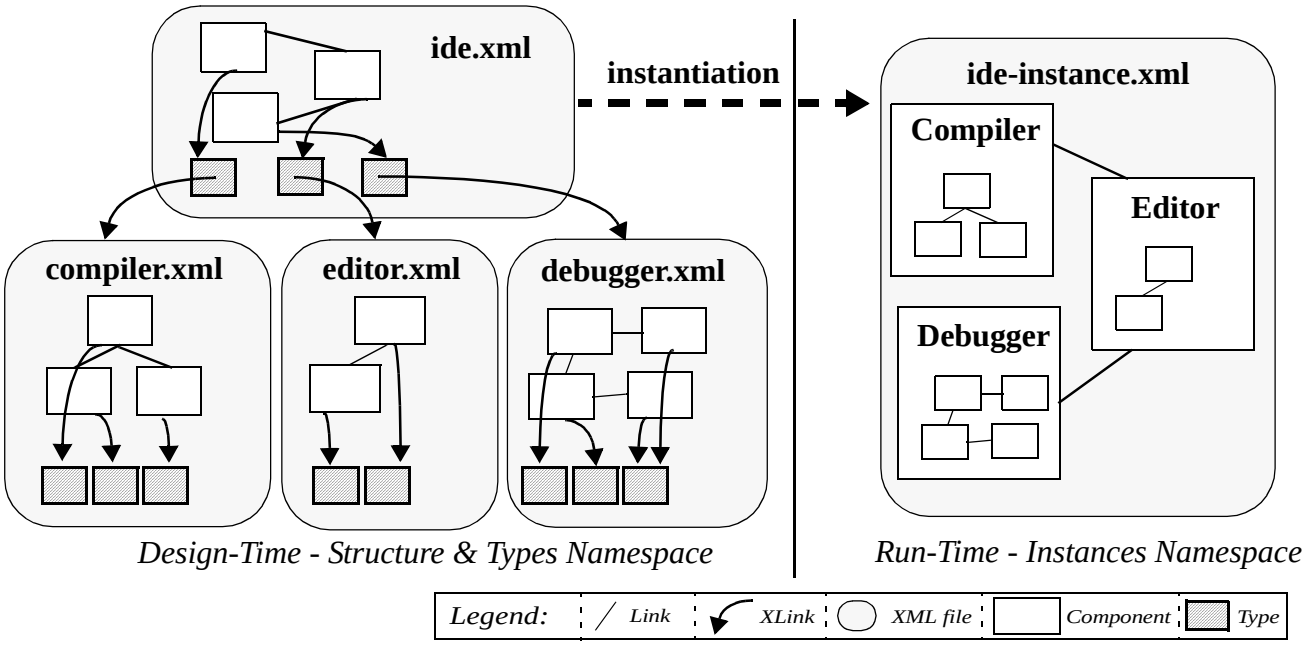

Fig. 2. Relationship between structure, types, and instances in xADL 2.0. Note that this diagram is somewhat simplified for understandability, with some elements and links not depicted.

and links are also semantically neutral; we assume behavior will be specified in future xADL 2.0 extensions.

Typical uses of the architectural prescription are to capture design rationale about the system and to serve as a template for automatic instantiation of the system.

5.2.2. Type System. The structure & types extension defines a programming-language style type system for xADL 2.0 elements. Architects can define a set of component, connector, and interface types. Each component and connector type contains a set of *signatures*, each of which references an interface type. These types, again, are all semantically neutral, and can be extended to provide semantic information. This allows the creation of different ADLs with differing type semantics. One such extension, which maps types to implementations, is described later. Each structural component, connector, and interface contains a reference to its type. With this type system, it is possible to make certain inferences about an architecture. For instance, two connectors of type "BusConnector" might have similar behavior or share an implementation. As another example, two components whose types expose the same set of interfaces may be substitutable for one another. Despite a lack of semantic information, these kinds of inferences already provide an important base level of functionality upon which to create useful abstractions and tools.

5.2.3. Hierarchical Construction with Types. The structure & types schema allows two different kinds of types: basic types and compound types. Basic types are user-defined, opaque, and have no substructure. Compound types, on the other hand, have their own structural prescription and set of types.

Recall that in the instance schema, hierarchical construction of a component or connector was represented directly in the component or connector description. This requires the entire system specification to be contained in a single document. In the design-time model, hierarchical construction is done through types. The internal structure of a compound type can be defined in a file other than the file containing the type definition. This allows architects to design complex components as separate architectures in separate xADL 2.0 documents, and link them into a larger application in a higher-level design document.

Fig. 2 shows a simplified diagram of an example integrated development environment. The three high-level components (editor, compiler, and debugger) are themselves be entirely separate applications capable of running independently. However, in this example, they are connected together in the IDE. Thus, each of them is specified in its own xADL 2.0 file, as shown in the left-hand side of the figure. This side shows the design-time documents for the compiler, debugger, editor, and IDE. In these design-time documents, the elements are from the structure & types schema. The right-hand side of the figure shows an instance document that depicts a running IDE system instantiated from the IDE's prescription, specified in the document shown on the left. The elements on the right hand side of the figure are from the instance schema.

5.2.4. Design-Time Groups. The grouping system described earlier for the instance schema is mirrored in the structure & types extension, allowing groups of structural elements and types for any purpose.

5.2.5. Instance-to-Structure Links. The structure & types schema augments the connector, component, and

interface instances from the instance schema. It adds a pointer from the instances to their structural (design-time) counterparts. This firmly establishes the connection between instances and the design-time elements that represent them.

6. Architecture Instantiation Schemas

6.1. Abstract Implementation Schema

One of the strengths of xADL 1.1 [13] was its mapping of component and connector types to implementations of those types (in the form of Java class files, for instance). If each type used in the system were mapped onto an implementation, the architecture could be instantiated and executed solely from the xADL 1.1 specification. We retained this feature in xADL 2.0, again mapping basic types to implementations. However, compound and variant types are not mapped to implementations, since the smaller-grained types that make up these composite types are mapped instead.

Depending on the language and tools being used, implementations of an architectural element can take many forms (Java class files, Windows .DLL's, etc.) Obviously, we cannot predict all these possible implementation methods in advance. So, we chose to develop a small *abstract* implementation schema that adds to xADL 2.0 an implementation placeholder on each component, connector, and interface type. This is analogous to a "virtual" or "abstract" class in an object-oriented programming language. Architecture documents replace this placeholder with concrete implementation details.

6.2. Java Implementation Schema

As an example of how the abstract implementation schema can be extended to provide support for a particular language, we have created the Java implementation schema. This schema adds to xADL 2.0 the ability to specify the location of the Java-language implementation of a component, connector, or interface type.

Specifically, each type is extended with a modeling construct that contains a set of pointers to class files. One class file is designated as the "main" class; this is the one that is actually instantiated. The other class files in the set are auxiliary class files, and their locations are specified to give Java class loaders the location of each class file required for a certain component, connector, or interface.

7. Configuration Management Schemas

As an example of how xADL 2.0 can incorporate information that is not traditionally associated with the field of software architecture, it includes support for architecture-based configuration management. Specifically, we included support for modeling the optionality of architectural elements (options), alternatives among architectural elements (variants), and the evolution of architectural elements (versions). Each one of these is specified in a separate extension, discussed below.

7.1. Options Schema

The options schema provides the ability to specify that certain components, connectors and links are optional when instantiating the architecture. To do this, the options schema extends the structural components, connectors, and links from the structure & types schema. Each optional element is annotated with a guard condition. The guard condition is programmatically evaluated when the architecture is instantiated. If the guard condition is satisfied, then the optional element will be instantiated and included in the running architecture. Otherwise, the element will not be instantiated.

7.2. Variants Schema

The variants extension provides the ability to specify that the type of certain components and connectors can vary when instantiating the architecture [12].

Recall that the structure & types extension allows two kinds of types: basic types (having no substructure) and compound types (having substructure). The variants extension adds a third kind of type: union types, here called *variant types*. Recall that in a programming language, a variable declared with a union type becomes one of many actual types when it is instantiated. Variant types in xADL 2.0 are similar. The variants extension extends the structure & types extension and allows architects to specify a variant type for components and connectors. A variant type is composed of several alternatives, one of which will be selected at instantiation time. A guard accompanies each alternative. If the guard for a particular alternative is satisfied, then that alternative is selected. Guards for a particular set of alternatives must be mutually exclusive.

7.3. Versions Schema

As a software architecture evolves and is deployed, new versions of components, connectors, and interfaces will naturally develop. Furthermore, multiple versions of a single component, connector, or interface may exist simultaneously in a single software system. Over time, older elements may be replaced with new versions of themselves, possibly at runtime. Newer elements may also be replaced with older elements, which can occur when an intolerable bug is found in a newer version of a component. Version information is also important for already-deployed systems, so deployment managers can evaluate and upgrade those systems.

The versions schema provides the ability to store version information about elements in an architecture. Specifically, it adds the following two abilities to xADL 2.0: the ability to capture version graphs for component, connector, and interface types, and the ability to relate each type to its respective version.

7.3.1. Version Graphs. The versions schema allows component, connector, and interface types to have associated version graphs. We chose to version *types* rather than *structural* elements for several reasons. First, xADL 2.0 associates basic types with implementations. Different versions of a component, connector, or interface type probably have different implementations. Second, versioning types rather than structural elements allows xADL types with subarchitectures to be versioned. This

allows designers to version an entire architecture—structure included. Since each version of a compound type is allowed to have its own structure, the structure of an application (or part thereof) may be evolved by versioning the compound type that wraps it. Similarly, to version a variant type is to version its set of possible alternatives and guards.

The version graph specifies the relationship between versions of element types, capturing the flow of its evolution. The version graph for an element type is a directed acyclic graph. Unlike RCS-style version trees, xADL 2.0 version graphs allow each node (version) to have multiple parent and child versions. A version with multiple parents indicates that it was created via a merge of its parents. xADL 2.0 also permits a parent-child relationship across version graph boundaries. This is meant for cases where *renaming* occurs; that is, when a new version of an element is split off to create an entirely new entity with its own version tree. Examples of renaming can occur when a new version of a component type has changed so significantly that it becomes a new component type, or when an existing component type is split into two or more parts, and each part evolves separately from that point on.

7.3.2. Links from Types to Versions. The versions schema extends component, connector, and interface types from the structure & types schema to add references from a type to a node in its version graph. This allows the use of versioning knowledge when manipulating the types and structure of an architecture. For example, it allows such operations as "replace component A with its latest version." Of note is that each type can be of exactly one version and that multiple versions of an element can be included in the same architecture.

8. Examples

Due to space constraints, the verbose nature of XML schema documents, and the complexity of the xADL 2.0 schemas and associated architecture descriptions, it is impossible for us to present a full treatment of the XML definition of a schema or architecture here. Furthermore, as we have stated, we do not expect users to interact directly with xADL 2.0 architecture descriptions. Rather, a growing set of COTS and in-house tools, described in Section 9, largely insulates users from the XML basis of xADL 2.0 documents. Readers interested in examining the schemas and examples of architecture descriptions directly can do so by visiting the xADL 2.0 Web site; its URL is given in Section 12.

9. Tool Support

In developing xADL 2.0, we leveraged several off-the-shelf tools and have so far built two custom tools to support xADL 2.0-based development.

9.1. Off-the-Shelf Tools

A growing number of XML schema-aware tools are becoming available to assist schema authors. Two such tools that were invaluable to us were XSV [24] and XML Spy [2]. XSV is the XML Schema Validator, built by Henry Thompson and Richard Tobin at the University of Edinburgh in Scotland. XSV was one of the first tools to support the latest W3C drafts of the XML schema specification. We used XSV to validate the syntactic correctness of our schemas. XSV also has the ability to validate an XML document against a schema (or set of schemas). We used this to validate the first xADL 2.0 example documents when the schemas were finished.

XML Spy, a commercial product from Altova GmbH, is a more graphical tool that can be used to create, edit, and visualize XML schemas and instance documents. While the version of XML Spy supporting XML schemas was not released in time for our initial development, we did use it for post-release analysis of our schemas. One of XML Spy's most useful features is the ability to generate a graphical representation (in the form of an annotated tree) of the types and elements defined in an XML schema. We found that this representation is far more useful for humans attempting to understand the schemas than the schemas themselves.

9.2. In-House Tools

DOM (the Document Object Model) provides an object-oriented interface to an XML document. It represents each element, attribute, and text segment in the document as an object that can be manipulated through the object's interface. DOM has been implemented in many languages on many platforms. Various DOM serializers exist that, given the DOM model, can output the corresponding XML document.

Because of DOM's relative ubiquity and future porting possibilities, we built a set of DOM-based libraries in Java that form a programmatic interface to xADL 2.0 documents. We built these libraries on top of an off-the-shelf XML parser and DOM implementation, Apache Xerces [3]. Unlike DOM, which has objects and interfaces that are necessarily generic (elements and attributes, for instance), our Java libraries provide interfaces and objects corresponding to the elements defined in the xADL 2.0 schemas. Rather than simple, general functions like `getElement(...)` and `addChild(...)`, our Java libraries expose xADL 2.0-specific functions like `addComponent(...)` and `addVersion(...)`. Each XML type defined in the xADL 2.0 schemas has a corresponding class and interface in our Java libraries. This insulates xADL 2.0 developers from the peculiarities of XML, such as the proper sequencing of child elements and the mapping of XML namespace prefixes to URIs. Admittedly, these APIs do require knowledge of the structure of a xADL 2.0 document. However, we are currently adding another layer of abstraction that we call the "convenience API" that will shield xADL 2.0 users from this.

A final key feature of the Java xADL 2.0 libraries is that they ignore unknown extensions found in xADL 2.0 documents, but leave the data from these extensions intact when changes are made to the document. This is extremely important since we expect xADL 2.0 to be the basis for many independently developed extensions. xADL 2.0 tools need to be able to handle specifications that contain extra information from these extensions, without losing or corrupting this information.

The Java code for these xADL 2.0 libraries can be generated programmatically, given the XML schemas. To do this, we built a tool called 'apigen.' Apigen is not

xADL 2.0-specific; rather, it can generate libraries for any xArch extension schema, including the xArch instances schema. Unfortunately, because of the complexity of the XML schema language, apigen only supports a subset of XML schema constructs. However, the subset supported by apigen is sufficient for the xArch core and all xADL 2.0 extensions to date. Support for additional XML schema constructs can be built into apigen with incremental effort.

10. Discussion

Our development of xADL 2.0 and our participation in the development of xArch was originally driven by a desire to build software architecture-based tools and development environments. Time and again, we found that software architecture researchers, including ourselves, had built their own internal representations of a software architecture for use in tools because extending an existing ADL was either too cumbersome or too time-consuming. Our previous work building the ArchStudio 2.0 [17] architecture-based development environment showed that providing a common repository (that we called ArchADT) for tools to store their architecture data reduced tool development time and increased interoperability greatly. However, many ArchStudio tool developers continued to create their own files and file formats for some data because ArchADT did not have an elegant extension mechanism.

xADL 2.0 solves this problem by making the architecture representation extensible. We anticipate that architecture tools will be built and modified to add their own information to xADL 2.0 representations, using automated tools like apigen to rapidly extend xADL 2.0 to store the additional information.

xADL 2.0's extensibility will allow us to quickly prototype and model new architectural constructs as our research interests evolve and expand. Currently, our focus is shifting to modeling dynamic, distributed software architectures that can be changed (and change themselves) at run-time. To support this, we will need an ADL that can store the state of components and connectors, events that are traversing the architecture, and a method to represent an architecture that is distributed across many machines. Since we do not yet know the exact set of modeling facilities we will use, we will need to experiment with and prototype new modeling constructs. xADL 2.0 provides us the basis with which to do this.

We realize that, compared to other, existing ADLs, the current set of features in xADL 2.0 does not incorporate features such as behaviors, constraints, and formally-specified protocols. This is partially due to the fact that we are exploring *new* constructs. Our success in developing xADL 2.0 extensions for these novel features indicates, to us, that incorporating more common features will be equally possible.

11. Conclusions and Future Work

xADL 2.0 is an extensible ADL built such that architecture researchers can quickly experiment with new modeling constructs and features in architecture description languages. To date, it includes several key features (separation between design- and run-time, implementation mappings, and configuration management) that we feel are inadequately represented in comparable ADLs. We will further extend xADL 2.0 in the near future with constructs corresponding to our research in architectural distribution and dynamism.

We realize that the extensibility provided by XML schemas as used by xADL 2.0 is closely related to the mechanisms used in extensible programming languages. Other relevant paradigms include intensional programming, aspect-oriented programming, and even configuration management concepts such as change sets. As part of our future evaluations, we intend to make a detailed comparison to those paradigms and adopt their unique features as may be necessary to further strengthen the extensibility of xADL 2.0.

It is our ultimate desire to refactor the ArchStudio environment [13] and tools to use an extensible xADL 2.0-based version of ArchADT for storing and manipulating architecture data. We are currently working on a set of "convenience APIs" that will be able to directly provide information about the architecture that is only indirectly provided in the xADL structure. For instance, each component instance has a reference to its type, but component types currently do not have references to their instances. Rather than also modeling these kinds of references in the xADL 2.0 extensions and relying on a user to maintain the links, the convenience APIs will provide automated functionality to discover, for example, the set of instances of a given type. The convenience APIs will also allow transparent traversal of references across files, providing the user with the illusion of a single, coherent architectural model—even if the model itself is distributed across many files. The goal of these convenience functions is to insulate users completely from the peculiarities of the xADL 2.0 document structure. Eventually, we intend to wrap the xADL 2.0 Java libraries and these convenience APIs in an event-based component that can update ArchADT in the next release of our ArchStudio environment.

Additionally, xADL 2.0's support for configuration management ties in with our research on CM-based architecture evolution. We plan to use xADL 2.0 as an architecture representation in future versions of Ménage [12]. The goal of the Ménage project is twofold: a) to precisely capture a product family architecture by jointly utilizing techniques from configuration management and software architecture and b) to use the resulting definition to create novel, component-based software development tools.

The xADL 2.0 specifications and tools are intended for software architecture researchers that want to experiment with new modeling constructs and manipulate them without rebuilding an entire modeling language from scratch, as has been done so many times before. XML provides the extensibility mechanism, and xADL 2.0 provides an expanded set of useful constructs that can serve as the basis (through additional extension) for more expressive ADLs. While we have not yet reached a stage in our development where we can comment on xADL 2.0-based tool interoperability, we anticipate that integrating xADL 2.0-based tools will be much easier than integrating tools that use their own proprietary representation for architectures.

12. URL

xADL 2.0, xArch, and their associated tools are available at:

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

13. Acknowledgements

The authors would like to thank David Garlan and Bradley Schmerl at Carnegie Mellon University for their cooperation on the creation of the xArch instance schema. The authors would also like to acknowledge the invaluable feedback and contributions of David Rosenblum, Yuzo Kanomata and Craig Snider at UCI, Nenad Medvidovic and his students at USC, Nicolas Rouquette and his team at JPL, and Alex Wolf, Dennis Heimbigner, and Nathan Ryan at CU Boulder. Finally, we would like to thank the anonymous reviewers for their assistance in focusing and clarifying many aspects of the paper.

14. References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] Altova GmbH. XML Spy Software. *http://www.xmlspy.com/*. January, 2001.

[3] The Apache Group. Xerces Java Parser. *http://xml.apache.org/*. January, 2001.

[4] D. Batory, "Product-Line Architectures." Invited presentation, Smalltalk und Java in Industrie and Ausbildung, Erfurt, Germany, October 1998.

[5] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings of the 5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, eds. Extensible Markup Language (XML) 1.0 (Second Edition). *http://www.w3.org/TR/2000/REC-xml-20001006*. October 6, 2000.

[7] T. Bray, D. Hollander and A. Layman, eds. Namespaces in XML. *http://www.w3.org/TR/1999/REC-xml-names-19990114/*. January 14, 1999.

[8] D. C. Fallside. XML Schema Part 0: Primer. *http://www.w3.org/TR/xmlschema-0/*. October 24, 2000.

[9] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.

[10] D. Garlan and M. Shaw. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing, 1993.

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

[12] A. van der Hoek. Capturing Product Line Architectures. In *Proceedings of the 4th International Software Architecture Workshop*, Limerick, Ireland, June 2000.

[13] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and R. N. Taylor. xADL: Enabling Architecture-Centric Tool Integration With XML. In *Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS-34)*, Maui, Hawaii, January 3-6, 2001.

[14] J. Kuusela. Architectural evolution. In *Proceedings of the First Working IFIP Conference on Software Architecture*, pages 471—478, Boston, Massachusetts, February 1999. Kluwer Academic.

[15] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, September 1995.

[16] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. *Proc. of 5th European Software Engineering Conference (ESEC '95)*, Sitges, September 1995.

[17] N. Medvidovic, P. Oreizy, R. N. Taylor, R. Khare, and M. Guntersdorfer. An Architecture-Centered Approach to Software Environment Integration. *Technical Report UCI-ICS-00-11, Department of Information and Computer Science, University of California, Irvine.* March, 2000.

[18] N. Medidovic, R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*. 26(1):70-93. January, 2000.

[19] The Open Group. ADML Document Type Definition. *http://www.opengroup.org/public/arch/p4/adml/adml.dtd.*

[20] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for product families in consumer electronics software. *Computer*, 33(2):78—85, March 2000.

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

[22] S. Pruitt, D. Stuart, W. Sull, and T.W. Cook. The Merit of XML as an Architecture Description Language Meta-Language. *Microelectronics and Computer Technology Corporation White Paper.* January 28, 2000.

[23] J. Spencer, ed. Architecture Description Markup Language (ADML): Creating an Open Market for IT Architecture Tools. *Open Group White Paper.* September 26, 2000.

[24] World Wide Web Consortium. Validator for XML Schema. *http://www.w3.org/2000/09/webdata/xsv.* September, 2000.