xADL: Enabling Architecture-Centric Tool Integration With XML

Rohit Khare[†] **Michael Guntersdorfer**[†] **Peyman Oreizy**[†]
Nenad Medvidovic[††] **Richard N. Taylor**[†]

[†]Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{rohit,mgunters,peymano,taylor}@ics.uci.edu

[††]Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
neno@usc.edu

Abstract

In order to support architecture-centric tool integration within the ArchStudio 2.0 Integrated Development Environment (IDE), we adopted Extensible Markup Language (XML) to represent the shared architecture-in-progress. Since ArchStudio is an architectural style-based development environment that incorporates an extensive number of tools, including commercial off-the-shelf products, we developed a new, vendor-neutral, ADL-neutral interchange format called Extensible Architecture Description Language (xADL), as well as a "vocabulary" specific to the C2 style (xC2). This paper outlines our vision for representing architectures as hypertext, the design rationale behind xADL and xC2, and summarizes our engineering experience with this strategy.

Keywords: *Software environments, software architectures, architecture description language (ADL), off-the-shelf tool integration, XML*

1. Introduction

The widespread adoption of Software Engineering Environments (SEE) predates the term itself. The rational decomposition of the development process and its artifacts can arguably be traced back to the advent of separate compilation. The zeroth generation of SEE integration took files as its primary unit of discourse between various tools. The well-known UNIX pipe-and-filter architectural style was entirely based on the concept that one tool's output data could be used as an input to another [12].

The first step, then, was a repository-centric approach for managing these artifacts. In such an Integrated Development Environment (IDE), different tools would work upon a central, shared database representing the product-in-progress. The archetype of this generation was the Stoneman reference model for the Ada Program Support Environment [7]. Interlisp [28] can be seen as one instance of this approach with suite of tools operating on a shared parse-tree. A versioned filesystem was another popular variant, notably Revision Control System (RCS) [29].

Continuing the ascent, a second generation of process-centric IDEs emerged in the 1980s which took relations between these artifacts and their associated workflows as its unit of discourse. Tools such as Marvel [3] assisted developers by automating basic process steps and coordinating the work of tools "outside" the development path proper. In the extreme, IDE support tools maintained *only* those relations, as in the Chimera Linkbase [2].

Finally, we see the current era as the advent of architecture-centric IDEs that control the evolution of software throughout its lifecycle using architecture descriptions as its primary unit of discourse. As an example, ArchStudio 2.0 assumes the existence of versioned repositories and process automation in its foundation, and so focuses on the design, evaluation, instantiation, and editing of C2-style architectures. Supporting tool integration in this generation now requires an open, hypertext web representing the entire product, from architecture down to development artifacts.

We developed an Extensible Markup Language (XML) syntax for Architecture Description Languages (xADL) and customizations to the C2 style in particular (xC2) in support of these goals. Furthermore, we integrated our XML Abstract Syntax Tree (AST) and an Abstract Data Type (ADT) representing the architecture-in-progress to the ArchStudio 2.0 environment on the fly, thereby highlighting the ease of dynamism in an IDE itself designed in the C2 style. By wrapping a parsed, shared representation of the work at hand behind a (potentially-distributed) event notification interface, our approach technologically updates the basic strategy of Field [25].

The balance of this paper focuses particularly on the role XML played in the integration of several tools within ArchStudio 2.0. We shall outline the origins and

promise of XML broadly, our detailed design of xADL and xC2, our implementation experience with it, and a discussion of its implications for broader and deeper tool integration within architecture-centric IDEs.

2. Extensible Markup Language (XML)

The HyperText Markup Language (HTML) allows the structural markup of World Wide Web documents. Now, HTML's evolutionary successor, XML, takes document markup to the next level, by offering human-readable semantic markup that is also machine-readable. As a result, XML makes it dramatically easier to develop and deploy new mission-specific markup, enabling the automation of the authoring, parsing, and processing of networked data.

Broadly speaking, the XML 1.0 standard [6] is a simplification of the Standard Generalized Markup Language (SGML) which itself dates back to the mid-1960s. XML should be seen as a toolkit for creating new elements (also known as "tags") and attributes upon them, as well as grammar rules governing the parse tree. All of these rules are captured in a Document Type Definition (DTD), which can be used to formally validate any XML instance (file) against it.

More usefully, though, XML defines a lower level of conformance known as "well-formed." This level merely assures that all the elements open and close properly, and so on. Such purely mechanical checking allows designers to "mix and match" elements from several DTDs. In particularly, the XML Namespaces facility allows us to interpret unknown element names as URLs which can be further investigated automatically or by hand. Namespaces thus disambiguate potentially conflicting tag semantics.

Our data integration strategy for ArchStudio 2.0 adopted XML for several of the properties introduced above:

- A text format governed by an open standard promised future-proof file formats. Furthermore, the advent of hybrid XHTML modules already provided a rich tagset for human-readable documentation and presentation of the architecture-in-progress.
- Well-formed XML accommodated multiple tools' own subtrees within the data model, as well as tool-specific attributes decorating existing elements.
- Namespaces explicitly articulated separate control over the vocabulary for describing high-level architectures in common (xADL), style-specific features (xC2), and tool-specific additional data.
- Intrinsic support for hypertext linking encouraged future refactoring of architecture description into separate resources describing individual components and types, potentially published remotely by several developers (hyperlinked reuse).
- Finally, rich protocols for accessing, managing, and versioning XML repositories already existed, in the form of WebDAV (Web Distributed Authoring and Versioning) extensions to HTTP and the XPointer language for hyperlinking directly within XML documents.

3. Approach

ArchStudio [22] is an architecture-centric IDE, based on the C2 architectural style [27]. The C2 style is an event- and component-based style that allows dynamic evolution of the software system at runtime [13]. C2 does so by enforcing system decomposition into units of computation and data store, called *components*, and units that enable the interaction among components, called *connectors*, as suggested by Garlan and Shaw [11]. The highly dynamic nature of the C2 style is primarily caused by a central rule of the style, which does not allow direct links between two components, but rather requires the involvement of a connector in between. Hence components may be plugged in and out of the system without leaving another component behind with a dangling link. The separation of architectural units into components and connectors may be regarded analog to the nodes and transitions concept of Petri nets [23], where direct links may only exist from nodes to transitions and vice versa, but not within each domain. However, the C2 concept differs insofar as connector may be linked directly to other connectors.

ArchStudio not only supports the development of C2 style software, but was itself implemented in the C2 style. Such support for software evolution made ArchStudio itself an excellent target for incremental tool integration. Version 2.0 built upon a suite of tools already developed for ArchStudio 1.0 to add a new shared repository format (xADL), new style-checking tools, and several commercial- and research-grade off-the-shelf technologies, including Rational Rose [24], Armani [20], Metamata [15], and JavaBeans [26] as shown in Figure 1.

The first step required introducing an abstract data type representing the architecture-in-progress, ArchADT, quite apart from its new representation in an XML-based abstract syntax tree (AST). Each step had a beneficial consequence:

1. Since the message-based C2 style transmits by-copy rather than by-reference, globally sharing an

ArchADT replaced the habit of sending enormous messages representing the entire architecture-in-progress. Furthermore, the new technique removed artificial sequence dependencies for tools that modified disjoint aspects of the architecture.

2. While ArchStudio 1.0 tool integration was either limited to tools that understood its internal application programming interface (API) or were mapped to it using component wrappers, the advent of XML at least eased data integration with off-the-shelf tools. Since the AST supported lowest-common-denominator access for tools which added new subtrees of information, or new attributes of existing objects, an XML AST enabled integration of tool-specific information transparently, within the ArchADT.

In particular, we designed xADL to be a shared language for representing a variety of possible ADLs. It introduces five basic tags, namely <Architecture>, <Component>, <Connector>, <ComponentType>, and <ConnectorType>, each with its own subtrees as well as hyperlinks between them.

- xADL
    - Architecture
        - Links
- Component
    - Supports
- ComponentType
    - Interface
        - Parameter
- Connector
    - Supports
- ConnectorType
    - Interface
        - Parameter

where

- *xADL->Architecture->Links* specifies a list of directed hyperlinks between component and connector instances (links to <Component> and <Connector> tags, respectively),
- *Component->Supports* specifies name and type(s) supported by a component instance,
- *ComponentType->Interface* specifies name and method interfaces for each component type,
- *ComponentType->Interface->Parameter* specifies input and output parameters of a component interface,
- *Connector->Supports* specifies name and type(s) supported by a connector instance,
- *ConnectorType->Interface* specifies name and method interfaces for each connector type, and
- *ConnectorType->Interface->Parameter* specifies input and output parameters of the connector interface.

xADL may be extended to support a particular architectural style, such as C2, by mixing-in additional XML Namespaces. In our case, xC2 added C2-specific tags, attributes, and constraints to the specification, as shown in the example in Figure 2.

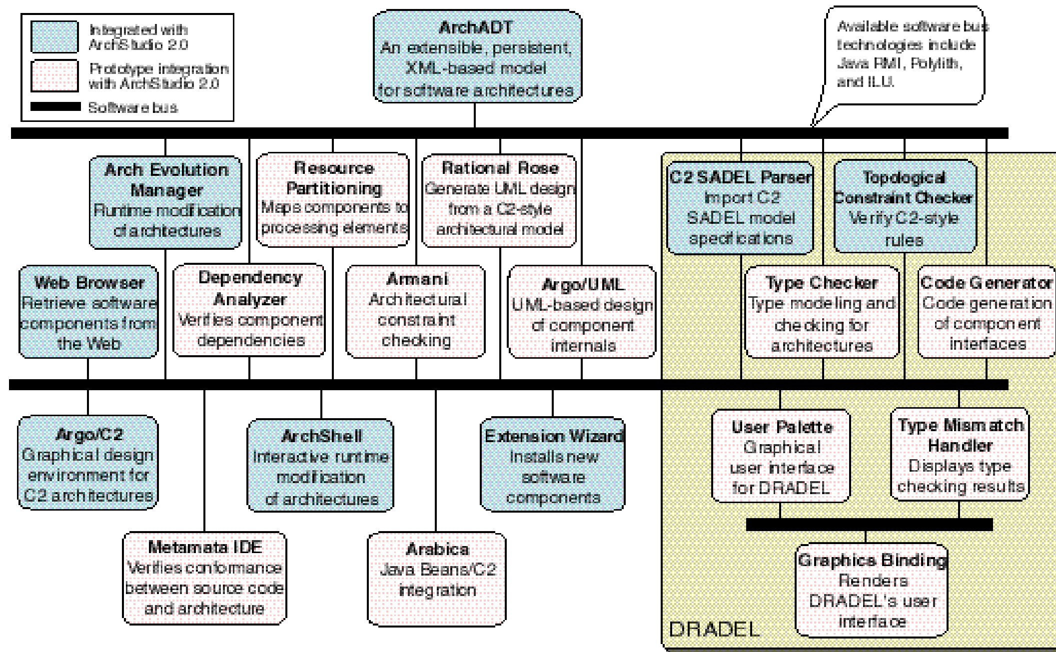

Figure 1. ArchStudio 2.0 integrates several tools according to the principles of the C2 style.

```
<!-- xC2 adds one parameter to the xADL:ConnectorType tag, namely a message filtering policy with a
default of none -->
<!ELEMENT xADL:ConnectorType(Supports*)>
<!ATTLIST xADL:ConnectorType name CDATA #REQUIRED
          xC2:filter   (no_filtering              |
                        notification_filtering    |
                        message_filtering         |
                        prioritized               |
                        message_sink)             'no_filtering'>
```

Figure 2. An excerpt from the combined xADL and xC2 DTD.

xADL also supports the storage of tool-specific information as well-formed XML data, though at the expense of formal validation against a single, unified DTD. Leaving the interpretation of data to participating tools still allows ArchStudio to centrally manage and monitor the architecture-in-progress, even if it does not "understand" the data it stores.

4. Implementation Issues

The first tool to be upgraded was DRADEL [19]. As the hub of the first-generation system, it was used to model architectures in the C2 style, check and enforce style constraints, manage heterogeneous subtyping, and generate application skeleton code. The central artifact representing the architecture-in-progress was a read-only, edit-externally C2SADEL text file.

Our new repository strategy was implemented at two layers: a "physical" AST that guaranteed storage of well-formed XML fragments; and a "logical" ArchADT that enforced the grammar and semantic rules (validity) of xADL and xC2. For the former service, we published a request-response message interface for basic tree manipulation: CreateTree, AddChildNode, DeleteNode, SetAttribute, GetNodeInfo, ReadFromFile, SaveToFile, and so on. At the logical layer, we preserved the existing interface at a higher level of abstraction, such as NewArchitecture, AddComponent, BasicSubtypeOf, and so on. Two functions, toAST and fromAST, were used throughout the class hierarchy to keep the two levels synchronized; each logical object in the architecture maintained a "shadow node" in the AST with its current state.

Thus, by contrast to the hand-coded text input parser of the previous generation, using off-the-shelf XML parsing technology automatically provided "round-trip" input and output of the AST at any point.

To dynamically evolve from the existing ArchitectureSpec object that reflected a C2SADEL-format input file to the new ArchADT object bound to an XML-format representation, we applied a transition strategy which mirrored the notification stream used to construct the former onto the latter. The central challenge within that process, in turn, was managing the extreme parallelism of our new XML AST store, since the original environment was entirely sequential by virtue of passing the entire ArchitectureSpec object by value from parser to parser. Naturally, replacing massive copying of the entire architecture-in-progress with many more small queries against a shared representation eased physical distribution of ArchStudio 2.0 across LANs. Replacing in-process procedure calls with remote ones has its limits at Internet-scale, however. While we did not enact ArchStudio 2.0 across high-latency public Internet links, the "chatty" pattern of AST access could accumulate into unacceptable overall performance.

Serializing edits to the shared AST was another difficult problem to address within the C2 event-based architectural style. Since it does not make any assumptions about the order messages are sent and retrieved within the system, edits must be tracked by unique identifiers or treated as idempotent (that is, reorderable). Of course, one atomicity solution is strict sequencing of requests and responses with one speaker at a time, as ArchStudio 1.0 did. This corresponds to a depth-first construction of the AST, while our parallel approach permits multiple tools to construct or analyze portions of the architecture simultaneously.

The problem, then, is that several messages or notifications of the exact same type, sometimes even with the same parameters, might be outstanding at the same time. Two notices that ChildAdded 38 <Component> 40 and ChildAdded 38 <Component> 41 cannot be reliably demultiplexed back to their respective initiators (the first numerical parameter is a common parent id and the second is a newly-minted nodeID not initially known to the requestor).

Our solution was an additional optional parameter which identified the calling component, ref_com. Sending a pointer to the calling object on a round-trip through the AST interface was essential to decentralizing access to it within ArchStudio2 (i.e. multiple parallel readers and writers). Note, though, that the sequential delivery of requests by the C2 connector

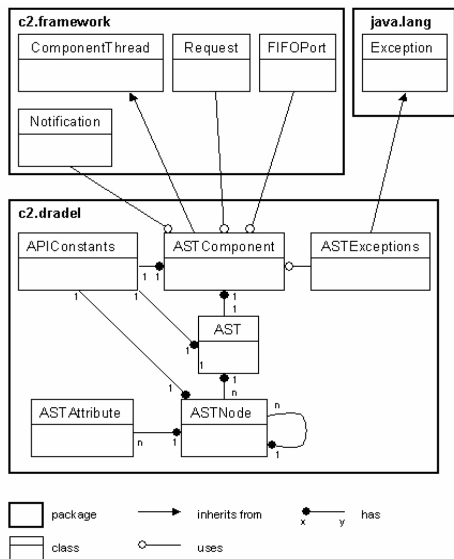

Figure 3. ArchADT component architecture in Booch notation [4].

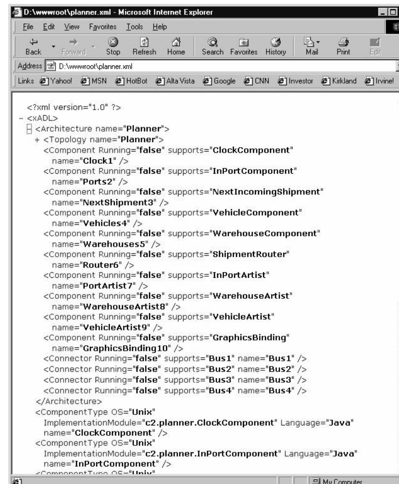

Figure 4. An example xADL file viewed in Microsoft Internet Explorer 5.0 rendered by our default XSL stylesheet.

interface wrapping the AST still implicitly serialized all I/O requests.

The complete decomposition of ArchADT includes several more classes, as shown in Figure 3. AST, ASTNode, and ASTAttribute represent the tree anchor (or root), nodes within the tree, and attributes of the tree nodes, respectively. Good Java programming style packaged the constant string values and request names within an APIConstants class, as well as a central ASTExceptions class. Finally, the entire suite was encapsulated by an ASTComponent class which handled the external event interface according to the C2 style.

5. Discussion

There are several interesting implications of our adoption of XML and our design of xADL and xC2 in particular. First, adopting a standard syntax for representing the tool data being integrated within ArchStudio 2.0 should generate compelling "network effects" for our product. That is to say, there is dramatic potential for reuse with XML over our previous custom text format. Already, we have been able to leverage off-the-shelf parsing technology, user interfaces (Figure 4), syntax-directed editors (Figure 5), and schema development tools.

Most of all, the XML developer community is already oriented towards enabling "mix-and-match" reuse of application-specific ontologies. For example, the World Wide Web Consortium is developing the Scalable Vector Graphics language (SVG, [8]) for drawings. A future Unified Modeling Language (UML, [5]) graphical editor could produce SVG documents which could be transparently annotated with xADL and xC2 descriptions of the components and connectors those boxes and lines represent.

Second, the approach we have adopted in xADL can be easily extended to support multiple architecture description languages (ADLs), even within a single XML schema. Our extensive study of ADLs [17] has indicated that most all mainstream ADLs agree on the existence of components, connectors, and their configurations. A small number of ADLs, including Rapide [13] and Darwin [14], do not explicitly model connectors. However, even these ADLs support simple component interconnections; furthermore, Rapide employs specialized "connection components" to support more complex interactions. Additionally, all ADLs model component interfaces and do so in a relatively uniform

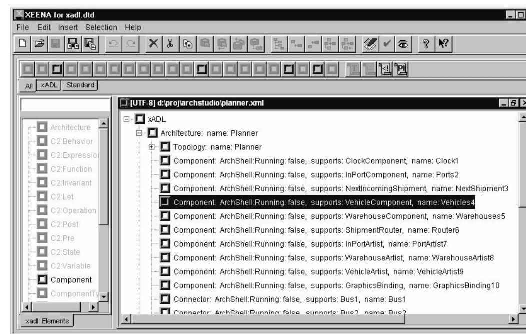

Figure 5. IBM's Xeena is a DTD-driven editor which can thus automatically enforce XML validation rules on xADL/xC2 files.

fashion. Therefore, these shared aspects of ADLs would become part of the basic xADL schema.

That basic schema could then be extended in a number of ways to represent the varying parts of architectural descriptions across ADLs, such as the manner in which ADLs model architectural semantics, support evolution (both at system design time and run time), constrain the architecture (and its evolution), and so forth. Thus, for example, an xADL schema could simultaneously describe architectures specified in C2SADEL [19] and Wright [1]. If a particular tool is interested in the static model of behavior, it would access C2SADEL's component invariants and pre- and postconditions; alternately, if the tool is interested in the system's dynamic semantics, it would access Wright's CSP-related items and ignore others. Another possibility that xADL affords us is the support for multiple configurations of the same set of components, where we access the part of the schema representing the specific configuration we are interested in, disregarding all other configurations.

In order to assess several of the hypotheses outlined above, we have performed an initial evaluation of xADL's ability to represent heterogeneous ADLs and do so in a single schema. To this end, in addition to C2SADEL, xADL has been used as the basis for representing significant portions of ACME [10], Darwin [14], and SADL [21]. xADL provided an adequate basis for this exercise. Similarly to the above discussed C2-specific tags in the DTD (xC2), the DTD resulting from this exercise also contains pertaining to the other three ADLs (xACME, xDarwin, and xSADL, respectively). For example, 6 depicts the DTD portions that deal with SADL's architectural constraints, ACME's architecture families, and Darwin's component instances. Several example architectures from the four ADLs have also been represented in this framework as a demonstration of xADL's utility.

xADL is not without shortcomings, however. Convenient representation of sub-architectures is the biggest need; several alternatives for achieving this are currently under investigation. As a wider circle of investigators evaluates and uses the technology we expect to see the schema evolve. We expect that the very properties of XML that attracted us in the first place will make this evolution a relatively convenient process.

6. Conclusions

We adopted XML as a key technology for enabling architecture-centric tool integration in the ArchStudio 2.0 IDE. The C2 style eased the evolution from the previous version's custom text file format, C2SADEL, to a generic XML AST as the repository. This had immediate benefits for integrating several tools' data in the same file, for annotating existing data without interfering with its original use, and for hyperlinking to external data transparently.

Furthermore, we developed a new ontology for describing entire families of Architecture Description Languages (ADLs). By extracting the five most common abstractions and their relations into a top-level xADL namespace, we were able to separately represent data specific to the C2 architectural style and C2SADEL in a subsidiary xC2 namespace; additionally, we were also able to represent the data specific to the ACME, Darwin, and SADL languages within their own namespaces.

These technologies directly aided a strictly distributed team to integrate a substantial set of research and commercial tools within ArchStudio 2.0. Our eventual aim is even wider, to support Internet-scale development, with potentially large and varying developer communities composing systems over long times and distances [9]. Representing architectures as hypertext

```
<!ATTLIST    xSADL:Constraint
             name         CDATA                    #REQUIRED
             type         CDATA                    #REQUIRED
             source       CDATA                    #IMPLIED
             destination  CDATA                    #IMPLIED >
<!ELEMENT xAcme:Family
             (xAcme:ComponentType | xAcme:ConnectorType |  PortType | xAcme:RoleType |
             xAcme:PropertyType | Component | Connector | Port | xAcme:Role |
             xAcme:Property | Topology | xAcme:Representation)+>
<!ATTLIST   xAcme:Family
             identifier   ID                       #REQUIRED>
<!ATTLIST  xDarwin:Inst
             name         CDATA                    #REQUIRED
             prop         CDATA                    #IMPLIED
             type         CDATA                    #REQUIRED>
```

Figure 6. An excerpt from the DTD generated in the process of modeling ACME, Darwin, and SADL in xADL.

affords us reach; extracting our ontology in XML promises depth, through integration with generic, non-ADL-aware XML applications.

7. Acknowledgments

We wish to acknowledge the following individuals for their participation in the work described in this paper. ArchStudio 1.0 was developed by P. Oreizy and N. Medvidovic. ArchStudio 2.0 was developed by P. Oreizy, R. Khare, M. Guntersdorfer, K. Nies, E. Dashofy, Y. Kanomata, R. Natarajan, A. Hitomi, R. Klashner, L. Pan, M. Dias, M. Vieira, S. Devanathan, and J. Robbins. Ebru Dincel and Roshanak Roshandel performed the exercise of modeling ACME, Darwin, and SADL, in addition to C2SADEL, in xADL.

This effort was sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement numbers F30602-97-2-0021, F30602-99-C-0174, F30602-0-2-0607, and F30602-00-2-0615. The effort was also sponsored by the National Science Foundation under grant number CCR-9985441. 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, Air Force Research Laboratory or the U.S. Government.

8. References

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

[2] K. M. Anderson, R. N. Taylor, and E. James Whitehead, Jr., Chimera: Hypertext for Heterogeneous Software Environments, *Proceedings of the 1994 European Conference on Hypermedia Technology ECHT '94*, Edinburgh, Scotland, 1994.

[3] N. S. Barghouti. Supporting Cooperation in the Marvel Process-Centered SDE, *5th ACM SIGSOFT Symposium on Software Development Environments*, edited by Weber, H. pp21-31 December 1992.

[4] G. Booch, Object-Oriented Analysis and Design with Applications, Benjamin/Cummings, Redwood, City, CA, 1996.

[5] G. Booch, I. Jacobsen, and J. Rumbaugh, The Unified Modeling Language User Guide, Addison-Wesley, Reading, MA, 1998.

[6] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, eds. Extensible Markup Language (XML) 1.0, *World Wide Web Consortium Recommendation*, 1998.

[7] DoD. Requirements for Ada Programming Support Environments: STONEMAN. United States Department of Defense, Office of the Under Secretary of Defense for Research and Engineering, 18 February 1980. NTIS-AD-A100 404/3.

[8] J. Ferraiolo, et al. Scalable Vector Graphics (SVG) 1.0 Specification, *World Wide Web Consortium Working Draft*, 1999.

[9] R. T. Fielding, E. J. Whitehead, Jr., K. M. Anderson, G. A. Bolcer, P. Oreizy, and R. N. Taylor, Web-based Development of Complex Information Products, *Communications of the ACM, 41(8), August* 1998.

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

[11] D. Garlan and M. Shaw, An Introduction to Software Architecture, *Advances in Software Engineering and Knowledge Engineering, Volume I*, World Scientific Publishing Company, NJ, 1993.

[12] B. Kernighan and R. Pike, The UNIX Programming Environment, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[13] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717-734, September 1995.

[14] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the Fifth European Software Engineering Conference* (ESEC'95), Barcelona, September 1995.

[15] Metamata IDE, Metamata Corp., http://www.metamata.com/

[16] N. Medvidovic, Architecture-Based Specification-Time Software Evolution, *Ph. D. Dissertation*, University of California, Irvine, 1998.

[17] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.

[18] 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.

[19] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering* (ICSE'99), Los Angeles, CA, May 1999.

[20] R. T. Monroe, Armani Language Reference Manual, *Technical Report CMU-CS-98-163*, Carnegie Mellon University, School of Computer Science, 1998.

[21] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4), pages 356-372, April 1995.

[22] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An Architecture-Based Approach to Self-Adaptive Software, *IEEE Intelligent Systems*, 14(3), pages 54-62. May/June 1999.

[23] J. L. Peterson. Petri Nets, *ACM Computing Surveys, 9(3):223-252, September* 1977.

[24] Rational Software Corp., Rational Rose 98: Using Rational Rose, 1998.

[25] S. P. Reiss, Connecting Tools Using Message Passing in the Field Environment, *IEEE Software, July* 1990.

[26] Sun Microsystems, Inc., Enterprise JavaBeans 1.1, *Draft Specification*, http://java.sun.com/products/ejb/newspec.html.

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

[28] W. Teitelbaum and L. Masinter, The Interlisp Programming Environment, *IEEE Computer, April* 1981.

[29] W. F. Tichy. RCS: A System for Version Control, *Software Practice and Experience*, 15(7):637-54, July 1985.

Appendix: The xADL / xC2 Document Type Definition (DTD)

```
<?xml version=”1.0” encoding=”US-ASCII”?>
<!-- Revision 1 of xADL/1.0 -->
<!-- xADL
    This is an arbitrary parent/container element that
    demarcates the beginning of a “xADL block” of
    information.

    In the future, we want xADL webs of documents spun
    from individual XML instances describing single
    instances, interfaces, or architectural configurations.
    However, initial implementations will use a single file,
    so we created a “surrogate parent” element which
    functions as the root for XML purposes -->

<!ELEMENT      xADL          (ComponentType |
                             ConnectorType |
                             Architecture)*>
<!-- ARCHITECTURE
    The Architecture tag represents a single configuration
    of instances by the directed edges between them.
    Thus, the content model is a list of Link tags. -->

<!ELEMENT      Architecture  (Component |
                             Connector )* ,
                             Topology>
<!ATTLIST Architecture name  CDATA
#REQUIRED>
<!ELEMENT      Topology      (Link*)>

<!-- LINK
    The Link Element represents a single directed edge
    between any two instances (Components or
    Connectors). In particular, the edge endpoint can be
    specified as a particular Port by using XLink Fragment
    identifiers, e.g., gameboard#Top, or
    vault#DepositorySlot. -->

<!ELEMENT      Link          EMPTY>
<!ATTLIST Link from          CDATA     #REQUIRED
               to            CDATA     #REQUIRED
               name          CDATA     #IMPLIED >

<!-- COMPONENT TYPE
    This tag has only one xADL attribute, name. At the
    basic xADL level, the only salient feature of a
    ComponentType is its Methods. A type hierarchy, if
    any, can be constructed with the Supports child
    element

    C2, in particular DRADEL, adds analysis of
    “behavior”, for which see the optional child tag
    Behavior -->

<!ELEMENT ComponentType     (Supports*,
                             Method*,
                             xC2:Behavior?)>
<!ATTLIST ComponentType  name CDATA
#REQUIRED>

<!-- CONNECTOR TYPE
    This tag has only one xADL attribute, name. At the
    basic xADL level, there are no further refinements on a
    Connector except its type relations (if any).

    C2 adds the attribute filter, specifying one of several
    predefined choices. The default setting is no_filtering--
    >

<!ELEMENT ConnectorType     (Supports*)>
<!ATTLIST ConnectorType
          name       CDATA      #REQUIRED
          xC2:filter (no_filtering|
          notification_filtering    |
          message_filtering         |
          prioritized               |
          message_sink )            'no_filtering'>
<!-- METHOD
    The Method tag describes a single operation upon
    several Parameters (which can represent inputs,
    outputs, or both).

    C2 adds the direction attribute.
    C2, in particular DRADEL, referred to these as
    InterfaceElements.
    C2, in particular DRADEL, maps each Method to an
    Operation for behavioral analysis.
    xC2:mapToOper cites a value for uid which occurs on
    an Operation (since it is not clear if Operation names
    are unique enough) -->

<!ELEMENT Method    (Parameter*)>
<!ATTLIST Method
          name       CDATA      #REQUIRED
          xC2:direction (provide | require)    #REQUIRED
          xC2:mapToOper IDREF        #IMPLIED >
<!-- PARAMETER
    To support arbitrary method signatures in xADL,
    parameter roles (in, out, inout; default of 'in') are
    specified separately for each parameter. This allows
    multiple results, for examples. The Parameter tag is an
    empty element. Its type attribute is an opaque string
    for our purposes (e.g. “Int”)

    C2, in particular DRADEL, maps each Parameter to a
    Variable for behavioral analysis. xC2:mapToVar cites a
    value for uid which occurs on an Variable (since it is
    not clear if Variable names are unique enough) -->

<!ELEMENT Parameter EMPTY>
<!ATTLIST Parameter
          name       CDATA      #REQUIRED
          type       CDATA      #REQUIRED
          role     ( in | out | inout )  'in'
          xC2:mapToVar IDREF  #IMPLIED >
<!-- COMPONENT
    The Component tag is a named instance that can point
    to one (or more) ComponentTypes it supports. It can
    expose these functions over one (or more) named
    Ports.

    NOTE: C2 LINK SETS
```

```
      C2 data structures traditionally put all of a brick's
      connections to other bricks on this one, but we have
      chosen to entirely externalize this responsibility to the
      Link tags, leaving it as a simple computation over the
      hyperweb to compute sets such as “aboveCompLinks”.
      -->

<!ELEMENT Component        (Port*,Supports*)>
<!ATTLIST Component  name        CDATA      #REQUIR>

<!-- CONNECTOR
      The Connector tag is a named instance that can point
      to one (or more) ConnectorTypes it supports. It can
      expose these functions over one (or more) named
      Ports. -->

<!ELEMENT Connector (Port*,Supports*)>
<!ATTLIST Connector name CDATA      #REQUIRED >

<!-- PORT
      The Port tag is a qualifier for naming Link endpoints
      within a given Component or Connector.

      NOTE: C2 USAGE
      C2-style architectures will have to explicitly add <Port
      type=”Top”> and <Port type=”Bottom”> to every single
      Component and Connector, and also specify #Top and
      #Bottom on every link. -->

<!ELEMENT Port  EMPTY>
<!ATTLIST Port    type   CDATA           #REQUIRED >

<!-- SUPPORTS
      The Supports tag specifies type-compatibility for
      Component and Connector. However, the downside is
      that XML verification alone can not enforce type-
      coherency (i.e. that a Component's Supports child can
      only link to ComponentTypes rather than
      ConnectorTypes).

      The “type” attribute must be a unique string occuring in
      some other ComponentType or ConnectorType's
      “name” attribute. -->

<!ELEMENT Supports EMPTY>
<!ATTLIST Supports type     CDATA      #REQUIRED >

<!-- xC2:BEHAVIOR
      The following xC2:prefixes tags all support C2, in
      particular DRADEL, in specifying component behavior.
      It also adds mapToOper and mapToVar attributes to
      Method and Parameter, respectively, to link behavioral
      specifications to interfaces.

      The xC2:Behavior tag has only one xADL attribute,
      name, which is implied as the name of the enclosing
      ComponentType. Its children map onto C2SADEL
      specifications for state variables, invariants, and
      operations. -->

<!ELEMENT xC2:Behavior        (xC2:State,
                               xC2:Invariant,
                               xC2:Operation*)>
<!ATTLIST xC2:Behavior  name  CDATA          #IMPLIED>

<!ELEMENT xC2:State (xC2:Variable | xC2:Function)* >

<!ELEMENT xC2:Variable  EMPTY>
<!ATTLIST xC2:Variable
            name      CDATA      #REQUIRED
            type      CDATA      #REQUIRED
            uid     CDATA        #REQUIRED  >
<!-- xC2:FUNCTION
      From and To values are supposed to be types, either
      language built-ins (Integer, etc.) or custom types -->

<!ELEMENT xC2:Function        EMPTY>
<!ATTLIST xC2:Function
            name      CDATA      #REQUIRED
            from      CDATA      #REQUIRED
            to        CDATA      #REQUIRED  >
<!ELEMENT xC2:Invariant       (xC2:Expression)*>

<!-- xC2:EXPRESSION
      Expression is an anonymous grouping tag to bracket
      unparsed expression text from the original C2SADEL
      spec.
      E.g.  <xC2:Expression>  x  \eqgreater  y  </
      xC2:Expression> -->

<!ELEMENT                     xC2:Expression   ANY>

<!-- xC2:OPERATION
      Operation is a named, uniquely identified grouping tag
      to bracket related information from the original
      C2SADEL spec. In particular, it may include at most
      one Let, Pre, and Post sections. -->

<!ELEMENT xC2:Operation
            (xC2:Let?, xC2:Pre?, xC2:Post?)>
<!ATTLIST  xC2:Operation
            name   CDATA         #REQUIRED
            direction  (provide  | require ) #REQUIRED
            uid     CDATA        #REQUIRED >
<!ELEMENT xC2:Let             (xC2:Variable)*>
<!ELEMENT xC2:Pre             (xC2:Expression)>
<!ELEMENT xC2:Post            (xC2:Expression)>
```