UNIVERSITY OF CALIFORNIA,
IRVINE

Supporting Stakeholder-driven, Multi-view Software Architecture Modeling

DISSERTATION

submitted in partial satisfaction
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Eric Matthew Dashofy

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor André van der Hoek
Professor David Redmiles

2007

Portions of Chapter 4 adapted from “A Comprehensive Approach for the Development of Modular Software Architecture Description Languages,” in ACM Transactions on Software Engineering (TOSEM), 14(2) (April, 2005) © ACM, 2005.
http://doi.acm.org/10.1145/1061254.1061258
Used with permission under Section 2.5 “Rights Retained by Authors”
of the ACM Copyright Policy

All other content Copyright © 2007 Eric Matthew Dashofy

The dissertation of Eric M. Dashofy
is approved and is acceptable in quality and form
for publication on microfilm and digital formats:

_____

_____

_____

Committee Chair

University of California, Irvine
2007

Table of Contents

List of Figures

List of Tables

Acknowledgements

I have omitted the formal dedication page on this dissertation, because it is impossible to fit everything I want to say on a single page. I am grateful for so much, to so many, and this is for all of you.

This is for my family, who taught me how to be who I am. This is for Mom and Dad, Keith, Grandma and Grandpa Dashofy, Grandma and Grandpa Nichols, Uncle Tim, Aunt Chris and Uncle Casey, Aunt Mary, Michael and Mary, Katie and Jennifer, Jacob and Christian. I wish you could all be here for this, and I love you all very much.

This is for Alicia, who came into my life when I thought I had everything figured out, and showed me so much more. I love you, too.

This is for Dick Taylor, who took a chance on an undergrad he'd never met eleven years ago, and gave me more opportunity, freedom, and trust than I could have ever imagined. Without him there is no way that I would have gone to graduate school at all, let alone finished a Ph.D. Because of him, I've gotten to travel the world, meet and learn from the best in our field, teach college, and participate in an epic courtroom battle to rival the best episodes of Law and Order. I'd do it all again, every bit.

This is for Neno Medvidovic, who took me in as an undergrad and shepherded me through the process of learning how to do research, coalesce ideas, write papers, and get them published.

This is for André van der Hoek, who pulled me into 136 against my will so long ago, and who allowed me to travel with him on the weird and wonderful journey through academia. I am far better for it.

This is for David Redmiles, who manages to keep me from being oblivious to a much bigger world, and who keeps me honest.

This is for Vince Belusko, Hector Gallegos, Scott Moore, Monica Scheetz, Chuck Barquist, Nicole Smith, Nora Allende, Judy Wilson, Rolando Monroy, and all the rest of the MoFos. You guys are the Yankees. You put a bat in my hands and told me to swing away. If I ever get the chance to work with a team half this good again, I will be lucky.

This is for John Yasuda, who was my friend even longer than he was my boss. It is not fair that he couldn't be here for this. My daruma has two eyes now. As well it should be.

This is for Rich Everman, Eric Raith, Joe and Lewis Chiang, Scott Hendrickson, Gabriel Knoy, Jigar Kotak, Chris Shen, Katherine Wigan, Matt Critchlow, Paul Duffy, Derrek Gabagat, Ann Williams, and the rest of the ANTs, both official and in spirit,

past and future. It was too much fun to be work. The carpet will always be blue under your feet.

This is for Christopher Rhodes and Sunil Kumar, who taught me the secrets of how to code for real, and who let me break the build and taught me how to fix it.

This is for Hazel Asuncion, Alex Baker, Debi Brodbeck, Ping Chen, Justin Erenkrantz, Joe Feise, Roy Fielding, John Georgas, Michael Gorlick, Scott Hendrickson (again!), Art Hitomi, Chris Jensen, Yuzo Kanomata, Rohit Khare, Chris Lueer, Roshni Malani, Emily Navarro, Kari Nies, Eugen Nistor, Jie Ren, Anita Sarma, Sundararajan Sowrirajan, Girish Suryanarayana, Chris van der Westhuizen, Jim Whitehead, and all the other students and staff I've worked with. You guys make it really hard to be a hermit, and it has been a privilege working with each of you.

This is for Haig Krikorian, Michael Marich, Peter Suk, Bill Wood, Peter Shames, Nicolas Rouquette, and all the people cited in this dissertation that not only believed in this work, but supported it, took it and used it and made it better. Without you, there really isn't any point.

This is for all the ISR and ICS staff—Kiana, Nancy, Laura, Susan, Sabina, Diane, Carol, and everyone who came before. You make hard things easy, and complicated things simple.

This is for all the ARCS ladies, especially Mary Lou Furnas, and Michael and Julie Penley. Your generosity has been astounding and I would not be here without you. You changed my life in innumerable ways, and while I can't pay you back, I will do my best to pay it forward.

I would finally like to thank and acknowledge the funding agencies and companies who have provided grants to make this research possible, namely the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation (NSF), The Boeing Company, NASA's Jet Propulsion Laboratory (JPL), and International Business Machines Corporation (IBM). I would additionally like to thank the ARCS Foundation, Inc., Orange County Chapter, and the Graduate Assistance in Areas of National Need (GAANN) program for scholarships and fellowships that supported me.

Curriculum Vitae

Eric Matthew Dashofy

Education

Ph.D. in Information and Computer Science; adviser Richard N. Taylor. University of California, Irvine, 2007.
M.S. in Information and Computer Science; focus on Software Engineering. University of California, Irvine, 2001.
B.S. in Information and Computer Science *Magna Cum Laude*; focus on Software Engineering. University of California, Irvine, 1999.

Awards and Honors

National Merit Scholar, 1995
Scholars Circle Award Winner, 1995
UCI Regents Scholar, 1995-1999
Campuswide Honors Program, 1995-1999
ICS Honors Program, 1997-1999
Member, Sigma Xi and Phi Beta Kappa Honor Societies
Excellence in Undergraduate Research Award, 1999
Recipient, Graduate Assistance in Areas of National Need (GAANN) Fellowship, 1999-2003
Division of Undergraduate Education Committee on Teaching "Promise as a Future Faculty Member" Award, 2002
Recipient, Achievement Rewards for College Scientists (ARCS) Scholarship, 2002-2004

Refereed Journal Publications

[J-1] "The Role of Middleware in Architecture-Based Software Development." Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, Vol. 13 no. 4, August, 2003, pp. 367-393.

[J-2] "A Comprehensive Approach for the Development of Modular Software Architecture Description Languages." Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2), April 2005, pp. 199-245.

[J-3] "Architecture-Centric Development: A Different Approach to Software Engineering." John C. Georgas, Eric M. Dashofy, Richard N. Taylor. ACM Crossroads, 12(4), Summer 2006, online edition. See

http://doi.acm.org/10.1145/1144359.1144365.

[J-4] "Moving Architectural Description from Under the Technology Lamppost." Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. *Information and Software Technology*, 49(1), January 2007, p12-31.

Refereed Conference and Workshop Publications

[RC-1] "Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures" Eric M. Dashofy, Nenad Medvidovic and Richard N. Taylor. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999, pp. 3-12. Acceptance rate: 49/260 (19%).

[RC-2] "Issues in Generating Data Bindings for an XML Schema-Based Language." Eric M. Dashofy. In *Proceedings of the Workshop on XML Technologies and Software Engineering (XSE2001)*, Toronto, ONT, Canada, May 2001.

[RC-3] "A Highly-Extensible, XML-Based Architecture Description Language." Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. In *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, Amsterdam, Netherlands, August 2001, pp. 103-112. Acceptance rate: 18/47 (38%).

[RC-4] "Representing Product Family Architectures in an Extensible Architecture Description Language." Eric M. Dashofy and André van der Hoek. In *Proceedings of the International Workshop on Product Family Engineering (PFE-4)*, Bilbao, Spain, October 2001, pp. 330-341.

[RC-5] "An Infrastructure for the Rapid Development of XML-based Architecture Description Languages." Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. In *Proceedings of the 24th International Conference on Software Engineering (ICSE2002)*, Orlando, Florida, May 2002, pp. 266-276. Acceptance rate: 45/303 (15%).

[RC-6] "An Approach for Tracing and Understanding Asynchronous Architectures." Scott A. Hendrickson, Eric M. Dashofy, and Richard. N. Taylor. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, Montreal, Quebec, Canada, October 6-10, 2003, pp. 318-322. Acceptance Rate: 41/170 (24%).

[RC-7] "An (Architecture-centric) Approach for Tracing, Organizing, and Understanding Events in Event-based Software Architectures." Scott A. Hendrickson, Eric M. Dashofy, and Richard. N. Taylor. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005)*, St. Louis, Missouri, USA, May 15-16, 2005, pp. 227-236.

Teaching Activities

Instructor: ICS 123 (Software Architecture, Distributed Systems, and Interoperability, undergraduate level, 48 students); adapted curriculum, lectured, designed and graded homework and exams.
Teaching assistant/guest lecturer: ICS 221 (Intro to Software Engineering, graduate level).
Teaching assistant: ICS 52 (Intro to Software Engineering, undergraduate level).
Guest lecturer/class consultant: ICS 228 (Software Environments, graduate level).
Guest lecturer/class consultant: Software Architectures (graduate level), University of Southern California.
Guest lecturer/class consultant: Software Architectures (graduate level), California State University, Fullerton.
Guest lecturer: User Interfaces and Software Engineering (graduate level).

Service Activities

External Reviewer for:

International Conference on Software Engineering (ICSE)
Foundations of Software Engineering (FSE)
IEEE Transactions on Software Engineering
FASE Conference
Journal of Information and Software Technology (IST)
Journal of Software Practice and Experience (SP&E)
ETRI Journal
Wiley Press Books
IEEE Software
IEE Proceedings on Software
Addison-Wesley Books

Support Activities, Foundations of Software Engineering (FSE) 2004.

Volunteer, *Ask A Scientist Night*, 2002-2003.

Abstract of the Dissertation

Supporting Stakeholder-driven, Multi-view Software Architecture Modeling

By

Eric Matthew Dashofy

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2007

Richard N. Taylor, Chair

A software system's architecture can be seen as the set of principal design decisions about that system. These design decisions can be captured in architectural models. Existing approaches to architecture modeling support general concerns well, but are difficult or impossible to adapt to new concerns or domains. This research examines how to support stakeholder-driven architecture modeling: modeling in a world where the system's stakeholders materially participate in deciding what kinds of design decisions will be modeled and at what level of detail, as well as how they will be visualized, analyzed, and applied in other development activities. Contributions include a highly-extensible language (xADL) and environment (ArchStudio) for stakeholder-driven architecture modeling, a demonstration that many architectural concerns can be modularized within this environment, and a set of design principles that guide and inform the construction of the environment. Application and use of the environment in a variety of domains and projects demonstrates its effectiveness, and evidence indicates that the amount of effort needed to develop extensions to the environment scales with the conceptual complexity of those extensions.

1 Introduction

How to design successful large, complex, software-intensive systems is an unsolved problem. Although the situation has improved somewhat in the last 10 years, failure statistics for software-intensive systems remain alarmingly high. The Standish Group's CHAOS report [167, 168] indicates that 15% of such systems fail outright, and more than 50% of projects are "challenged"—being significantly over budget or past deadline. Additionally, only 52% of required features make it into released products.

The specific causes for these failures can be debated, but we can draw some conclusions from this data. Errors or problems introduced in early development activities, such as conceptualization, requirements capture, and design cost orders of magnitude more than errors introduced in late phases of development such as implementation and testing [150]. The magnitude of these failures suggests that the problems experienced are not small, easily correctable faults introduced late in development, but flaws introduced early. This implies that significant progress in reducing software project failures might be made by improving the earliest development activities, such as requirements and design.

Although improvements in both requirements and design activities have the potential to greatly reduce software costs and risks, this research focuses on design activities. During software design, design decisions are made affecting various aspects of the system. Some of these are high-level decisions or guidelines that affect the system as a whole; some are low-level decisions that affect the construction of only a single software element.

These design decisions do not comprise a software system's implementation—ideally, they characterize the implementation. Fundamentally, design decisions are an abstraction of a system's implementation—eliding lower-level details and focusing on narrow aspects of the development problem. Documents and other artifacts recording these design decisions are *models* of the software system—concrete abstractions of the system implementation.

Developing and maintaining system models is a costly, time-consuming activity. In order to make modeling worthwhile, the models themselves must provide more value than they cost. Models' value can come from a number of sources—finding errors early when the cost to fix them is relatively low, increasing understanding of the system among stakeholders to reduce confusion, and so on. One way to maximize the value of modeling is to focus modeling efforts on the most important parts—the principal aspects—of a system's design. These principal aspects of a design can be termed a system's *architecture*.

Focusing on principal—that is, architectural—design decisions helps to identify what should be modeled. Beyond this, however, modeling introduces a set of additional challenges: deciding how design decisions should be modeled, at what level of detail, how to visualize the models, how to keep them consistent, and so on. Answering these challenges is the focus of this research. While this effort is certainly not the first to attempt to address these software architecture modeling challenges, it differs from previous modeling efforts in several critical aspects.

One of the key deficiencies of previous modeling approaches is that they are, for the most part, *capability-driven*. That is, the value provided by an approach is based on

its ability to model a particular aspect (or set of aspects) of a system, as chosen by the developers of the modeling approach. The relative importance of various kinds of design decisions, and at what level of detail they can be specified, is also chosen by the approach developers. This conflicts with the natural expectation that the people involved in the development and use of the system should choose what is important in their design, not just the providers of modeling approaches. A partial remedy is to select multiple capability-driven solutions that roughly match the needs of a project, but this introduces further problems. It still leaves gaps between what is needed and what is provided—some concerns might be modeled at too low a level of detail, while others might be modeled too abstractly, certain concerns might not be modeled at all, and it is extraordinarily difficult to define or understand consistency relationships between the various approaches.

Beyond this, adapting existing approaches to meet a particular project's modeling needs is often infeasible. This follows from the capability-driven nature of these approaches: it is not expected that users of the approach will be the ones to define what can be modeled. As such, any adaptability mechanisms provided by existing approaches tend to be limited—in the best cases allowing specializations of existing capabilities, but rarely offering the ability to integrate entirely new capabilities.

This research involves an approach to architecture modeling that takes its cue from developers and others who have a stake in the successful construction and use of the target system. This approach is *stakeholder-centric*, rather than capability-centric. In this approach, system stakeholders play a much larger role in influencing how modeling is done, and how related concerns (e.g., visualization, consistency checking) are

addressed. Here, stakeholders denote any party or organization that has an interest, or stake, in the successful outcome of system development. In particular, the stakeholders most interested in a system's architecture will be architects and other developers, particularly when an architecture includes many design decisions that relate to technical concerns. However, not all architectural concerns are exclusively technical—some concerns such as scheduling, task partitioning, the relationships of products in a product-line, and so on may be managerial or business concerns. Domain-specific concerns might be of interest to domain experts that are not necessarily software or system developers. These stakeholders will not be directly involved in the design of modeling environments, but may provide substantial input as to what kind of information should be captured and how it is visualized; other individuals with software development experience may act as their proxies for adapting the modeling environment to capture their concerns. Stakeholder-driven design is an increasingly prominent trend in software-development. Bannon [13], for example, discusses the general move in the human-computer interaction community toward increased user involvement, where users move from being 'objects of study' to active agents within the design process.

1.1 Research Questions and Hypotheses

Moving from a capability-centric to a stakeholder-centric modeling approach means rethinking how notations, tools, and techniques are constructed. Primarily, it means providing novel mechanisms by which project stakeholders can create new modeling capabilities and adapt existing capabilities to their own needs. All of this together comprises an *environment* for stakeholder-centric architecture modeling. The

primary research questions raised, then, surround this environment. Can such an environment be constructed at all? What aspects of the environment make it suitable for such adaptation and customization? What design decisions should guide its construction? How much effort is required to adapt the environment to support a new or extended capability?

These research questions can be reformulated into testable hypotheses, which form the foundation of this dissertation. These hypotheses are:

H1. An environment can be constructed that supports stakeholder-driven multi-view software architecture modeling, addressing modeling, visualization, and consistency checking;

H2. Within such an environment, architectural concerns can be supported by interdependent, reusable modules corresponding to concerns that can be composed into modeling support for a particular project or need;

H3. These capabilities can be enabled by employing a particular set of design principles (which are identified) in the construction of the environment;

H4. The size of the support modules will be commensurate with a level of development effort suitable to a single developer, generally less than 5,000 lines of code per concern.

These hypotheses are elaborated upon and evaluated in the remainder of this dissertation. The first hypothesis, **H1**, is an existence hypothesis. This is validated through the construction and implementation of a modular notation (xADL) and extensible environment (ArchStudio) for stakeholder-centric, multi-view software architecture modeling. These show that such an environment can be constructed. The

second hypothesis, **H2**, relates to a critical aspect of the environment, namely the ability to modularize modeling support for architectural concerns. Such modularization allows modeling support for a concern to be reused in multiple projects or domains, and also facilitates divergent extension of the environment—the ability for different groups to customize the environment in potentially conflicting ways. Previous, monolithic modeling environments were deficient in these respects, limiting their customizability. The third hypothesis, **H3**, calls out the separation of essence from accident in the construction of the environment. The constructed environment shows that a solution to the problem of stakeholder-centric modeling exists, but does not necessarily reveal what aspects of that environment are essential to the solution and which are simply details of one particular implementation. This separation is achieved through long experience constructing and refining the environment over several years and hundreds of releases, and vetting the environment in diverse contexts. The final hypothesis, **H4**, is a quantifiable hypothesis about the effectiveness of the environment: if extensions to the environment are prohibitively costly to develop, then there is little reason to use the environment at all (since developing solutions from scratch would be of comparable difficulty).

These hypotheses are further evaluated through the application and use of the environment in many projects, both by its developers and others. This experience has refined the environment and motivated the expansion of its capabilities, as well as provided valuable data about the effectiveness of the environment in solving real-world modeling problems. Several primary experiences are described in this dissertation: the application of the environment to the modeling of different aspects of a software-

defined radio (SDR), the use of the environment to model and simulate the software architecture of a U.S. Air Force AWACS aircraft, the integration of the environment into a space mission data systems development process, and the adaptation of the environment to support space data system modeling. Additionally, eleven other projects are described that make use of some or all of the environment's features and extensibility mechanisms.

The remainder of this dissertation is organized as follows. Chapter 2 discusses background material and related work. Chapter 3 provides precise definitions for important terms used throughout the text. Chapter 4 provides a detailed description of this work's approach, specifically focusing on the technical details of the xADL architecture description language and ArchStudio environment. Chapter 5 catalogs the design principles that guide xADL and ArchStudio's development, providing rationale for each principle as well as information about how xADL and ArchStudio reflect that principle. Chapter 6 describes further evaluation activities in which I or others have applied this approach in real-world contexts, or to attack research problems related to software architecture. Finally, Chapter 7 summarizes the work and discusses some future directions.

2 Background

The need for a discipline of software architecture is intimately tied to the "software crisis" [48]—the complexity of software-intensive systems outpacing our collective ability to deal with that complexity. The earliest efforts to define and attack this problem resulted in McIlroy's seminal 1968 paper [104], which laid out the need for looking at software above the level of lines-of-code or simple modules. Later, DeRemer and Kron made a distinction between what they termed "programming in the large" and "programming in the small" [45]. Their notion of programming-in-the-large made concepts of abstraction, modularity, and structured programming concrete by introducing a new way of thinking about system construction: as a composition of interdependent modules, configured by a "module interconnection language" (MIL). MILs were the first languages to formally express the structure of system modules and certainly influenced later, richer architecture description notations.

The modern conception of software architecture, however, has its roots in Perry and Wolf's seminal paper of 1992 [129]. Perry and Wolf characterized architecture broadly, in terms of three key concepts: elements, form, and rationale. Architectural *elements* include processing, data, and connective elements. *Form* governs how those elements may be arranged and configured. *Rationale* captures the reasoning behind the design decisions that are expressed in the selection of elements and a form. They also discuss the concept of an *architectural style*, a set of design decisions or constraints on an architecture focused on one or more salient properties. They note that there is no hard-and-fast dividing line between an architecture and an architectural style, although how the decisions and constraints are used (and reused) can be an indicator.

2.1 First-Generation Architecture Description Languages

Following the publication of Perry and Wolf's paper and with funding from DARPA programs such as Arcadia [94] and EDCS [162], the mid-1990s saw a surge of academic research on software architecture. This research was characterized by a variety of projects from different institutions, each of which explored a different set of concerns from an architectural perspective. Each project resulted in the development of a notation and tools for capturing the architecture of the software system, with special support for the particular concern under consideration. These notations were termed 'architecture description languages,' or ADLs.

While still falling within the domain of Perry and Wolf's characterization of architecture as elements, form, and rationale, first-generation ADLs focused on a specific subset of this domain. Specifically, these ADLs placed a heavy emphasis on architectural *structure*—where the elements under consideration were primarily discrete software components (the loci of computation) and connectors (the loci of communication). A related element is the architectural *interface*—an explicit specification of the services provided or required by a given component or connector. The *form* of these elements is the particular topology and interconnections among them, usually expressed as links between interfaces. Few of these ADLs provided explicit support for *rationale* capture. The trend of structural modeling was so strong in these early ADLs that, in their survey of first-generation ADLs, Medvidovic and Taylor [106] used the ability to capture structure (components, connectors, interfaces, and links) as the litmus test for whether a notation was an ADL or not.

Representative examples of this first generation of ADLs included Darwin, Wright, and Rapide. These example projects are summarized here, followed by commentary on their contributions as a group and their relationship to stakeholder-driven modeling.

2.1.1 Darwin

Darwin [103], from Imperial College in London, allows users to capture the architecture of distributed systems consisting of discrete components that communicate via well-defined pathways using diverse interaction mechanisms. As a language, Darwin has a canonical textual representation in which components and their interconnections are described. There is an associated graphical representation that depicts Darwin configurations in a more accessible but less precise way.

Like the other first-generation ADLs, Darwin focuses primarily on the capture of architectural structure. In Darwin, systems are modeled as a set of interconnected components. There is no notion of explicit software connectors in Darwin, but a component that facilitates interactions could be interpreted as a connector. Darwin components expose a set of provided and required services. Services in Darwin correspond to the notion of provided and required architectural interfaces. Configurations are specified by a set of bindings between interfaces. Darwin also has support for hierarchical composition—that is, components that have internal structures also consisting of components, services, and bindings. Through these constructs, Darwin provides a way to rigorously specify the structure of a system.

```
component WebServer{
    provide httpService;
}
```

```
Component WebClient{
    require httpService;
}

component WebApplication{
inst
    S: WebServer;
    C: WebClient;
bind
    C.httpService -- S.httpService;
}
```

Figure 1. A very simple Web application in the Darwin language.

Figure 1 shows a very simple one-client, one-server system as expressed in Darwin's textual form. In it, a Web server component, providing a single service (httpAccess), is connected to a Web client component with a single required service.

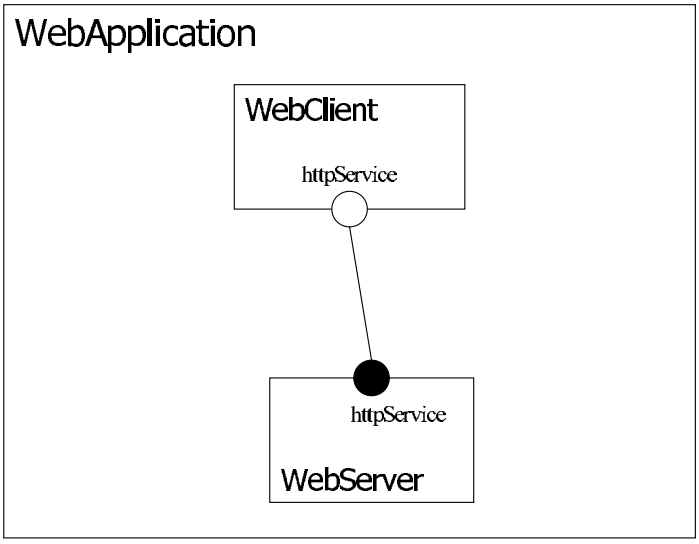


Figure 2. The simple Web application in Darwin's graphical format.

Figure 2 shows the same system, only in Darwin's graphical format. Provided services/interfaces are represented as filled circles and required services/interfaces are represented as empty circles.

One of the most interesting and distinctive aspects of Darwin is its inclusion of programming-language-like looping and conditional constructs. Most ADLs require the

explicit enumeration of each element in an architecture, and Darwin indeed supports this (as in the above example). However, elements can also be repeated and parameterized. For example, the above Web application can be adapted into a multi-client version using these constructs.

```
component WebServer{
    provide httpService;
}

component WebClient{
    require httpService;
}

component WebApplication(int numClients){
    inst S: WebServer;
    array C[numClients]: WebClient;
    forall k:0..numClients-1{
        inst C[k] @ k;
        bind C[k].httpService -- S.httpService;
    }
}
```

Figure 3. A more complex Web application with multiple clients.

Figure 3 shows the textual form of a multi-client web application with *numClients* clients. The description can be parameterized with different values for *numClients* to describe different variants of the application with different numbers of clients.

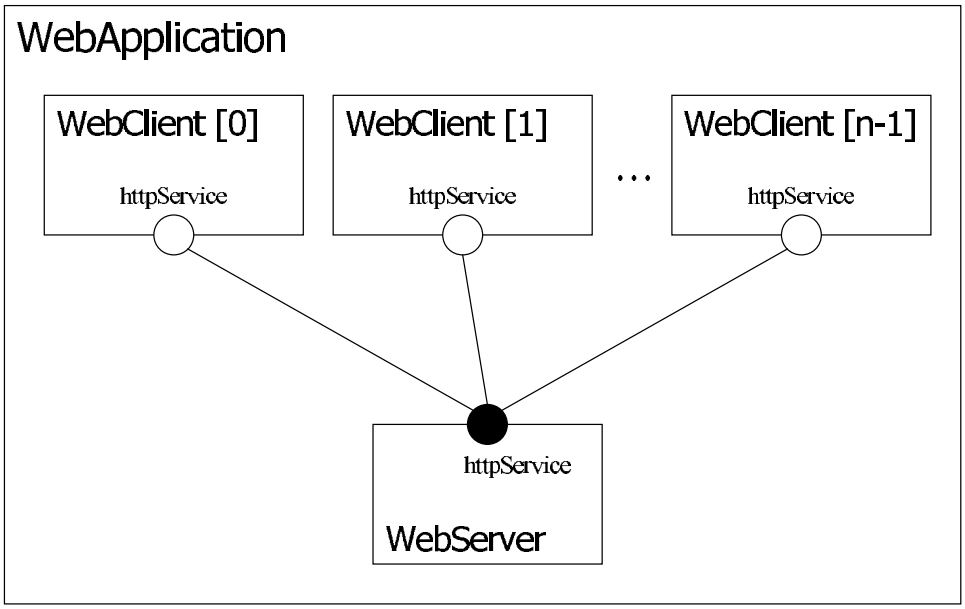


Figure 4. A graphical depiction of the multi-client Web application.

Figure 4 shows how this multi-client application might be depicted graphically. Note here that the graphical description does not rigorously capture all the detail expressed in the text; this is an example where a notation's graphical visualization is less precise than its canonical (in this case textual) visualization.

From an architectural modeling perspective, Darwin provides a straightforward, rigorous notation for capturing architectural structure. However, it provides little else, and is really a starting point for more complex architecture description languages such as Koala, a derivative of Darwin that is described later.

2.1.2 Wright

The Wright language [6, 7] is focused on checking component interactions through their interfaces. Interfaces in Wright are specified using a notation derived from the Communicating Sequential Processes (CSP) formalism [76]. Wright specifications can be translated into CSP, and then analyzed with a CSP analysis tool like FDR [57].

These analyses can determine, for example, if connected interfaces are compatible, and whether the system will deadlock. Additionally, Wright has the ability to specify stylistic constraints as predicates over instances. Wright represents an increased amount of formalism in architecture modeling, using the mathematical verifiability of CSP to draw non-obvious conclusions (such as the ability of a system to deadlock) from an architecture description.

```
Component WebServer
  Port httpService (behavior specification)
  Computation (behavior specification)

Component WebClient
  Port httpService (behavior specification)
  Computation (behavior specification)

Connector Call
  Role Caller = call → return → Caller[]§
  Role Callee = call → return → Callee[]§
          Caller.call → Callee.call → Glue
  Glue = []Callee.return → Caller.return → Glue
          []§

Configuration WebApplication
  Instances
    S : WebServer
    C : WebClient
    ClientToServer : Call

  Attachments
    C.httpService as ClientToServer.Caller
    S.httpService as ClientToServer.Callee

End WebApplication.
```

Figure 5. A partial model of a simple Web application in Wright.

Figure 5 shows a (partial) specification of the simple one-client Web application in Wright, in which the client calls the server through a connector that behaves as a procedure call (as an HTTP request/response pair [56] does). For simplicity, only the behavioral specification of the connector itself is fully elaborated.

As is obvious from this example, Wright specifications have much in common with Darwin specifications. Wright captures architectural structure in a manner similar to Darwin. Primary elements are components and connectors (unlike Darwin which specifies only components). Architectural interfaces are also included in Wright, called *ports* on components and *roles* on connectors. Unlike Darwin, components, connectors, and interfaces (ports and roles) are all annotated with CSP-based descriptions of their behavior; these are what makes Wright specifications analyzable.

While these formal specifications convey substantial analytical power, they come at a high price. The specifications themselves are difficult to understand, write, and debug for someone not skilled in the CSP formalism. Additionally, the complexity of these specifications grows with the complexity of the components and systems they describe. For a system with hundreds of thousands or millions of lines of code, it would be infeasible to write complete behavioral specifications—specifications of partial behavior or behavioral aspects would have to be used instead. Additionally, there is no mechanism for translating Wright specifications directly to code, so coding errors or infidelity could lead to properties (such as deadlock freedom) that were verified in a Wright specification not being present in the actual, developed system.

2.1.3 Rapide

Rapide [102] is an architecture description language developed to specify and explore dynamic properties of systems composed of components that communicate using events. In Rapide, events are simply the result of an activity occurring in the architecture. Traditional procedure call-style interactions are expressed as a pair of events: one for the call and one for the return value.

The power of Rapide comes from its organization of events into Partially Ordered SETs, called POSETs. Rapide components work concurrently, emitting and responding to events. There are causal relationships between some events: for example, if a component receives event A and responds by emitting event B, then there is a causal relationship from A→B. Causal relationships between events A and B in Rapide are defined by (from the Rapide documentation):

- *A* and *B* are generated by the same process, or
- A process is triggered by *A* and then generated *B*, or
- A process generated *A* and then assigns to a variable *v*, another process reads *v* and then generates *B*, or
- *A* triggers a connection which generates *B* or
- *A* precedes *C* which precedes *B* (transitive closure).

As a program runs, its various components will generate a stream of events over time. Some of these events will be causally related (by one of the above relationships). Some of them may occur around the same time as other causally related events, but be unrelated.

Rapide provides two main capabilities of interest to architects. First, Rapide specifications can be simulated by the Rapide toolset. In the toolset, simulated components emit and respond to events as specified in their Rapide descriptions. In architectures where some events are causally independent of each other, the simulator runs non-deterministically: different runs of the simulator on the same architecture may produce different event traces (although the partial orders will still be intact). Simulator outputs can be run through the *dot* graph layout tool to generate hierarchical graph

depictions of each simulation run. Second, assertions can be made about event orderings that are checked by the simulator. If a violation of an assertion is found during a simulation run, then the violating trace is reported.

```
type Pump is interface
action in  On(), Off(), Activate(Cost : Dollars);
       out     Report(Amount : Gallons;
                      Cost : Dollars);
behavior
    Free : var Boolean := True;
    Reading, Limit : var Dollars := 0;
    action In_Use(), Done();
begin
   (?X : Dollars)(On ~ Activate(?X)) where $Free => Free := False;
                                                    Limit := ?X;
                                                    In_Use;;
   In_Use => Reading := $Limit; Done;;
     Off or Done  => Free := True; Report($Reading);;
```

Figure 6. A portion of the "gas station" example written by the Rapide developers

Figure 6 shows a portion of a simple Rapide application used to simulate the operation of a gas station, from the Rapide documentation. This portion describes the operation of a gas pump. Here, when the pump receives an 'on' event followed by an 'activate' event (accompanied by a number of dollars of gas to pump), several things happen. First, an internal state variable of the pump, 'Free,' changes to false. Another internal state variable, 'Limit' changes to the amount of gas to pump, and an internal event 'In_Use' is fired. When the pump receives an internal In_Use event, the internal 'Reading' variable is set to the Limit (simulating that all the gas has been pumped; intermediate states such as "not done yet" are not included in this version of the simulation). An internal event 'Done' is also fired. When the internal 'Done' event fires, or the pump is turned off externally (by an operator or customer object, for example), the pump is set back to a free state and an external 'Report' event is fired, indicating how much gas was actually pumped.

As with Wright, significant formal specification annotates traditional elements of architectural structure (components, interfaces, connections, and configurations of these). These formal specifications express the internal and external behavior of the various components, although in a manner completely different than Wright. Instead of formal verification, Rapide uses simulation as a heuristic technique to indicate behaviors and find constraint violations. Like Wright, the complexity and un-familiarity of the Rapide language can lead to a high learning curve and scaling problems when dealing with large applications.

2.1.4 Reflections on First-Generation ADLs

Darwin, Rapide, and Wright are examples that characterize the early development and proliferation of ADLs. They all share a common core of structural modeling—components, connectors, interfaces, and configurations. In fact, the syntax for expressing structure in each of these languages is eerily similar, despite the fact that the languages were developed by independent groups at different universities. Beyond this, however, each language includes features that go beyond simple structure. These include Darwin's ability to use loops and conditionals to create variant architectures, Wright's CSP specifications to describe behavior in terms of states, and Rapide's specifications to describe behavior and interactions in terms of events. In some cases, these features conflict (Wright and Rapide behavioral specifications attack a similar problem in different ways) but in other ways they are complementary (Darwin's looping constructs might make specifying complex architectures with many similar elements in Wright or Rapide easier).

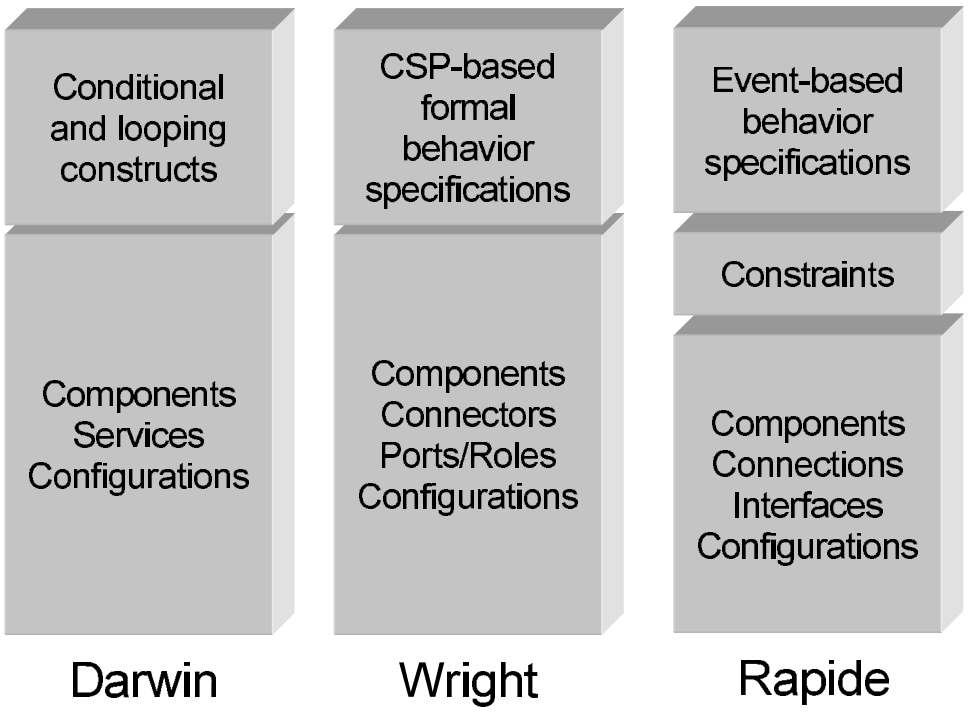


Figure 7. The 'stovepipes' of modeling created by first-generation ADLs.

Conceptually, the situation with first-generation ADLs is shown in Figure 7. Each possesses a common 'core' of structural modeling, with slightly different interpretations or names for the same concepts. The real modeling power of these ADLs comes from distinguishing features built atop these core features.

Lack of compatibility and interoperability among these ADLs limits their overall usefulness. Each ADL's syntax, semantics, and tool-set are divorced from all the others. None of these ADLs provides a mechanism for extending the language or the tool-set to incorporate or experiment with new features. If stakeholders want the benefits of more than one ADL, they must effectively model their system again and again. If they want to specialize their models to support additional concerns not well-supported by an existing ADL, they have no recourse. This situation is the genesis of this dissertation's research.

2.2 Acme—An Architecture Interchange Language

The interoperability problem among first-generation ADLs was identified in the 1990s. In 1995, researchers at Carnegie Mellon University began developing Acme [61], a notation for capturing software architecture that was more flexible than other first-generation ADLs. At its core, the notation captures common structural elements—*components*, *connectors*, interfaces (called *ports* and *roles*, as in Wright), configurations via links (called *attachments*) and hierarchical structure through a notion of internal *representations*. Acme's flexibility comes primarily in the form of *properties*—name-value pairs of strings that are associated with Acme elements.

One of the original intents of Acme was to serve as an *architectural interchange language*—a generic format that serves as a go-between for other ADLs. That is, specifications in languages like Wright, Rapide, and Darwin could be translated into Acme. Common concepts (components, connectors, and so on) would be translated into Acme syntax, while distinguishing features would be embedded in the properties. By creating translators between Acme and various other ADLs, architectural descriptions could be interchanged among them. This could allow, in theory, a visualization or analysis technique in one language's toolset to be applied to an architecture description developed in another.

```
System web_application = {
    Component web_client = { Port send-http-request; };
    Component web_server = { Port receive-http-request; };
    Connector http = { Roles { sender, receiver}};
    Attachments {
        web_client.send-http-request to http.sender;
        web_server.receive-http-request to http.receiver;
    }
}
```

Figure 8. A simple web application as specified in Acme.

Figure 8 shows the simple Web application in unadorned Acme. Syntactically and semantically, Acme without properties is very similar to Darwin. Adding properties allows additional data to be specified.

```
System web_application_2 = {
    Component web_client = {
        Port     send-http-request;
        Property vendor : Microsoft;
        Property product: Internet Explorer;
        Property version : 7.0sp1;
        Property supported-http-versions : 1.0, 1.1;
    };
    Component web_server = {
        Port receive-http-request;
        Property vendor : Apache;
        Property product : httpd;
        Property version : 2.0.48;
        Property supported-http-versions : 1.0, 1.1;
    };
    Connector http = {
        Roles { sender, receiver}
        Property http-version: 1.1;
    };
    Attachments {
        web_client.send-http-request to http.sender;
        web_server.receive-http-request to http.receiver;
    }
}
```

Figure 9. The same Acme description annotated with properties.

Figure 9 shows the same description, where several of the elements are annotated with additional properties. They increase the precision of the specification, capturing, for example, the precise origin and version of the various components. Properties to include, for example, Wright or Rapide behavioral specifications could also be included.

As an architectural interchange language, Acme had limited success. The vastly differing semantics of the target ADLs provided little hope that interesting analyses in, for example, Rapide, could be applied to specifications developed in Wright.

Furthermore, translations to and from target ADLs are necessarily lossy—Wright cannot encode Rapide information, and vice-versa.

In terms of extensibility and flexibility, Acme was an advance over contemporary ADLs. However, its use of uninterpreted string-based properties as the extensibility mechanism is a simple solution to a complex problem. There is no vocabulary or set of constraints for Acme properties, and any such support for a particular sub-language of Acme (encoded in property values) would have to be defined and maintained entirely in tools, or as a tacit agreement among Acme architecture description writers. Acme's potential is also constrained by the fact that its core elements (describing architectural structure) are fixed, and those core elements can only be decorated with properties. Acme provides little guidance to users as to how to support property-based extensions—how to visualize them, analyze them, and so on are tasks left up to the user. Complex architectural concepts—those going beyond simple structure—require the ability to define new, complex elements at the level of core elements. Accompanying visualizations and analyses must also be developed for these elements. Acme provides no significant guidance as to how to develop such extensions, and adding new core elements would require a substantial re-design of the language and tools commensurate with adding them to any other first-generation ADL.

The latest tool support for Acme is provided in an environment called AcmeStudio [149]. AcmeStudio is an integrated environment of tools provided as plug-ins to the Eclipse [51] development environment. AcmeStudio provides a graphical editor and constraint analysis capability for Acme. AcmeStudio exposes several plug-in extension points that can be used by the environment to call out to user-created

extensions. In this regard, AcmeStudio goes beyond many other environments in supporting stakeholder-centric extensibility. AcmeStudio's effectiveness in supporting modular extension, extensions to visualizations and analysis capabilities, and divergent extensions is difficult to assess.

2.3 Early XML-based ADLs

The development and rapid adoption of XML, the eXtensible Markup Language [25], prompted some groups to experiment with the use of XML as a basis for creating new architecture description languages. Two primary ADLs emerged from these experiments, ADML and xADL 1.0.

ADML, the Architecture Description Markup Language [151], is an Open Group standard for encoding architecture descriptions in XML. The design of the language is based on Acme, and contains similar primary constructs: components, connectors, ports, roles, attachments, representations, and so on. ADML also uses Acme-style properties to allow users to decorate these core elements. ADML does improve on Acme by adding a notion of types that is based on meta-properties. Each type defines a set of meta-properties that prescribe what properties can be used on an instance of that type. Beyond this, the primary advantages of ADML over Acme are conferred by XML: the availability of off-the-shelf parsers, validators, data binding generators, serializers, and so on. These tools make constructing support for the ADML notation easier, but offer little help in the areas of visualization and analysis. ADML was released with optimistic assumptions that the combination of standardization and the use of XML would motivate tool vendors to support ADML and for a community to

evolve around it. In retrospect, substantial tool support and community-based evolution of ADML failed to materialize.

xADL 1.0 [96] was another early effort at developing an XML-based architecture description language. xADL also contains a core set of elements that describe structure: components, connectors, links, and so on. Extensions to xADL 1.0 were done through the addition of elements and attributes from additional, external XML namespaces. The difficulty with incorporating these additions is that they must be manually mixed into the xADL 1.0 DTD. It is not possible, for example, to define the base xADL 1.0 concepts in one DTD and extend them in another. xADL 1.0's use of XML conferred the same benefits of off-the-shelf tool support as ADML. xADL 1.0 was supported by an environment of integrated tools called ArchStudio 2.0, discussed further in Section 5.1.

ADML and xADL 1.0's recognition of XML as the basis for developing new modeling notations was well-founded. XML and its associated tool support do indeed solve a number of practical issues and standardize things like parsing, serialization, data binding generation, and so on. Both provide some form of extensibility—properties and meta-properties in ADML, ersatz namespaces in xADL 1.0. ADML properties suffer from the same problems as Acme properties, although meta-properties offer some new opportunities for prescribing elements. xADL 1.0's inclusion of additional elements from other namespaces allows nearly arbitrary language extensions, but without accompanying principles about how to create or incorporate these extensions. Neither approach is accompanied by guidance as to how to support visualization and analysis with respect to extensions.

2.4 The Unified Modeling Language (UML)

Perhaps the most well-known and prevalent notation used for capturing software designs is the Unified Modeling Language (UML) [21]. It has achieved more mainstream support than any other notation for modeling software-intensive systems in recent memory. As its name implies, UML was derived through the unification of multiple influential modeling approaches: the Booch method [19], Rumbaugh's Object Modeling Technique (OMT) [144], Jacobson's Object-Oriented Software Engineering (OOSE) [88] method, Harel's statecharts [68], and various other sources.

UML's approach to modeling can be likened to a Swiss Army Knife—providing a number of distinct capabilities or tools together in a single package. In UML's case, these capabilities come in the form of UML *diagrams*—individual graphical notations for capturing different aspects of a software system. UML 2.0 includes thirteen such diagrams. Among these are the *class* diagram, which documents the classes in an object-oriented system and their relationships, the *state* diagram, which documents different application states in a form similar to finite-state automata, and the *sequence* diagram, which depicts how elements communicate with each other and exchange data over time.

Whether UML is an ADL, and how suitable it is for that purpose, has been the subject of extensive study and debate [3, 20, 64, 75, 77, 107, 138]. Early versions of UML did not have explicit support for modeling structural concerns like components and connectors in ways that were comparable to those used in first-generation ADLs. This led Medvidovic and Taylor, in their 1996 survey of ADLs, to conclude that UML was not an ADL at all. UML's acceptance as an ADL has evolved since that time,

however. First, the maturation of the field of software architecture is increasingly indicating that structure is not the only relevant architectural concern—principal design decisions address non-structural aspects of systems as well. In this broader light, even UML 1.x can be considered an ADL (albeit one deficient in the area of structural modeling), and Medvidovic and Taylor acknowledged this in a 10-year retrospective/extension of their original paper [108]. Second, UML itself has evolved. Specifically, UML 2.0's *component* diagram was significantly overhauled to better capture architectural structure using elements similar to those found in other ADLs such as Darwin.

Figure 10. A UML 2.0 component diagram depicting the structure of a simple Web application

Figure 10 shows such a UML 2.0 component diagram, depicting the structure of a simple three-component Web application. Components are depicted as boxes with labeled provided and required ports (interfaces) connecting them. Explicit connectors are not part of UML 2.0, although components can be specialized to describe connectors if needed.

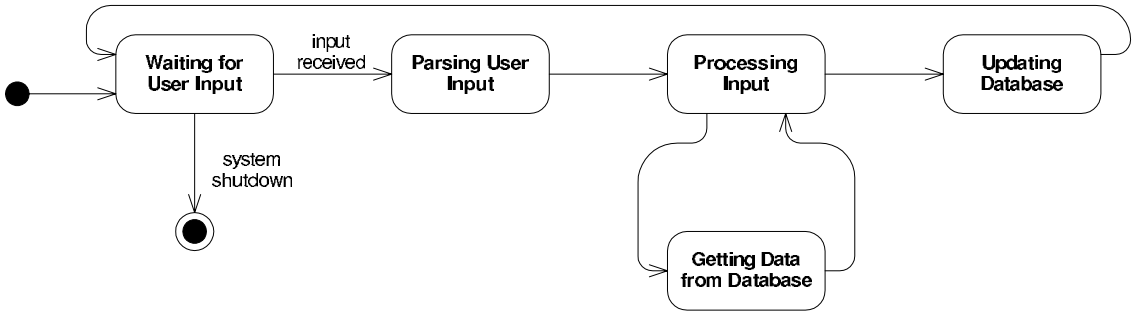


Figure 11. A UML 2.0 state diagram depicting the states of the Web application.

Figure 11 shows a UML 2.0 state diagram depicting the states and state transitions of the same application. Note that while this diagram purports to depict the same application, it has no direct correspondence to the elements in the component diagram.

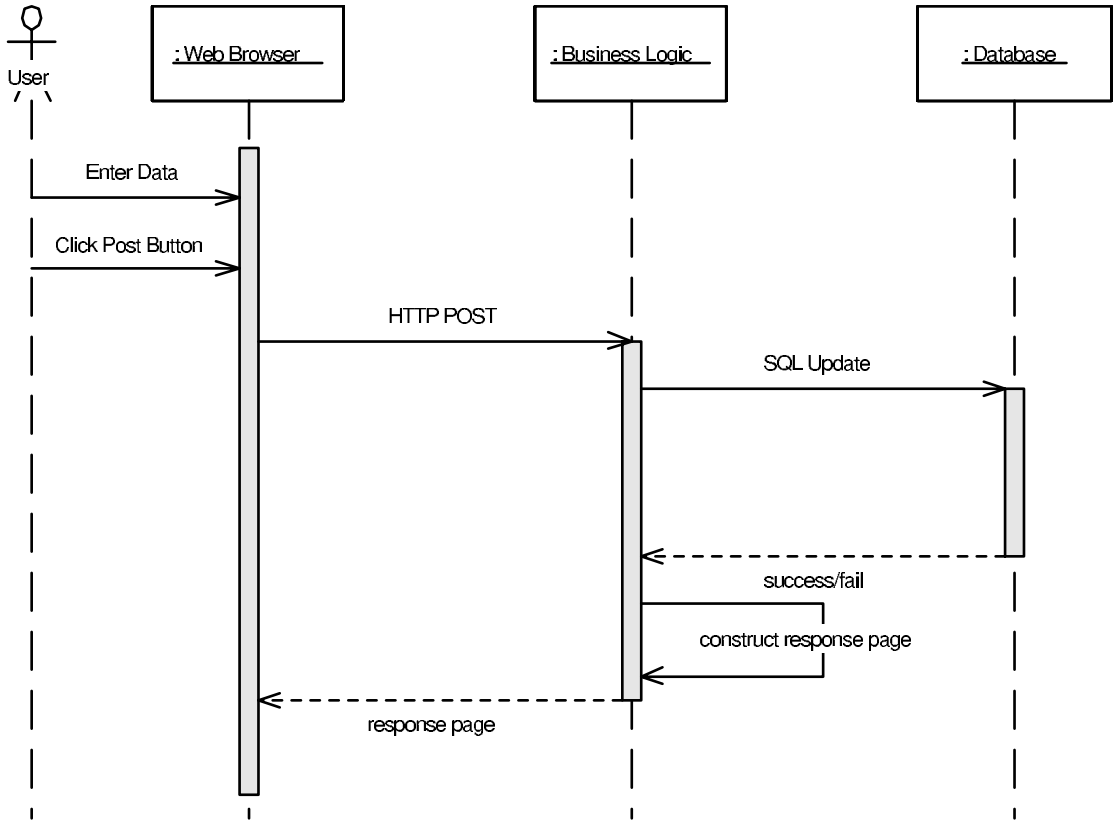


Figure 12. A UML 2.0 sequence diagram for the Web application.

Figure 12 shows a UML 2.0 sequence diagram describing the same application. Here, there is a more explicit connection to the component diagram; the communicating elements in the sequence diagram correspond to components in the component diagram.

Several aspects of UML permit user customization. UML is, to an extent, purposefully ambiguous. For example, while a class in a class diagram is ostensibly intended to represent a class in an object-oriented programming language, it may in fact be used to represent a code module or part of a data structure. Diagrams are also used as abstractions, eliding away some details and focusing on others. As such, they may not be intended to be a complete or comprehensive description of some part of a system. By employing somewhat ambiguous semantics, UML gives users more freedom to attach particular interpretations to diagrams and symbols that are favorable to their current project or domain.

UML also includes three built-in extension mechanisms that allow users to customize the notation. These are *tagged values*, *stereotypes*, and *constraints*. Tagged values are the same as Acme properties—they are arbitrary name-value pair properties that can decorate existing UML elements with additional information. Tagged values can be used to simply provide additional information to people reading the diagrams, or they can be interpreted by automated tools such as code generators or analyzers. Stereotypes are a mechanism by which subtypes or subclasses of individual elements can be defined. For example, UML's generic *class* element can be annotated with an <<interface>> stereotype or <<exception>> stereotype to indicate a special kind of class (here, an interface class or an exception class). Stereotypes can have associated tagged values that are implicitly applied to all elements bearing the stereotype. For

example, every association (link) that is stereotyped to <<httpConnection>> might be accompanied by a tagged value “protocol = ‘http’.” Constraints allow UML users to express constraints on UML diagrams, elements, and element relationships. These constraints are, like tagged values, expressed as decorations on individual elements and may be expressed in any constraint language convenient to the user, including natural language. However, UML is accompanied by OCL [175], a language specifically designed to express element and relationship constraints in a rigorous, machine-checkable manner. Tagged values, stereotypes, and constraints can be grouped together in a *UML profile*, a set of extensions that adapt UML for a particular purpose.

A final option exists for customizing UML—extension of UML through its metamodel. UML’s syntax is defined in a meta-language called MOF, the meta-object framework [122]. In it, UML classes and associations are used to express allowable elements and their relationships. Here, class diagrams are used in a manner similar to generic entity-relation [30] (ER) diagrams. UML cannot be completely interpreted based on the MOF alone, however. Additional external documents and books are needed to describe the semantics of the various elements as well as the graphical ‘look’ of each diagram. Because UML is defined in the MOF, UML’s MOF description can be changed or extended to add new diagram and element types to UML. This is useful when specializing existing diagrams and elements through stereotypes, tagged values, and constraints is insufficient to support a proposed extension. However, the resulting language is no longer UML, and existing UML tools will fail to recognize the MOF changes. For this reason, MOF-based extensions to UML are rare. SysML [159] is a prominent example of a MOF-based UML extension, in which new diagram types and

semantics are added to UML to improve its efficacy for system modeling. In the case of SysML, its developers (a group of large and influential companies and organizations [160]) held enough sway with tool vendors to convince them to adapt tools to support these new extensions. Nonetheless, difficulties in adapting tools were sufficient enough that the group has now planned an alternate strategy of developing a UML profile for SysML [125].

2.5 Second-Generation ADLs

Despite the modeling research provided by first-generation ADLs and broad capabilities of UML, not all modeling concerns are covered by these approaches. Research into architectural modeling has continued into the 2000s, with the creation of several new modeling approaches. These approaches differ from first-generation ADLs and UML in that they address more specialized concerns: those motivated by business goals, specific domains, or individual projects. Two representative examples of this second generation of ADLs are AADL and Koala.

2.5.1 AADL—The Architecture Analysis and Design Language

The Architecture Analysis and Design Language (AADL, formerly the Avionics Architecture Description Language) [55] is an ADL for specifying system architectures. While its historical name indicates that its initial purpose was for modeling avionics systems, the notation itself is not specifically bound to that domain—instead, it contains useful constructs and capabilities for modeling a wide variety of embedded and real-time systems such as automotive and medical systems. It is an outgrowth of the first-generation ADL MetaH developed by Honeywell [17] and is now developed

collaboratively by a group of industrial and academic organizations. AADL is a Society of Automotive Engineers (SAE) standard, and as such is guided by a larger, more open group than its predecessor MetaH.

AADL can describe the structure of a system as an assembly of components. It has special provisions for describing both hardware and software elements, and the allocation of software components to hardware. It can describe interfaces to those components for both the flow of control and data. It can also capture non-functional aspects of components (such as timing, safety, and reliability attributes). Syntactically, AADL is primarily a textual language, although a graphical visualization and a UML profile for it are under development. The syntax of the language is defined using BNF production rules.

The basic structural element in AADL is the component. AADL components are defined in two parts: a component type and a component implementation. A component type defines the interfaces to a component—how it will interact with the outside world. A component implementation is an instance of a particular component type. There may be many instances of the same component type. The component implementation defines the component's interior—its internal structure and construction. One additional element that affects components is a component's category. AADL defines a number of categories (or kinds) of components; these can be hardware (e.g., memory, device, processor, bus), software (e.g., data, subprogram, thread, thread group, process), or composite (e.g., system). The category of a component prescribes what kinds of properties can be specified about a component or component type. For example, a

thread may have a period and a deadline, whereas memory may have a read time, a write time, and a word size.

AADL is supported by an increasing base of tools, including a set of open-source plug-ins for the Eclipse [51] software development environment that provide editing support and import/export capabilities through the extensible markup language (XML) [25]. An additional set of plug-ins is available for analyzing various aspects of AADL specifications—for example, whether all the elements are connected appropriately, whether resource usage by the various components exceeds available resources, and whether end-to-end flow latencies exceed available time parameters.

```
data sensor_control_data
end sensor_control_data;

data sensor_data
end sensor_data;

bus local_bus_type
end local_bus_type;

bus implementation local_bus_type.pci
properties
  Transmission_Time => 30 ns;
  Allowed_Message_Size => 4 b;
end local_bus_type.pci;

system sensor_type
features
  network : requires bus access
            local_bus_type.pci;
  sensed  : out event data port sensor_data;
  control : in event data port sensor_control_data;
end sensor_type;

system implementation sensor_type.temperature
subcomponents
  the_sensor_processor :
              processor sensor_processor_type;
  the_sensor_process : process
              sensor_process_type.one_thread;
connections
  bus access network -> the_sensor_processor.network;
  event data port sensed ->
              the_sensor_process.sensed;
  event data port control ->
```

```
              the_sensor_process.control;
properties
  Actual_Processor_Binding => reference
              the_sensor_processor applies to
              the_sensor_process;
end sensor_type.temperature;

processor sensor_processor_type
features
  network : requires bus access local_bus_type.pci;
end sensor_processor_type;

process sensor_process_type
features
  sensed  : out event data port sensor_data;
  control : in event data port sensor_control_data;
end sensor_process_type;

thread sensor_thread_type
features
  sensed  : out event data port sensor_data;
  control : in event data port sensor_control_data;
properties
  Dispatch_Protocol => periodic;
end sensor_thread_type;

process implementation sensor_process_type.one_thread
subcomponents
  sensor_thread : thread sensor_thread_type;
connections
  event data port sensed -> sensor_thread.sensed;
  event data port control -> sensor_thread.control;
properties
  Dispatch_Protocol => Periodic;
  Period => 20 ms;
end sensor_process_type.one_thread;
```

Figure 13. Partial AADL model of an example sense-compute-control system.

Figure 13 shows a partial AADL model of an example sense-compute-control application, a typical target application for AADL. The selected portion shows a temperature sensor driver, running on a physical processor connected to a local 33Mhz PCI bus.

The first thing to note about this specification is the level of detail at which the architecture is described. A component (`sensor_type.temperature`) runs on a physical processor (`the_sensor_processor`), which runs a process

(`sensor_process_type.one_thread`), which in turn contains a single thread of control (`sensor_thread`), all of which can receive control instructions through an 'in' port (`control`) and output temperature data through an 'out' port (`sensed`) over a PCI bus (`local_bus_type.pci`). Each of these different modeling levels is connected through composition, attachment of ports, and so on. This level of detail emphasizes the importance of tools, such as graphical editors, for modeling this information in a more easily understandable fashion.

The second thing to note is that several of the elements are annotated with specific properties that describe their operation in more detail. For example, the PCI bus transmits 4 bytes (32 bits) of information every 30 nanoseconds, and the sensor process runs and samples the temperature every 20 miliseconds. It is these details, tailored for real-time concerns, that make AADL's analysis tool-set possible.

AADL is a complex notation but the analysis capabilities that can be applied to it are substantial. Although it has not achieved the ubiquity or support of UML, the fact that it has gained traction in industrial communities puts it ahead of most first-generation ADLs in terms of influence—despite its relative complexity. This is indicative of the real need for domain-specific architecture modeling and analysis techniques. The difficulty of the AADL community in creating tool support for the notation, particularly visualizations, indicates that the development of AADL might be assisted by the existence of an environment in which new ADLs such as AADL can be constructed quickly.

2.5.2 Koala

Consumer electronics is a dynamic and highly competitive domain of product development. For decades, devices such as televisions and cable descramblers were relatively simple devices with a few, well-defined capabilities. Over the years, these devices have become more and more complex, largely due to enhancements in their embedded software. The latest incarnations of these devices include features such as graphical, menu-driven configuration, on-screen programming guides, video-on-demand, and digital video recording and playback. In a global marketplace, each of these devices must be deployed in multiple regions around the world, and specifically configured for the languages and broadcast standards used in those regions.

The increasing feature counts of consumer electronic devices are accompanied by fierce competition among organizations, and it is just as important to keep costs down as it is to deploy the widest range of features. From a software perspective, keeping costs down can be done in two primary ways: limiting the cost of software development and limiting the resources used by the developed software and thus the costs of hardware needed to support it. Additionally, manufacturers often "multi-source" certain parts. That is, they obtain and use similar parts – chips, boards, tuners, and so on — from multiple vendors, buying from vendors who can offer the part at the right time or the lowest price, and providing a measure of insurance against the failure of one particular part vendor to deliver. If the parts are not completely interchangeable, software can be used to mask the differences.

Product-line architectures provide an attractive way to deal with the diversity of devices and configurations found in the consumer electronics domain. Product line

architectures allow a single model to express the architecture of multiple systems simultaneously, through the explicit modeling of variation points. Each variation point captures a number of possible alternatives. Products in the product-line are selected by choosing from the alternatives at each variation point.

Philips Electronics has developed an approach called Koala [126] to help them specify and manage their consumer electronics products. Koala is primarily an architecture description language derived from the Darwin ADL, described earlier. Koala also contains aspects of an architectural style, however, since it prescribes specific patterns and semantics that are applied to the constructs described in the Koala ADL. Koala, like Darwin, is effectively a structural notation: it retains Darwin's concepts of components, interfaces (both provided and required), hierarchical compositions (components with their own internal structures) and links to connect the interfaces. In addition to these basic constructs, Koala has special constructs for supporting product-line variability. Koala is also tightly bound to implementations of embedded components: certain aspects of Koala, such as the method by which it connects required and provided interfaces in code, are specifically designed with implementation strategies in mind, such as static binding through C macros.

Koala's main innovations over Darwin include:

IDL-based interface types: An interface type in Koala is a named set of function signatures, similar to those found in the C programming language. For example, the interface to a TV tuner in Koala might be declared like this:

```
interface ITuner{
      void setFrequency(int freqInMhz);
      int getFrequency();
}
```

The ITuner interface type may be provided or required by any number of Koala components. When a provided and a required interface are connected, the provided interface type must provide at least the functions required by the required interface type.

Diversity interfaces: One of the philosophies of Koala is that configuration parameters for a component should not be stored in the component; instead, configuration parameters, including selection of alternatives, should be accessed by the component from an external source when needed. This allows the application to be configured centrally, from a single component or set of components whose purpose is to provide configuration data for the application. “Diversity interfaces” are special required interfaces that are attached to components and are used by each component to get configuration parameters.

Switches: A switch is a new architectural construct that represents a variation point. It allows a required interface to be connected to multiple different provided interfaces. When the variation point is resolved, only one of the connections will actually be present. Which provided-required interface pair is connected depends on a configuration condition. A switch is connected to a diversity interface to get its configuration parameters, just as a component would. Depending on the values returned through the diversity interface, the switch will route calls to one of the required interfaces connected to it. If this means that there will be disconnected components, that is, components that will never be invoked, then Koala will not instantiate these components to save resources.

Optional Interfaces: Several components may provide similar, but not identical, services. For example, a basic TV tuner component may have only the ability

to change frequencies, but an advanced TV tuner may be able to search for valid frequencies as well. It is possible for callers to a TV tuner to include an optional interface, and query whether this interface is actually connected or not. If it is connected, the caller can make calls on the optional interface; if not, the caller should behave/degrade gracefully.

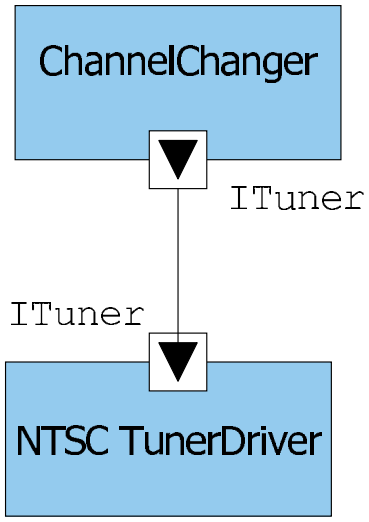


Figure 14. A basic Koala architecture for a television set.

Figure 14 shows a basic Koala architecture for a television set, one of many consumer-electronics devices for which it was developed. The similarities to Darwin's are obvious, although the graphical depiction is slightly different: the structural components communicate through explicit, matched provided and required interfaces. With respect to this research, it is interesting to note that although Koala is an intellectual descendant of Darwin—importing the same basic structural concepts, for example—it does not actually leverage the syntax or tools that were developed as part of the Darwin project.

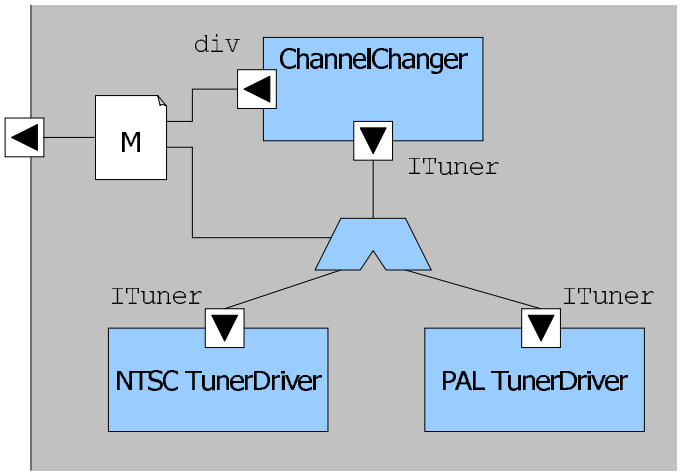


Figure 15. A Koala architecture for a product-line of televisions.

Figure 15 shows a variant of the previously shown simple TV architecture; this architecture specifies a product-line of televisions. Here, the ChannelChanger component calls out through a diversity interface to determine the system configuration; for some markets it will communicate with an NTSC tuner, for others it will communicate with a PAL tuner. This single specification captures two architectures with a single point of variation (the tuner selection). By adding additional variation points—through switches and diversity interfaces—the number of captured architectural variants can be increased exponentially.

2.5.3 Reflections on Second-Generation ADLs

Second-generation architecture description languages are, in many salient ways, natural extensions of their first-generation counterparts, tempered by the influence of UML. Instead of concentrating on generic and formal concerns as first-generation ADLs did, second-generation ADLs address a wider variety of concerns. For example, AADL addresses concerns that are domain-specific, arising from particular needs in the avionics, automotive, and embedded systems domains. Koala's product-line support

addresses a business goal—reuse of common aspects of architectures to reduce maintenance and development costs in a world where software development costs are rising and profit margins are dropping.

Although the selection of concerns in second-generation ADLs is broader, their construction is similar to that of first-generation ADLs. A core of common features—for example, modeling components, connectors, and interfaces—is expanded upon with distinguishing features. In AADL's case, these features include multi-level modeling and end-to-end analysis capabilities. Koala is distinguished by its ability to model explicit variation points and product lines.

As with first-generation ADLs, users who want to customize or extend second-generation ADLs have little recourse. The syntax of these ADLs remains fixed, and even basic decorative extension mechanisms (like UML stereotypes and tagged values) are missing.

2.6 Viewpoint Frameworks

Software and system architects and developers, particularly those from the practitioner community, are increasingly looking to a group of *de jure* and *de facto* standards for guidance in modeling architectures. Developers of these standards refer to them collectively as *frameworks*. The common thread uniting these frameworks is that they enumerate a set of viewpoints, or perspectives, from which architectures are described. Each viewpoint is intended to capture a different aspect of the architecture, usually centered around a concern. Because of the focus on defining viewpoints, these frameworks are referred to here as *viewpoint frameworks*.

Representative viewpoint frameworks include 4+1, RM-ODP, and DoDAF (formerly C4ISR). UML could also be seen as a viewpoint framework, because it too defines multiple perspectives from which to model architectures. The primary distinction between UML and these viewpoint frameworks is that UML defines individual notations for each viewpoint. These viewpoint frameworks, on the other hand, serve mostly to guide architects in *what* to capture, but leave out almost all the details about *how* to capture—including what notations to use. A brief description of representative viewpoint frameworks follows.

2.6.1 The 4+1 Viewpoint Framework

The 4+1 view model [98] was originally described by Philippe Kruchten. It is a general framework for thinking about how to document the architecture of a system from 5 different viewpoints. These are:

Logical Viewpoint: Captures the functional requirements of the architecture—services that the system should provide to its end-users. Designers generate logical views by decomposing the system into a set of (key) abstractions.

Process Viewpoint: This viewpoint addresses concurrency, distribution, system integrity, and fault tolerance. Here is where notions of time, concurrency, and synchronization get modeled. The process view deals with startup, shutdown, resource access, and how threads get assigned to various elements of the architecture.

Development Viewpoint: This viewpoint focuses on how the software is organized in a software development environment—into packages, libraries, subsystems, and so on. These grouping elements may also be organized, for example, hierarchically.

Physical Viewpoint: The physical viewpoint describes how the software elements described in the logical, process, and development views are deployed on various hardware devices/nodes.

Use-Case Viewpoint (the +1 viewpoint): This viewpoint captures use cases of the system, similar to the use-case viewpoint of UML.

As with UML, these viewpoints capture various aspects of a system's architecture (in fact, the 4+1 viewpoint framework could be seen as a direct inspiration for many features in UML). Kruchten's description of the 4+1 views does include notations associated with each of the viewpoints. However, these notations are incompletely described, with only a paragraph or two at most. These notations are borrowed from earlier notations (e.g., Booch diagrams), or are extended versions of earlier notations. Indeed, the contribution of the 4+1 framework is the selection of views and their contents.

2.6.2 RM-ODP

RM-ODP [86] is an ISO standard for describing open distributed processing (ODP) systems. Although the characteristics of ODP systems are described over several pages of the RM-ODP standard, they can generally be described as information systems developed to run in an environment of independent, heterogeneous processing nodes connected by a network. From an architectural perspective, the RM-ODP standard defines five viewpoints and describes associated viewpoint 'languages' for documenting the architecture of an ODP system. These viewpoints are (descriptions copied from the specification):

Enterprise: A viewpoint on the system and its environment that focuses on the purpose, scope and policies for the system.

Information: A viewpoint on the system and its environment that focuses on the semantics of the information and information processing performed.

Computational: A viewpoint on the system and its environment that enables distribution through functional decomposition of the system into objects which interact at interfaces.

Engineering: A viewpoint on the system and its environment that focuses on the mechanisms and functions required to support distributed interaction between objects in the system.

Technology: A viewpoint on the system and its environment that focuses on the choice of technology in that system.

These viewpoints (and their associated languages) have various levels of concreteness. The computational and engineering viewpoints are more concrete and thus more constrained, since they have the goal of helping to guarantee interoperability and portability of components. The enterprise and information viewpoints are more general, and therefore their languages are less constrained.

Although the RM-ODP standard discusses ‘languages,’ what is described is not a set of concrete languages in the sense of programming or architecture description languages. Instead of prescribing a particular notation for a viewpoint, the RM-ODP specification prescribes a number of *characteristics* that a notation describing the viewpoint must meet. Thus, RM-ODP viewpoint languages are not languages in the traditional sense, but rather sets of requirements for a language. RM-ODP does include

a separate specification describing, at a very high level, how various formal specification languages (e.g., LOTOS [85], Z [152], SDL-92 [164]) might be used to describe architectural semantics for RM-ODP architectures. This is done by listing an RM-ODP concept, and then identifying a formal specification language concept that could be used to implement it. It is doubtful whether this, on its own, is enough to give users serious guidance as to how to leverage these formal specification languages to implement RM-ODP.

RM-ODP describes, in prose, a set of general concepts that apply to all viewpoints (e.g., object, environment of an object, interface, and so on), and then describes, again in prose, any specializations or extensions of these concepts for each viewpoint that must be described in the viewpoint language. The description of each viewpoint language includes additional rules that describe how the entities in the language should be used. For example, structuring rules describe what sorts of documents/diagrams are included in the viewpoint, and what entities appear there. Binding rules describe links between entities in the viewpoint language. As noted above, there are more specific rule-sets for the more concrete viewpoints (i.e., computational and engineering).

2.6.3 DoDAF—The Department of Defense Architecture Framework

DoDAF [50] is a U.S. Department of Defense standard for documenting systems architectures. It is the successor to the previous C4ISR architecture framework, and supersedes it. DoDAF is a multi-viewpoint approach, although it defines ‘view’ in an interesting way. DoDAF defines three ‘views’:

The operational view (OV): The OV "identifies what needs to be accomplished, and who does it." It defines processes and activities, the operational elements that participate in those activities, and the information exchanges that occur between the elements.

The systems view (SV): Products in the systems view describe the systems that provide or support operational functions, and the interconnections between them. The systems in the SV are associated with elements in the OV.

The technical standards view (TV): The TV represents a collection of standards, (engineering) guidelines, rules, conventions, and other documents intended to ensure that implemented systems meet their requirements and are consistent with respect to the fact that they are implemented according to a common set of rules.

It also recognizes some cross-cutting concerns that affect **all views (AV)**. Each of these views represents a high-level perspective on a system. Within each of these DoDAF views, several architecture products—documents—are defined. The DoDAF terminology is somewhat different from that used in this thesis: a DoDAF view corresponds to a set of viewpoints, with each kind of product corresponding to a viewpoint.

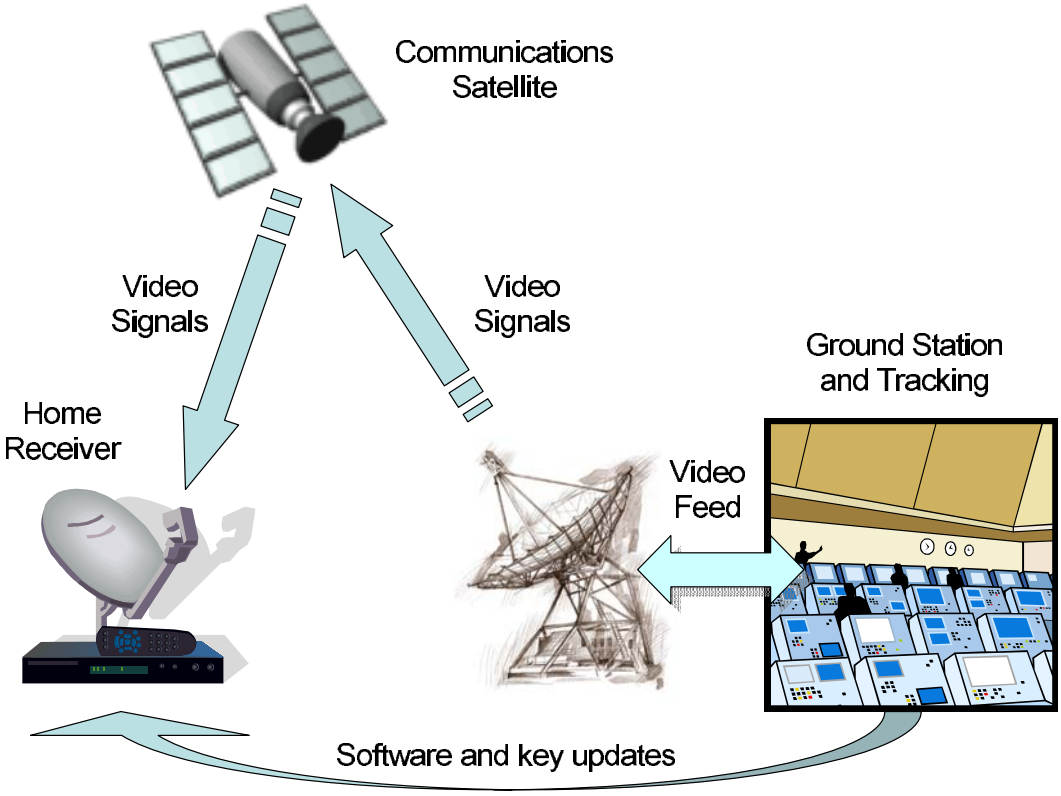


Figure 16. Example DoDAF OV-1 view of a direct-broadcast satellite TV service.

While DoDAF describes in great detail the breadth of information that should be captured in an architecture description, and categorizes the captured information into viewpoints, it does not prescribe any particular method for documenting the various viewpoints. For example, Figure 16 shows an example DoDAF OV-1 (High Level Operational Concept) view of a direct-broadcast satellite system. Here, the architecture is described notionally at a high-level, but the notation is arbitrary (and, in this case, chosen for aesthetic appeal rather than rigor).

DoDAF users can choose any subset of the viewpoints they want, and document them using any notation or tool they want. The DoDAF specification does indicate various UML diagrams that can be used to create documents for most (but not all) of the

DoDAF viewpoints, but it is not very clear as to exactly how this is to be done, and various organizations interpret this in different ways. (As an attempt to rectify this, the OMG has very recently submitted a request for proposals for DoDAF UML profiles [124]). The DoDAF specification also shows example documents for each viewpoint, but these are often simple and not in any particular canonical notation.

2.6.4 Reflections on Viewpoint Frameworks

As noted above, the primary strength of viewpoint frameworks is in identifying *what* should be modeled, but not *how* it should be modeled. Each viewpoint in a framework serves as a sort of checklist that architects can use to determine whether any important concerns have been omitted or overlooked in a modeling effort. In the case of extensive frameworks like DoDAF, they must also use their own judgment to decide which viewpoints are important and should be modeled in great detail, and which viewpoints can be modeled more casually.

With respect to stakeholder-centric modeling, viewpoint frameworks offer stakeholders a great degree of modeling freedom, but little guidance as to how to use it. Since they generally avoid prescribing even a sample concrete notation for capturing each viewpoint, stakeholders are left to select their own notations and address any deficiencies in their selections in an ad-hoc manner. Viewpoint frameworks do, however, motivate the need for better support for stakeholder-centric modeling: by recommending the modeling of an extensive set of viewpoints, not all of which are well-supported by existing notations, viewpoint frameworks are implicitly requiring stakeholders to find or adapt their own notations to the task. An environment for

developing or adapting modeling approaches to framework viewpoints quickly would be a key driver in the successful application of a viewpoint framework's guidance.

2.7 Meta-Modeling Environments

A related technology area that does not necessarily address architecture modeling is that of meta-modeling environments. These are tool-sets that allow users to define a modeling language or ontology in some selected meta-language, and then provide additional support for visualization, constraint evaluation and enforcement, and so on. Examples of such environments include Protégé [154], GME [100], and MetaCase [109]. These meta-modeling environments have many features that make them conducive for developing stakeholder-centric architecture modeling environments. The ability to rapidly define an ontology means that new syntactic constructs can be defined easily, and these tools can often create form-based or graphical editors automatically. Each environment includes some constraint language, often similar to OCL [175], that can express relationships between ontological elements that can be continuously enforced or validated when the user is building models. External tools can often be integrated through exposed interfaces or plug-in mechanisms, such as GME's COM-based [176] interfaces.

It is possible that the research described in this dissertation could have been done using one of these meta-modeling environments as a basis, as these environments satisfy or can help to satisfy many of the design principles described in Chapter 5. However, these environments themselves do not satisfy all the needs of stakeholder-centric architecture modeling. For example, these environments tend to have limitations with respect to partial ontology reuse and divergent extension. Additionally, graphical

visualizations in these environments tend to be easy to generate but limitedly customizable: in general, all elements in the ontology are either represented as nodes, and all relationships are expressed as edges between those nodes. Customization is limited to the graphical appearance of these nodes and edges, and it is difficult to develop complex visualizations where concepts do not naturally map to nodes and edges. These environments also offer little to no guidance about architectural modeling and ontologies, since they are meant to be generic tools for many kinds of domain-specific languages. An exception is the SoftArch [67] environment and related projects, which focus on modeling of architecture descriptions and refinement to further implementation artifacts, primarily in the context of object-oriented development.

3 Definitions

In the earlier matter, key terms such as ‘architecture’ and ‘model’ are used consistently, but without formal definitions. As is evident from the survey of background and related work in Chapter 2, many terms such as ‘view’ and ‘viewpoint’ are used inconsistently from one approach to another. This reflects the constant characterization and recharacterization of the entire area of architecture and architectural modeling as different organizations have struggled to organize and differentiate concepts. The definitions that follow have been selected based on a broad understanding of this previous work, and are also inspired by the context of stakeholder-centric modeling. Some definitions have been adapted specifically for discussion of this research, but many terms are used with their ordinary meaning, and for these a simple dictionary definition suffices. Where ordinary meanings are specified, appropriate definitions are selected from English dictionaries [47].

3.1 System

This term is used frequently, but with its ordinary meaning.

Definition: A *system* is a group of interacting, interrelated, or interdependent elements forming a complex whole.

In this research, the systems under discussion are software systems—that is, systems where the elements under consideration are primarily software, or entities that interact with software (e.g., hardware, networks, and so on).

3.2 Architecture

'Architecture' is the most critical term and concept to understand in the context of software architecture modeling, but its definition is one of the least agreed-upon. Nearly every architect has their own personal take on what architecture is and is not. Perry and Wolf [129] define architecture in terms of three concepts: elements, form, and rationale. Garlan and Shaw [148] refer to "gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives." The Software Engineering Institute (SEI) catalogs literally hundreds of different definitions on a website [28], most of which have been submitted by visitors who have their own opinions about what architecture is.

A key insight that must be taken into account when defining architecture is that not all systems are alike. Concerns that are absolutely critical for one kind of system may be of only marginal importance in another. While certainly some concerns are generally important across a wide variety of projects, attempting to identify a subset of concerns and term those concerns 'architectural' for all systems is narrow-minded. Instead, a broad perspective is needed—one that can be used as a basis for discussing architecture in the wide variety of software systems that are constructed.

Definition: A software system's architecture is the set of principal design decisions about the system.

This definition is among the broadest and most all-encompassing of definitions proposed for software architecture, but it is not entirely without precedent [89]. In a

way, most other definitions of architecture can be seen as derivations from, or specializations of, this definition. Fully understanding this definition requires the elaboration of two concepts: *design decisions* and *principality*.

This definition characterizes architecture as a composition of design decisions—indeed design decisions can be thought of as architectural quanta. Design decisions can encompass every aspect of the system under development, including:

- design decisions related to system structure – for example, “the architectural elements should be organized and composed exactly like this”,
- design decisions related to behavior (also referred to as functional) – for example, “data processing, storage, and visualization will be handled separately”,
- design decisions related to interaction – for example, “communication among all system elements will occur only using event notifications”, and
- design decisions related to the system’s non-functional properties – for example, “the system’s dependability will be ensured by replicated processing modules.”

Design decisions can exist across the spectrum of detail. The most general design decisions may concern global system properties—for example “the system shall be maintainable” or “storage of data elements shall be separate from computation in the system.” Detailed design decisions are more specific—“there shall be a component that stores the system state, called ‘StateComponent,’ that exposes the following

interface…" These three example decisions are clearly related—the more detailed decisions could be seen as refinements or effects of the more general decisions. Understanding the relationship between design decisions at different levels of detail or abstraction is one of the key challenges of architecture development.

Another important term that appears in the above definition is "principal." This implies that some decisions are important enough to grant them "architectural status." Conversely, not all design decisions are architectural—some will be at a level that is too detailed for principal design. The system's goals drive which design decisions are "principal" and which are not. For example, in a medical device, design decisions affecting the system's reliability and fault tolerance will obviously be principal. In a desktop application, decisions affecting usability and aesthetic appeal may be principal. Ultimately, it is the system's *stakeholders* (including, but not restricted to the system's architects) that must decide which design decisions are important enough to include in the architecture. 'Stakeholder' here is used with its ordinary meaning:

Definition: A *stakeholder* is a person or group that has an investment, a share, or an interest in something.

In this research, the target of interest or investment will generally be a software system.

3.3 Architecture Models and Modeling

Definition: An *architectural model* is an artifact or document that captures some or all of the design decisions that make up a system's architecture.

Definition: *Architectural modeling* is the effort to capture and document the design decisions that make up a system's architecture.

When we define architecture as a set of design decisions, then it follows naturally to define an architecture model as an artifact that captures and documents those design decisions. This introduces a purposeful distinction between the architecture of a system and the documentation of that architecture. This implies that a system may have an architecture, even if that architecture (or portions thereof) is not formally documented and resides only in the minds of key stakeholders. This is certainly a commonly experienced—if less than ideal—phenomenon.

Notations provide the means with which design decisions are documented.

Definition: An *architectural modeling notation* is a language or means of capturing design decisions.

Notations provide a way to encode, structure, and format design decisions. Different notations will be capable of (or optimized for) the capture of certain classes of design decisions.

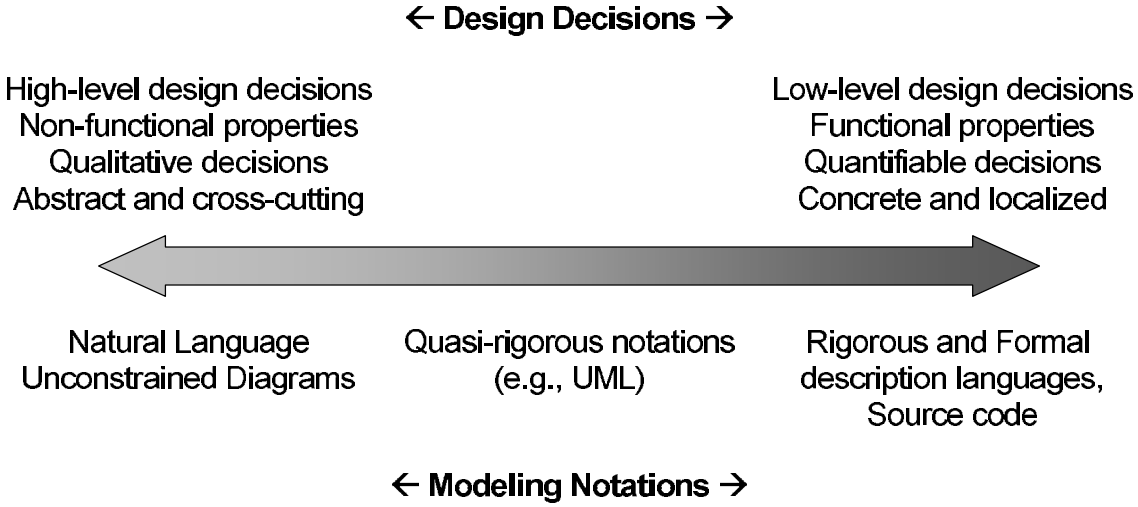


Figure 17. Spectrum of design decisions and appropriate notations.

Figure 17 shows a spectrum of kinds of design decisions matched with a spectrum of available notations. The selection of a particular notation to capture a design decision will affect the level of accuracy and precision with which that design decision can be captured. In general, it is easier to capture high-level, abstract design decisions in less formal languages, such as natural language. As design decisions become increasingly concrete, more rigorous, structured notations can be employed. Some notations, like Wright, go so far as to use mathematical formalisms to capture detailed design decisions about behavior. Notations like UML are somewhere in the middle—the syntax is more focused and rigorous than a natural language like English, but is still (to an extent) ambiguous.

3.4 Views and Viewpoints

Views provide a way to organize and present the design decisions that make up an architecture. If an architecture is characterized as a set of design decisions, then views can be characterized as (possibly overlapping) subsets of those design decisions.

Definition: A view is a set of design decisions related by a common concern (or set of concerns).

This definition organizes design decisions around a concern, or set of concerns. This is consistent with the multi-view frameworks and notations described earlier. For example, a UML 2.0 component diagram is a system view organized around the concern of the componentization (that is, functional decomposition and structural form) of a system.

Definition: A *viewpoint* defines the concern or criteria that are used to select the design decisions that form a view.

A view is an instance of a viewpoint for a specific system. A viewpoint is a filter for information, and a view is what you see when you look at a system through that filter. Put another way, if a view is characterized as a subset of design decisions, then the associated viewpoint can be characterized as the criteria with which that particular subset is selected.

For example, consider the domain of systems where software components are distributed across multiple hosts. The deployment viewpoint is a perspective on such systems that captures design decisions related to how components are assigned to hosts. The deployment view of a particular system captures how components are assigned to hosts for that particular system.

3.5 Visualization

Modeling is the activity of documenting, or 'writing down' the design decisions that make up a software architecture. To use these models involves interacting with those models—reading, understanding, writing, editing, transforming, and so on. This is the domain of architecture *visualization*.

Definition: A *visualization* is the 'user interface' through which stakeholders interact with an architectural model.

This is an intentionally broad definition of visualization. As a user interface, a visualization can be broken down into two key aspects: *depiction* and *interaction*.

Definition: *Depiction* is how the architecture is displayed to stakeholders.

Definition: *Interaction* is how the user interface elements associated with depictions allow stakeholders to explore and manipulate the architecture.

Visualizations are, in effect, interactive projections of architectures (or subsets of architectures—that is, views). Visualizations can come in many forms. Some visualizations are primarily (or solely) textual—they depict design decisions as a formatted stream of characters, akin to source code. Others are primarily (or solely) graphical—they depict design decisions as a set of organized symbols, often leveraging textual annotations to decorate those symbols. The relationship between visualizations of a model and the model itself are subtle. In effect, stakeholders never interact with models directly—a model is an abstract set of design decisions. Instead, stakeholders are always interacting with models through visualizations—projections of those design decisions in one or more concrete forms. Many notations have a preferred (or 'native') visualization, this is referred to as the *canonical visualization*. For example, UML's canonical visualization is its graphical one, but UML has other visualizations. XMI [123] is a textual visualization of UML, organized using XML. Both depict the same underlying design decisions, but in different ways. Darwin also has both textual and graphical visualizations. It is easy—and often harmless—to conflate a notation with its canonical visualization. However, it is important to recognize that there is still a distinction between how design decisions are modeled and how they are depicted.

3.6 Consistency

Architectural consistency, particularly consistency among views, is a critical property to establish and maintain. Viewing architecture as a set of design decisions gives us a straightforward way to reason about consistency as a property of the relationship between design decisions.

Definition: Design decisions are consistent if they make mutually compatible (i.e., non-contradictory) assertions.

This definition, like the above definition of architecture, is open-ended. For design decisions that address the same aspects of the system in the same terms and at the same level of detail, it is usually straightforward to locate inconsistencies (because they emerge as directly incompatible propositions, e.g. ‘A and not A’). For design decisions with different levels of abstraction or precision, defining and determining consistency is a more subtle process. This process involves stakeholders and their judgments; in this sense this is a stakeholder-centric definition of consistency.

3.7 Architectural Style

Through the construction of complex software systems over many decades, patterns, techniques, and knowledge about how to construct different kinds of systems have been extracted. When these concern architecture, they are often bundled into an *architectural style*.

Definition: An *architectural style* is a named collection of design decisions applied in a particular development context and intended to elicit known beneficial qualities.

An architectural style represents a ‘package’ of reusable design knowledge informing the development of architectures. The design decisions in this package, often expressed at the level of systemwide principles, guidelines, or constraints, are intended to elicit known beneficial qualities in the target system. For example, an architectural style may require that no two components assume that they run in a shared memory space. By adhering to this constraint, systems become easier to distribute across

multiple machines. Many architectural styles have been cataloged over the years. Some are simple and have few constraints, such as pipe and filter or implicit invocation [148]. Others are more complex, or hybridize multiple styles, such as C2 [161] or Weaves [65].

4 Approach

Architecture modeling is a multifaceted activity. Each approach surveyed in Chapter 2 provides a set of elements to its users—notations, visualizations, processes, consistency checking tools, and so on. Architecture modeling cannot be supported by a single one of these elements—a modeling solution must be composed of many of these elements working in concert. This indicates the need for not just a new notation or visualization strategy, but an *environment* for architecture modeling.

Many environments for architecture modeling have been created, and representative environments are surveyed in Chapter 2. However, none of these environments represents a solution for *stakeholder-driven* architecture modeling—modeling architectures in a world where stakeholders materially participate in defining the modeling approach. This motivates the first hypothesis of this dissertation, restated here:

H1. An environment can be constructed that supports stakeholder-driven multi-view software architecture modeling, addressing modeling, visualization, and consistency checking,

This hypothesis is an existence hypothesis: it asserts that such an environment can be constructed. To evaluate this hypothesis, I have constructed such an environment, called ArchStudio. The key insight that distinguishes ArchStudio from other architecture modeling environments is that it is constructed with *pervasive extensibility* in mind throughout. That is, each aspect of the environment can be extended, in a principled way, by its users. A related distinguishing insight in ArchStudio is that no modeling concept should be privileged over any other. Each of

ArchStudio's modeling capabilities is built up using ArchStudio's extensibility mechanisms. This differs from, for example, UML stereotypes or Acme properties, where extensions are clearly delineated from 'core' concepts. This puts each ArchStudio extension on equal footing, and gives all users the same power to adapt the environment.

The following sections describe the construction of the ArchStudio environment in terms of its main points of extensibility—the notation, visualizations, consistency checking, and custom tool integration. The intent here is to focus on how the environment is constructed—a discussion of the underlying principles and design guidelines that drove the construction follows in Chapter 5.

4.1 xADL 2.0: ArchStudio's Extensible Modeling Notation

When constructing an architecture modeling environment, the notation is the logical place to start. The notation defines the meets and bounds of what can be modeled, indicates how the information in the model is organized, and often implies certain desired semantics for that information[1]. Visualizations, consistency checkers, and other tools all work within the bounds of the information encoded by the architecture modeling notation.

To understand how architecture modeling notations are defined, they must be considered at the meta-language level: the languages in which the notations themselves

[1] A notation itself rarely enforces desired semantics, except to the extent that the notation could prevent a user from encoding semantically-invalid information via syntactic restrictions. External tools and editors are much more likely to understand and enforce the semantics of a notation.

are defined. For natural language, the meta-language is a set of grammar rules which may be loosely-defined or have large numbers of exceptions. For more rigorous notations like Darwin or Acme, the meta-language used is likely to be some variant of Backus-Naur Form (BNF) [4], which define a language in terms of a set of production rules that can be used as inputs for tools such as lex and yacc [93] to generate language parsers, compilers, and other tools. UML defines its own meta-language, called the meta-object framework (MOF) [122], which closely resembles a UML class diagram. Each symbol (including things like association arrows) in UML is defined as a class in the MOF, and the MOF uses UML-like associations to define the relationships between these symbols (for example, ‘one-to-many’ or ‘generalization’).

In an environment for stakeholder-driven modeling, the environment must support users defining and re-defining the architecture modeling notation at the meta-language level. This means that meta-language descriptions of the architecture modeling notation must be provided as part of the environment itself, and must be user-modifiable.

Most architecture modeling notations can be described as monolithic: they are defined at the meta-language level without internal partitions or segmentation. BNF-based languages are defined in a single BNF document, for example. UML is a partial exception: there are boundaries between each diagram type. However, UML can also be seen as a set of individual, independent notations (one for each diagram type). In this light, UML is really a set of many monolithic notations.

Monolithic notations are not adequate to support stakeholder-driven modeling, however. Here, users are allowed to change the notation at the meta-language level. If a

monolithic notation is used, there is no way to confine or understand the scope of the effects of any change to the notation. Is a core concept being fundamentally changed, or are details being added? Put another way, with monolithic notations each change to the notation creates an entirely new notation. Each modification to the language, no matter how minor, may affect how all the environment's other tools—for visualization, analysis, and so on—are constructed.

ArchStudio takes a fundamentally new approach in the design of its architecture modeling notation—one of *modular* language design. Its underlying modeling notation, called xADL 2.0 [39, 43, 44], is not defined in one large block of meta-language. Instead, xADL 2.0 is defined in a set of meta-language modules that are composed into the final xADL 2.0 language. Each module adds some modeling feature or features to the language—perhaps the ability to describe components and connectors, or the ability to indicate that some particular element in the architecture is optional.

Not all meta-languages are well-suited to developing modular languages such as xADL 2.0. BNF, for example, does not include a way for one production rule in the meta-language specification to redefine another production rule. Many meta-languages and environments exist in which an extensible, modular language can be created, among them XML, Lisp [155], GME [100], and DOME [79]. From a technical and theoretical perspective, ArchStudio could have used any of these to define its modeling notation (although they are not all equivalent). From a practical perspective, however, XML has far more support from standards committees, tool vendors, and practitioners than any other alternative. XML is platform-neutral, unbound to any particular hardware or network architecture. Many ancillary standards support XML, such as XLink [46],

which allows linking within and between XML documents, and XPath [33], which allows indexing of specific elements and attributes within a document. A plethora of off-the-shelf tools for constructing, visualizing, storing, and manipulating XML documents are available. This is not to say that XML is by any means perfect, even for ArchStudio—specific drawbacks and selection criteria for meta-languages in this context are described in Chapter 5. For ArchStudio, XML's support for modular extensibility and extrinsic benefits outweigh its technical drawbacks.

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

Figure 18. Sample data marked up in XML format.

XML documents are text documents in which some of the text is marked up using specially formatted tags that define the beginning and end of a segment of marked-up text. Data delimited by a start and end tag is known as an element; start tags can contain additional annotations called attributes. Data marked up with elements and attributes is shown in Figure 18.

XML is a language, but in a weak sense. It defines the general format of elements and attributes (using angle-brackets to delimit tags, for example). A document conforming to this basic structure of tags and attributes is known as a *well-formed* XML document. XML does not define what elements are allowed, in what order, with what attributes, with what content, and so on. For example, the data in Figure 18 is obviously marked up with a particular structure. The name 'name' has been given to the outermost tag, and the attribute 'lang' has been used to indicate the (natural) language in which the name is expressed—in this case, US-English. This selection of tag name and attribute

name/format is arbitrary—the tag ‘moniker’ could have been used in place of ‘name’ and the attribute ‘language’ could have been used in place of ‘lang.’ Plain XML does not provide a way to make this distinction. To do that requires a grammar of elements and attributes. When a grammar of elements and attributes exists, an XML instance document (such as the one in Figure 18) can be checked against this grammar. If the document is well-formed and also conforms to the grammar, it is known as a *valid* XML document.

It is possible to introduce a grammar of elements and attributes over XML documents using one of several available meta-languages. Several meta-languages for defining XML grammars are available, two of which have been standardized by the World Wide Web Consortium (W3C): DTDs and XML schemas [18, 54, 170]. DTDs offer limited mechanisms for supporting modular extensibility, as evidenced in the Modularization of XHTML standard [8]. However, XML schemas are much more capable in this regard, offering an easy way to define constructs in one schema and extend them in another through a mechanism very similar to object-oriented subtyping.

A subtype can add new elements and attributes to the content model of its supertype. Unlike in object-oriented types, however, XML schema subtypes can also be more restrictive than their supertypes in certain ways. Specifically, they can restrict the cardinality of elements in their supertype. For example, if an element in the supertype has cardinality 0-n, the cardinality of the element in a restricted subtype could be 0-0, eliminating the element entirely.

The type system over XML elements and attributes provided by XML schema provides substantial abilities to define modular languages. In particular, two types of extensions can be created:

1. Extensions that add new types of elements and attributes to the language.
2. Extensions that specialize or add on to existing elements and attributes in the language.

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

Figure 19. A simple definition of an architectural component in XML schema.

For example, Figure 19 shows a simple definition of an architectural component, as expressed in an XML schema[2]. Here, a component comprises an identifier, a description, and zero or more interfaces. Identifiers, descriptions, and interfaces are assumed to be defined elsewhere in the schema document. An extension of type (1) above, might add something completely new to this description language: an architectural link, for example.

```
<complexType name="Link">
  <sequence>
    <element name="description" type="Description"/>
    <element name="point" type="Point"
             minOccurs="2" maxOccurs="2"/>
  </sequence>
  <attribute name="id" type="Identifier"/>
```

[2] Examples figures in XML or XML schema in this dissertation will often be fragments and elide some or all XML namespace information for clarity; these figures are meant to be instructive and illustrative rather than normative.

```
</complexType>
```

Figure 20. A simple definition of an architectural link in XML schema.

Figure 20 shows what the definition of a (first-class) link might look like. This definition is independent of the above definition of a component, although the endpoints of the link may use XLinks to indicate components or interfaces in instance documents. This is an example of an extension that would add an entirely new construct to the architecture modeling notation.

```
<complexType name="OptionalComponent">
  <complexContent>
    <extension base="Component">
      <sequence>
        <element name="guardCondition" type="GuardCondition"
                 minOccurs="1" maxOccurs="1"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Figure 21. An extension to the earlier 'Component' type adding a guard condition.

Figure 21 shows a different kind of extension: one that adds on to a previously defined type. Here, the earlier definition of a component is expanded with a guard condition to make an optional component—one that is included in the architecture only if the guard is satisfied. Because of the way XML schema subtyping works, every OptionalComponent can be treated as an ordinary Component by tools reading an instance document, since OptionalComponents must still contain an identifier, description and zero or more interfaces. XML tools are generally programmed to ignore unknown attributes and elements, so the additional presence of a guard condition would not be problematic for tools that only understand how to deal with plain components.

Using these two extension mechanisms (definition of new element types and specialization through subtyping), complex languages can be created. XML Schema

does not restrict where subtyping may be applied—in the above example, descriptions or interfaces could also be subtyped, and they could contain new elements that are also subtyped in various ways. New types and subtypes can be grouped together in schema documents. Through the use of XML namespaces, types in one schema can refer to (to include or to extend, for example) types in another schema. The grouping of element and attribute types and subtypes into schemas is arbitrary. A collection of schemas makes up a complete grammar for an XML-based language—in this case, an XML-based language for describing architectures.

The use of XML Schema as a meta-language for architecture models offers many direct benefits in the context of stakeholder-driven modeling:

- It places few limits on what can be expressed in an architectural model;
- It allows new modeling features to be added and existing features to be modified over time;
- It allows experimentation with new features and combinations of features, and
- It allows modeling features, once defined, to be reused in other projects.

4.1.1 An Overview of xADL 2.0 Modules

The insight to modularize the architecture modeling notation, and the selection of an appropriate meta-language do not help to answer a fundamental question: can architecture modeling languages be modularized in this way? Conceptually, this modularization should be possible: refer back to Figure 7 to see how modeling features may be broken up. The breadth of modeling notations and approaches available leaves no shortage of potential features to modularize and recompose. To evaluate whether it

was possible to define an architecture description language modularly, a representative set of concerns was selected and modularized. This set of modules also provides a base of generic, reusable modeling features that can be used as a springboard for more domain- or project-specific extensions. These modules comprise the xADL 2.0 language.

In xADL 2.0, modules correspond roughly to ‘features’ (that is, capabilities and concepts) of the modeling language. Criteria for how to modularize an architecture description language into features in this way are discussed in Chapter 5.

The collection of schemas is evolving, making it difficult to define the precise boundaries of xADL 2.0 at any given time. By convention, the schemas included in xADL 2.0 are those posted on the xADL 2.0 website [83] and supported by the official release of ArchStudio. New schemas and corresponding extensions to ArchStudio from outside contributors are encouraged and vetted by the ArchStudio development community for possible inclusion on the official website.

To date, many xADL 2.0 modules have been developed. Some of these are generic and apply in many different domains, such as those used for modeling architectural structure. Some add features to the language such as the ability to model product-line architectures or map structures to implementations. Some are project-specific modules, adding data to xADL that is used only by a single tool or research project. The existence of this variety of modules is evidence that architecture modeling can be effectively modularized into interdependent schemas, corresponding to different modeling features, and that these schemas can then be composed into an architecture modeling notation.

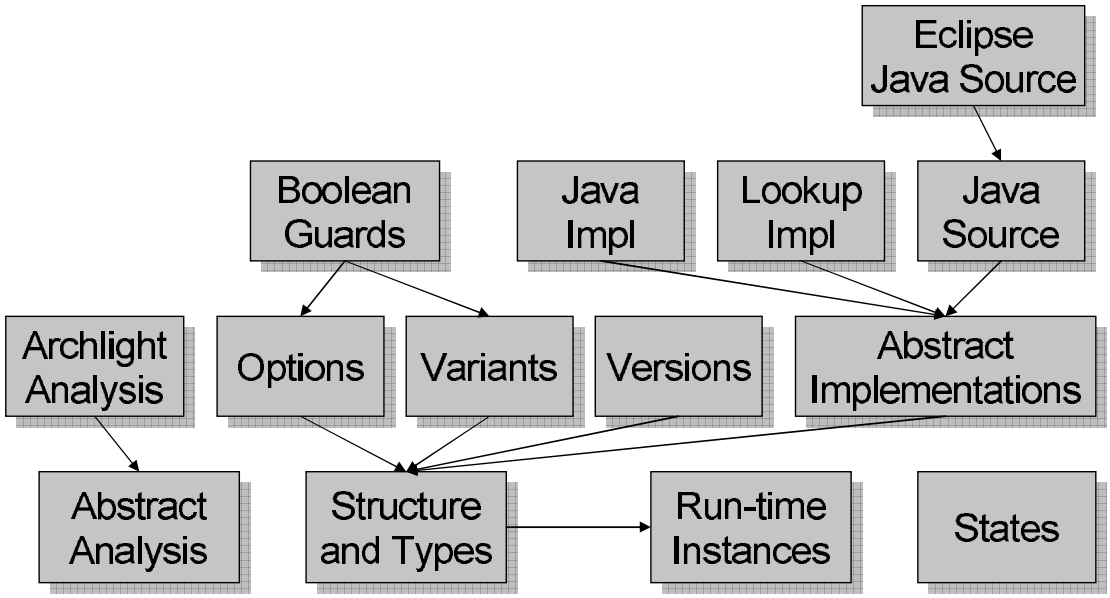


Figure 22. Selected xADL schemas and dependencies.

Figure 22 shows a set of selected xADL 2.0 schemas, as well as their conceptual dependencies. (The actual XML schema dependencies are somewhat different for two reasons: first, schemas will often reuse a simple construct like an identifier or description from another schema; second, XML schema's single-inheritance subtyping model limits divergent extension and caused the introduction of several artificial dependencies. These issues are discussed in Chapter 5). The first thing to notice about the dependency tree is that it is relatively wide: many features are conceptually orthogonal and do not interfere with one another. The xADL 2.0 experience thus far indicates that while feature interactions do exist and must be dealt with on a case-by-case basis, they are not pervasive and do not represent a limit on the extensibility of the language itself. The design of the language modules themselves has an effect on this property, and guidelines for designing language modules to avoid feature interactions are discussed in Section 5.7.

The lowest-level modules in the tree (those with the fewest dependencies) are the most generic. Modules higher in the tree add more detail to the language and thus become more specialized. As is evident on the right side of the tree, for example, an abstract specification for implementations is provided in a base schema, and increasingly specific classes of implementations (e.g., Java source code, Java source code in the Eclipse environment) build on that base schema.

Purpose	Schema	Features
Design-time and run-time structural modeling	Instances	Run-time component, connector, interface, and link instances, sub-architectures, general groups.
	Structure & Types	Design-time components, connectors, interfaces, and links, component, connector, and interface types, sub-architectures, general groups.
Product-line architectures using explicit points of variation	Options	Ability to specify that some architectural elements are optional depending on a guard condition.
	Variants	Ability to specify components and connectors whose types vary based on a guard condition.
	Versions	Ability to specify version graphs for architectural elements.
Implementation Mappings	Abstract Implementation	Placeholder for implementation data that indicates placement of implementation data on component, connector, and interface types.
	Java Implementation	Implementations that map to Java binaries.
	Lookup Implementation	Implementations that map to names that can be looked up in a name registry.

	Java Source Code	Implementations that map to Java source files.
	Eclipse Java Source Code	Implementations that map to Java source files developed in the Eclipse development environment.
Architectural analysis and consistency checking	Abstract Analysis	Placeholder for analysis test data at the document level.
	Archlight Analysis	Associates architectural documents with Archlight tests that can be run on the document.
Architectural states	States	Simple statecharts describing architectural states.

Table 1. Selected xADL schemas and the features they add.

Table 1 shows a selected set of xADL schemas and briefly describes the purpose of each schema—that is, what modeling features that schema adds to the xADL language. Each schema is described below. Where appropriate, examples will illustrate how xADL might be used to model the software system supporting a hypothetical consumer electronics product.

4.1.2 Design-time and Run-time Structural Modeling

Many traditional ADLs have viewed software architecture as a design-time artifact, focusing their modeling capabilities on the design phase of development. However, research into run-time software evolution and dynamism has shown that it is useful to maintain an architectural model of the system at run-time as well [128, 146]. The design-time and run-time models of a system will be similar, but not identical. At design time, a system model may contain basic metadata about elements (author, size, textual descriptions), specification of intended (not observed) behavior, and constraints

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

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

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

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

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

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

Sub-architectures: Components and connectors may be atomic (not broken down further at the architectural level) or composite. Composite components and connectors have internal architectures, called sub-architectures.

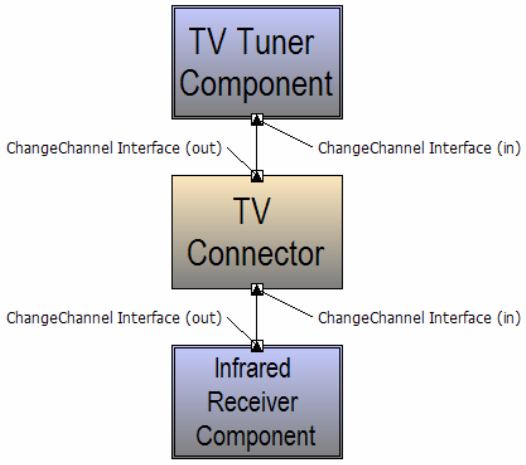
General Groups: Groups are simply collections of pointers to elements in the architecture description. In these schemas, a group has no semantics. Groups with specific meanings (e.g., common authorship, common platform, similar functionality) can be specified in extension schemas.

Because the INSTANCES and STRUCTURE & TYPES schemas have definitions for the basic structural elements (components, connectors, etc.), they tend to be core schemas—used extensively in architectural models and the basis for many extensions of xADL 2.0. By maintaining definitions of these elements in both schemas, they can be extended separately: the INSTANCES schema should be extended to provide additional run-time data and the STRUCTURE & TYPES schema should be extended to provide additional design-time data. To provide traceability, XLinks connect run-time elements to their design-time counterparts.

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

control. These two components are connected by a software connector that allows the infrared receiver component to send signals to the TV tuner to change the channel.

```
<xArch>
  <archStructure id="tvset">
    <description>TV Set</description>
    <component id="tuner">
      <description>
        TV Tuner Component
      </description>
      <interface id="tuner.channel">
        <description>
          ChangeChannel Interface
         (in)
        </description>
        <direction>in</direction>
      </interface>
    </component>
    <component id="ir">
      <description>
        Infrared Receiver Component
      </description>
      <interface id="ir.channel">
        <description>
          ChangeChannel Interface
         (out)
        </description>
        <direction>out</direction>
      </interface>
    </component>
    <connector id="tvconn">
      <description>
        TV Connector
      </description>
      <interface id="tvconn.in">
        <description>
          ChangeChannel Interface
          (in)
        </description>
        <direction>in</direction>
      </interface>
      <interface id="tvconn.out">
        <description>
          ChangeChannel Interface
          (out)
        </description>
        <direction>out</direction>
      </interface>
    </connector>
```




```
    <link id="link1">
      <description>
        Tuner to Connector
      </description>
      <point>
        <anchor
          href="#tuner.channel"/>
      </point>
      <point>
        <anchor
          href="#tvconn.out"/>
      </point>
    </link>
    <link id="link2">
      <description>
        Connector to IR
      </description>
      <point>
        <anchor href="#tvconn.in"/>
      </point>
      <point>
        <anchor
          href="#ir.channel"/>
      </point>
    </link>
  </archStructure>
</xArch>
```

Figure 23. xADL 2.0 structural view of the software architecture of a simple television set with accompanying diagram. Some XML data (e.g., namespaces) elided for clarity.

Figure 23 shows how the xADL STRUCTURE & TYPES schema might be used to model the structure of this system, using two depictions—a textual depiction showing the structure of the XML (with some ancillary markup data such as namespaces left out for clarity), and a graphical depiction of the same architecture as generated by ArchStudio's graphical editor, Archipelago (which is discussed below).

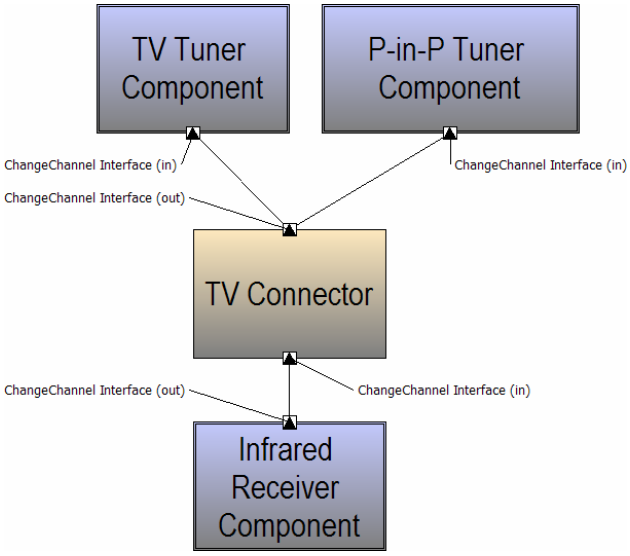
4.1.3 Design-Time Types

Typing is an important concept in most ADLs. xADL 2.0 incorporates the base structures of a typing system that supports type equality and composition. The purpose of xADL's typing system is to allow users a way to relate different instance elements by way of their types, and to specify common information in a single location (e.g., on the type) instead of on each instance. More formal type systems, such as Darwin's pi-calculus-based system, are intended to be specified in extensions.

In xADL 2.0, the use of types is optional. Each component, connector, or interface can optionally contain an XML link to a type; multiple elements can share a type. The type serves as a construct where common properties of elements can be specified once (in the type) instead of in each element declaration. In the STRUCTURE & TYPES schema, these types consist of a unique identifier and a textual description along with a set of signatures. Signatures are prescribed interfaces; two components or connectors of the same type should have the same types of interfaces.

This can be demonstrated by adding types to the television example. One additional component will be added to the television: a picture-in-picture tuner. Televisions with picture-in-picture require two tuners to support the display of both

```
<xArch>
  <archStructure>
  <!--Contents from previous
      figure included here, but
      omitted for space. Also,
      each component, connector
      and interface would have a
      type link, and each
      interface would have a
      signature link, as shown.-->
    <component id="piptuner">
      <description>
        Pic in Pic Tuner
      </description>
      <interface
        id="pinptuner.channel">
        <description>ChangeChannel
          Interface (in)
        </description>
        <direction>in</direction>
        <signature href=
          "#tuner_type.channel"/>
        <type href=
          "#changechannel_type"/>
      </interface>
      <type href="#tuner_type"/>
    </component>
    <link id="link3">
      <description>
        PinP Tuner to Conn
      </description>
      <point>
        <anchor href=
          "#piptuner.channel"/>
      </point>
      <point>
        <anchor
          href="#tvconn.out"/>
      </point>
    </link>
  </archStructure>
  <archTypes>
    <componentType
      id="tuner_type">
      <description>TV Tuner Type
      </description>
      <signature
        id="tuner_type.channel">
        <description>
          ChangeChannel Sig (in)
        </description>
       <direction>in</direction>
        <type href=
          "changechannel_type"/>
      </signature>
    </componentType>
```




```
    <componentType id="ir_type">
      <description>Infrared Receiver Type
      </description>
      <signature id="ir_type.channel">
        <description>ChangeChannel Sig
         (out)</description>
        <direction>out</direction>
        <type href="changechannel_type"/>
      </signature>
    </componentType>
    <connectorType id="tvconn_type">
      <description>TV Connector Type
      </description>
      <signature id="tvconn_type.in">
        <description>
          ChangeChannel Sig (in)
        </description>
        <direction>in</direction>
        <type href="changechannel_type"/>
      </signature>
      <signature id="tvconn_type.out">
        <description>
          ChangeChannel Sig (out)
        </description>
        <direction>out</direction>
        <type href="changechannel_type"/>
      </signature>
    </connectorType>
    <interfaceType
      id="changechannel_type">
      <description>
        Change Channel Interface
      </description>
    </interfaceType>
  <archTypes>
</xArch>
```

Figure 24. TV set xADL model with an additional component and types.

channels simultaneously, but since the main tuner and the picture-in-picture tuner are basically identical, they will share a type.

Figure 24 shows an augmented version of the xADL model in Figure 23, this time with the addition of the picture-in-picture tuner and types for each component, connector, and interface. Note that each component, connector, and interface is linked to its type with a ‘type’ XLink, and each interface is additionally linked to the signature that prescribes it with a ‘signature’ XLink. This helps to establish and check type consistency.

Types are also used in xADL 2.0 to support design-time sub-architectures (that is, components or connectors that have internal architectures also specified in xADL 2.0). Thus, two components or connectors that share a type also share an internal architecture. Specifically, types can have an optional XLink to another ArchStructure element describing the internal architecture, as well as a set of mappings between signatures on the type and interfaces on components and connectors in the sub-architecture. These mappings serve to link the outer architecture with the inner one.

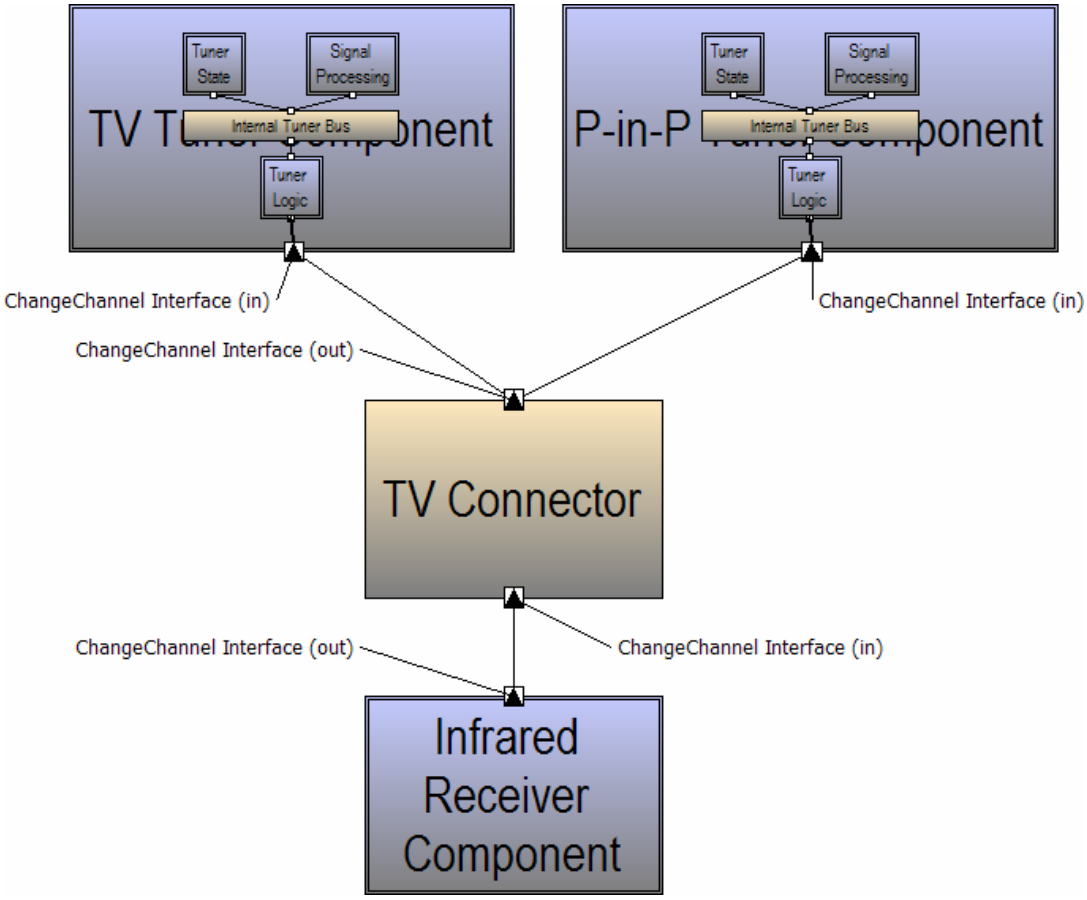


Figure 25. An augmented diagram of the xADL TV set example, here with sub-architectures.

An XML visualization of a xADL document with sub-architectures is too complex for this dissertation; however Figure 25 shows the graphical visualization of such an architecture: here, the tuner components have internal architectures defined, further detailing their construction. The outer ChangeChannel interface is mapped to an interface on the internal TunerLogic component. As both components share a type, they also share an internal structure.

4.1.4 Modeling Product Line Architectures

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

evolve over time. As a product evolves, so does its architecture. Second, software products are often part of a larger product line. A product line is a family of related software products that share significant portions of their architectures, with specific points of variation. A single product line may contain products that are localized for specific regions, or represent different feature sets for marketing purposes (e.g., Standard, Professional, Enterprise). Koala, discussed above in Section 2.5.2, is an example of an architecture description language that does support product-line modeling.

From a modeling perspective, both architectural evolution and product lines can be addressed by applying traditional concepts from configuration management to software architectures. Inspired by previous research in modeling architectural product lines [22, 34, 126], xADL 2.0 integrates these concepts in the form of three schemas: the VERSIONS, OPTIONS, and VARIANTS schemas [42]. Versions record information about the evolution of architectures and elements like components, connectors, and interfaces. Options indicate points of variation in an architecture where the structure may vary by the inclusion or exclusion of a group of elements (components, connectors, links, and so on). Variants indicate points in an architecture where one of several alternatives may be substituted for an element or group of elements. These three schemas are orthogonal, such that modelers can choose to use only the VERSIONS, OPTIONS, or VARIANTS schema, or any combination of the three.

4.1.4.1 Versions

The VERSIONS schema adds versioning constructs to xADL 2.0. It defines version graphs for component, connector, and interface types. In xADL 2.0, architecture

element types are the versioned entities. The decision to version types rather than elements such as components and connectors was made because it meshes well with the capabilities of the type system. For example, by versioning types, it is possible to have multiple instances of the same version of a component, connector, or interface by simply creating multiple instances of a type. Different versions of a component, connector, or interface can coexist in an architecture as well, simply by creating instances of types that share a version tree. This makes sense because, as architectures evolve, newer versions of elements may have different characteristics than older versions (e.g., additional signatures).

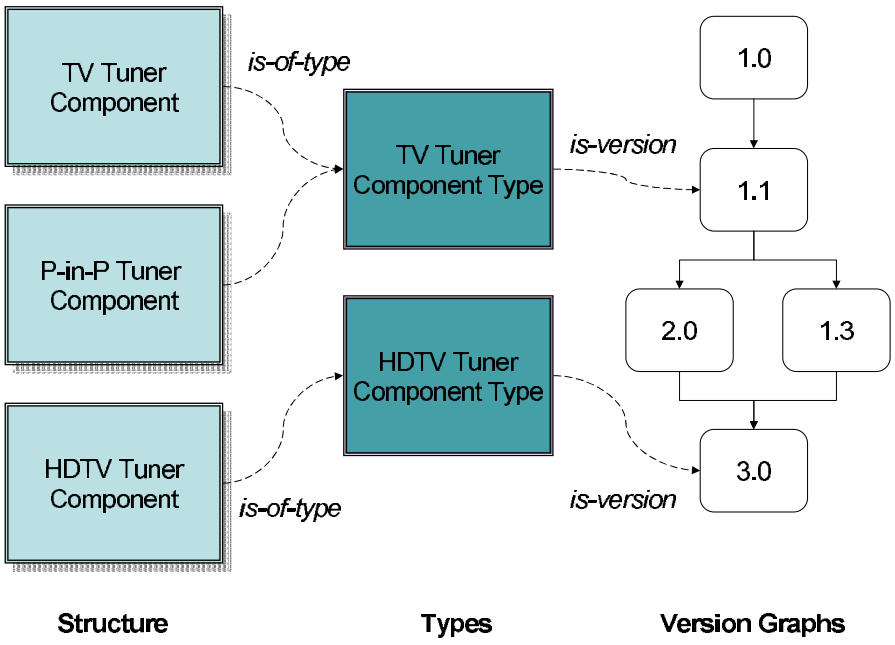


Figure 26. Relationships between structure, types, and version graphs in xADL 2.0.

The relationship between structural elements (e.g., components), their types, and version graphs is depicted in Figure 26. Here, three components (the TV tuner, the P-in-P tuner, and an HDTV tuner component) are associated with two component types. Both the TV tuner and P-in-P tuner are of the same type (as in our above example), but

the HDTV tuner is of a different type. The version graph shows that the TV Tuner Type and the HDTV Tuner Type share a common lineage. The HDTV Tuner Type is a later version of the TV Tuner Type, but both versions may be included in the same architecture if necessary.

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

In keeping with the generic nature of xADL 2.0 schemas, version graphs do not constrain the relationship between different versions of individual elements, for instance, that they must share some behavioral characteristics or interfaces. Such constraints may be specified in extension schemas and checked with external tools.

4.1.4.2 Options

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

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

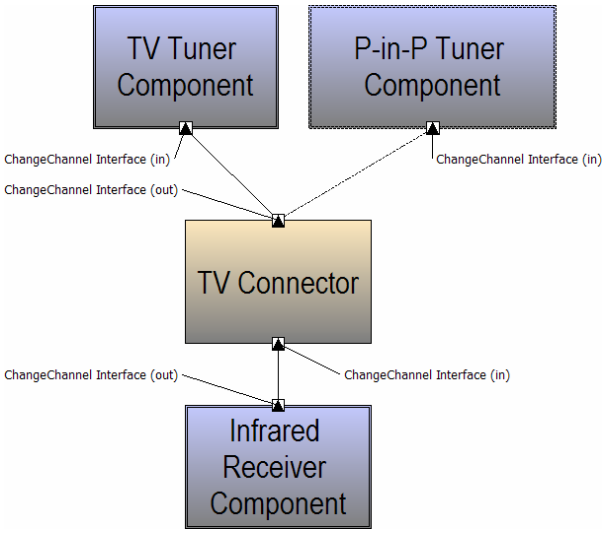


Figure 27. Product line architecture of televisions with optional picture-in-picture tuner.

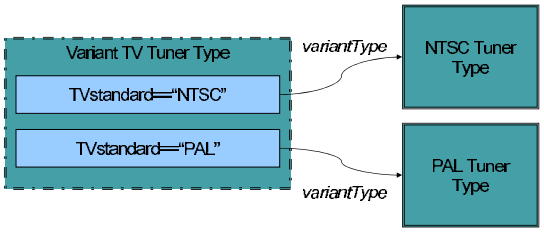
In our television example, we may wish to turn our single product architecture into a product-line by adding optionality. By making the picture-in-picture tuner optional, along with its link to the TV connector, we can create a product line that describes two products: a television with picture-in-picture and a television without it. The changes that we must make to our existing product description are shown in Figure 27. Here, we add an `<optional>` element to both the picture-in-picture tuner and its link to the connector. The guard condition for both `<optional>` elements is `hasPinP=="true"` so these elements will only be included in the architecture if the target product has picture-in-picture (i.e., the variable hasPinP is bound to the value 'true'). Variable-value bindings are established in a selector tool, described later in Section 4.9.2.

It is possible to express architectures with guards that could result in incomplete or incorrect architectures (e.g., if the link's guard in the example above were missing or different than the guard on the P-in-P tuner). When the product line is sculpted down to a single product via use of the selector, these inconsistencies will become apparent, and many of them (such as a dangling link) will be caught by ArchStudio's consistency checks.

4.1.4.3 Variants

The VARIANTS schema allows the specification of variant component and connector types in an architecture. A variant component or connector type defines a set of possible alternatives. Each alternative is a component or connector type accompanied by a guard condition, similar to the conditions used in the OPTIONS schema. Guards for variants must be mutually exclusive—that is, the conditions must be set up such that a variant type cannot resolve to two different types simultaneously. When the guard condition for one alternative is met, that alternative's concrete component or connector type is substituted in place of the variant type.

```
<xArch>
  ...
  <archTypes>
    <componentType id="tuner_type">
      <description>
        Variant TV Tuner Type
      </description>
    <signature
      id="tuner_type.channel">
      <description>
        ChangeChannel Sig
      </description>
      <direction>in</direction>
```



```
  <type
    href="#changechannel_type"/>
</signature>
<variant>
  <guard>
    <booleanExp>
      <equals>
        <symbol>
          TVstandard
        </symbol>
        <value>NTSC</value>
      </equals>
    </booleanExp>
  </guard>
  <variantType
    href="#NTSC_tuner_type"/>
  </variant>
<variant>
  <guard>
    <booleanExp>
      <equals>
        <symbol>
          TVstandard
        </symbol>
        <value>PAL</value>
      </equals>
    <booleanExp>
  </guard>
  <variantType
    href="#PAL_type"/>
  </variant>
</componentType>
```

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

Figure 28. Example of variant types in xADL 2.0.

The television product line can be additionally diversified by considering multiple international markets. Televisions distributed to North America or Japan will need NTSC tuners, while televisions distributed to most of Europe will need PAL tuners. We can express this by replacing the existing tuner type with a variant type that links to two possible concrete types: an NTSC tuner type and a PAL tuner type. This is shown in Figure 28. In this example, the structural description of the architecture does not change at all: both the main and picture-in-picture tuners retain the link to the same type (tuner_type). Now, however, that type has become variant: the tuners may be NTSC or PAL tuners, depending on the target market. The variant type includes two

individual variants (i.e., alternatives), expressing guard conditions (`TVstandard=="NTSC"` and `TVstandard=="PAL"`, respectively) and pointing via XLinks to ordinary concrete types representing an NTSC tuner and a PAL tuner. The included diagram shows this situation conceptually.

4.1.5 Implementation Mappings

Another important feature of xADL 2.0 is its support for mapping design-time architectural elements onto executable code. Several ADLs such as MetaH [17] support or require a mapping between an architecture specification and its implementation. This is essential if a software system is to be automatically instantiated from its architecture description, and can help to manage the transition from system design to implementation.

Since xADL 2.0 is not bound to a particular implementation platform or language, it is impossible to know in advance exactly what kinds of implementations will be used. Obvious possibilities include Java classes and archives, Windows DLLs, UNIX shared libraries, and CORBA components [121], but making a comprehensive list is infeasible. To address this, xADL 2.0 adopts a two-level approach.

The first level of specification is abstract and defines where implementation data should go in an architecture description, but not what the data should be. This indicates to future developers how to extend the xADL 2.0 schemas to add data for a new implementation platform. The xADL 2.0 ABSTRACT IMPLEMENTATION schema extends the STRUCTURE & TYPES schema, and defines a placeholder for implementation data. This placeholder is present on component, connector, and interface types. The intent is that two elements of the same type share an implementation. The second level of

specification is concrete, defining what the implementation data is for a particular platform or programming language. Concrete implementation schemas extend the ABSTRACT IMPLEMENTATION schema. For example, xADL 2.0 includes a JAVA IMPLEMENTATION schema that concretely defines a mapping from components, connectors, and interface types to Java classes. Other schemas such as the LOOKUP IMPLEMENTATION schema and the JAVA SOURCE CODE schema map to different kinds of implementation artifacts.

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

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

Figure 29. Mapping a component type to a Java binary implementation in xADL 2.0.

Implementation data for one of the component types in the television set example might look like Figure 29. Here, the television tuner component is implemented by two Java classes residing in the same JAR archive. The main class takes an initialization parameter of the television's model, which is useful if the component's implementation is multi-purpose or reusable in multiple contexts (e.g., for many television models). In this example, only the `<implementation>` element is specified by the ABSTRACT IMPLEMENTATION schema. Everything inside that element is specified by the JAVA IMPLEMENTATION schema—other implementation schemas would have different contents for this element.

4.1.6 Architectural Analysis and Consistency Checking

Architectural analysis and consistency checking schemas in xADL allow architecture descriptions to carry along with them information about the consistency rules that the description is meant to follow. The ABSTRACT ANALYSIS schema provides a placeholder (as a top level element) for analysis data. Extension schemas can provide additional information for different analysis tools and engines. One such engine is the Archlight framework, ArchStudio's primary analysis framework (described in detail below). The Archlight Analysis schema adds the ability for a document to reference different tests—consistency checks—offered by Archlight, and set whether those tests are enabled on the document or not. If a test is enabled, then it means that the architect intends for the document to pass that particular test.

```
<xArch>
  ...
  <analysis>
    <test enabled="true" id="test.uuid.1">
      <description>
        Tool: Schematron; Category: xADL 2.0/Structure and
        Types/Types/Connectors/Interfaces/Signature Link Valid
      </description>
    </test>
    ...
  </analysis>
</xArch>
```

Figure 30. Example of an enabled Archlight test in a xADL 2.0 document.

Figure 30 shows how an enabled test looks in xADL 2.0. Note that the description for an Archlight test is quite simple: it refers to the test through the test's universally-unique ID (UUID) and contains a short description of the test. All information needed to perform the test is contained elsewhere in the xADL 2.0 document and the test itself is provided by the Archlight framework. It would be possible, for some types of tests, to encode the test directly in xADL 2.0 with a different

sort of extension schema. Archlight tests do not include the test criteria in the xADL document because it makes it easier to evolve the tests themselves—if a bug were found in a particular test, and that test's specification was found in the xADL document itself, then all old xADL documents would become "buggy" and would need to be updated.

4.1.7 Other xADL Schemas

The schemas described in the above sections are a representative sample of xADL schemas; they have been selected for elaboration because they are the most commonly used and well-supported by ArchStudio and other tools. As discussed above, however, xADL is an evolving language, and individual users have the option to extend the language by adding their own schemas as necessary. Often, these extension schemas add detail to xADL for a specific research project or application. The projects for which these extensions were developed are described below in Section 6.5.

For example, the MESSAGE RULES schema allows users to specify rules that describe the behavior of message-based systems, in which components and connectors are treated as black-boxes that respond to the receipt of particular messages by emitting new messages to other components. This schema is used by the MTAT project [70, 72], described later, to heuristically infer causality relationships between messages received and emitted by a component or connector.

The STATES schema allows users to specify simple statecharts [68] (akin to those found in UML) in xADL 2.0. These statecharts are used by ArchStudio's "stateline" feature (see Section 6.1.4) to describe architectural states, with each state in the statechart corresponding to a particular product variant in a product line.

The SECURITY schema allows users to add security properties to a xADL description. These include identifying principals (individuals or entities that can access some element of the architecture) and privileges (indicating the permissions that various principals have). The SECURITY schema also incorporates parts of the XACML language [120] for additional policy specification. This schema was used primarily in the Secure xADL research project [135] that is the subject of Jie Ren's Ph.D. dissertation.

4.2 xADL 2.0 Tool Support

xADL 2.0 introduces tremendous new challenges with respect to language tool support, because the language itself is modular and is expected to be extended by its users in potentially unforeseen ways. Tools cannot rely on the language's syntax being static, nor can they rely on the language's original developers being the only ones that update the language (and thus able to update the tool support to match). xADL's tool support, then, makes extensive use of generic tools, generators, and reflection to provide the level of dynamicity required to support the protean underlying language.

xADL 2.0 is accompanied by a foundation layer of tool support that provides basic capabilities to xADL users: parsing, editing, serializing, low-level editing, and so on. These foundational tools tend to be syntax-driven: they leverage knowledge of the syntax and structure of the language (as expressed in the xADL schemas) to provide their services. However, they make no assumptions about the semantics of the language beyond what is expressed in the schemas. These syntax-based tools are, in turn, the foundation for other tools that are built with the intended semantics of specific schemas in mind.

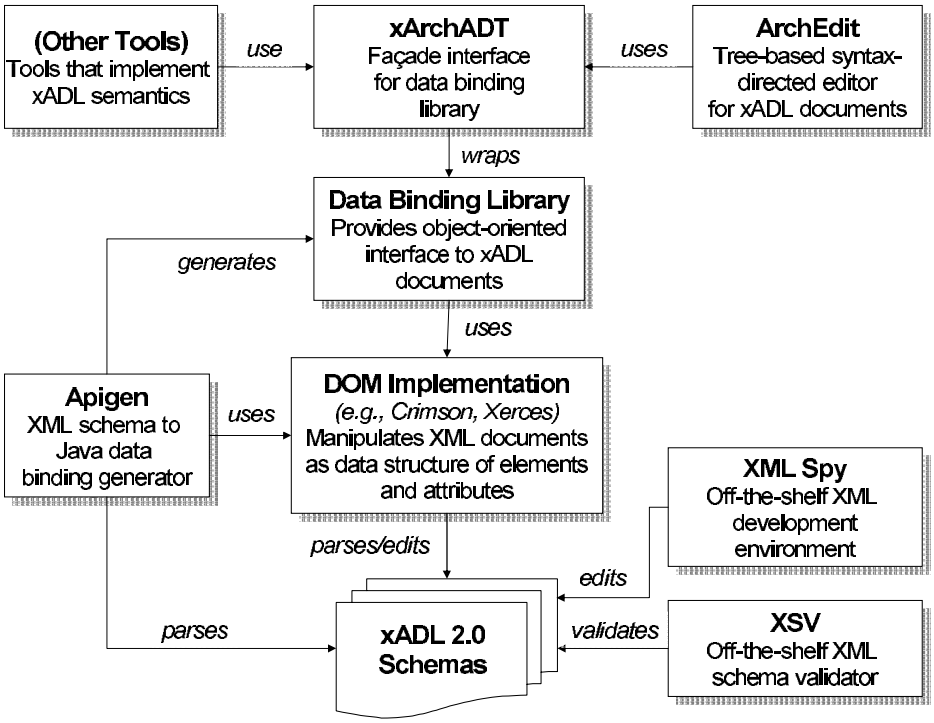


Figure 31. Syntax-driven layer of xADL 2.0 tools and their relationships.

Figure 31 shows the xADL 2.0 syntax-directed tools and their relationships. These tools are described individually in the following sections.

4.2.1 Off-the-Shelf tools: DOM, XSV, XML Spy

At the core of xADL 2.0's syntax-directed tools is an implementation of the DOM (document object model) interface [99]. DOM is an object-oriented interface that allows calling programs to parse, manipulate, and serialize structured documents, particularly XML documents, as a tree of objects. DOM implementations exist on many platforms in many programming languages, with the general DOM API translated into each target programming language as necessary. There are several DOM implementations for the Java programming language, among them Sun's Crimson [157] and Apache Xerces [11].

Several alternative APIs exist for manipulating XML documents, among them SAX [145] and JDOM [69]. SAX is a lightweight parsing API meant for high-performance applications that read, but generally do not manipulate, XML documents. In SAX, a parser reads through an XML document one token at a time and informs a user-provided object (through a callback interface) of each recognized token. This approach does not require more than a small part of the document to be in memory at any given time, but again is useful only for parsing. JDOM provides features similar to DOM, but takes advantage of the specific capabilities of the Java programming language to create a somewhat 'cleaner' interface than traditional DOM. DOM was chosen as the parser of choice for xADL because of its ubiquity (in particular, a DOM implementation ships with every copy of Java beginning with Java 2 SE, version 1.4), as well as the potential for more easily translating existing tools into other languages for which DOM implementations are available.

DOM implementations treat an XML document as a tree of 'node' objects, which can be explored as raw nodes or through subtypes—document nodes, element nodes, attribute nodes, text nodes, and so on. DOM does not take into account any particular schema mechanism (not even DTDs), so all documents are treated uniformly regardless of conformance to a particular syntax of elements and attributes. Names of elements and attributes, for example, are passed as string parameters to DOM calls, and callers are responsible for manipulating documents in conformance with schema rules. For this reason, manipulating a document in a complex, multi-schema, multi-namespace notation like xADL 2.0 is difficult in DOM because the onus is on the programmer to remember and maintain all document structure constraints, namespaces, and so on.

XSV [171] is an off-the-shelf XML schema validator. XSV performs two functions. First, it verifies whether an XML schema (such as a xADL schema) is conformant to the XML schema specification. This helps schema authors write standards-compliant schemas and debug existing schemas. Second, it can validate an instance document (such as a xADL document) against a schema or set of schemas. This is useful for determining whether the document conforms to the syntax specified by the target schemas: errors in validation may cause problems for tools that parse or process the document.

XML Spy [9] is one of many commercial XML 'development environments' that provide an integrated collection of tools for creating XML documents and schemas. XML Spy, for example, allows the development of XML schemas using a convenient tree-based graphical notation with a GUI rather than the complex textual notation normally used. XML Spy also includes tools for validating and transforming instance documents. Because all xADL 2.0 documents are XML documents that conform to a set of known schemas, XML Spy and other XML tools like it can all be used to manipulate xADL documents. In the development of xADL, XML Spy was primarily used to create, edit, visualize, and validate xADL schemas.

4.2.2 Data Binding Library

Off-the-shelf XML tools such as DOM implementations tend to support manipulation of documents in terms of low-level XML concepts like elements, attributes, and text segments. Building tools that work with these low-level constructs directly is cumbersome and error-prone, as it requires the tools themselves to manage

elements like namespaces directly and to ensure that documents conform to XML schemas.

When schemas are available, a more 'friendly' programmatic interface to XML documents can be created based on the language prescribed by the schemas. For xADL 2.0, this interface is provided through a data binding library [41]. The xADL data binding library provides a set of Java classes corresponding to elements and attributes specified in the xADL 2.0 schemas. These classes hide XML details such as namespaces, header tags, sequence ordering, etc. Whereas a generic XML API like DOM exposes functions like `addElement(...)` and `getChildElements(...)`, classes in the data binding library expose functions like `addComponent(...)` and `getAllInterfaces(...)`. Internally, the library uses the DOM implementation provided with Java to manipulate the underlying XML document. The Data Binding Library is automatically generated by another tool in the xADL infrastructure, Apigen, as described in the next section.

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

Figure 32. Excerpt from the xADL Structure & Types schema, defining a component.

Consider the XML definition of a component shown in Figure 32, excerpted from the xADL 2.0 STRUCTURE & TYPES Schema.

```
        void setDescription(IDescription value);
        void clearDescription();
IDescription getDescription();
```

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

Figure 33. Data binding library interface for the class representing a xADL component.

For this type, the data binding library includes a Java class that exposes the interface shown in Figure 33.This demonstrates that, despite having no knowledge of the semantics of the ADL, the data binding library exposes functions that are closer (in terms of their level of abstraction) to the concepts relevant to a software architect. This makes building architecture tools with the data binding library more intuitive and less error-prone than building them with an XML API like DOM. Furthermore, it reduces the number of lines of code necessary to manipulate an XML-based architecture description significantly, by a ratio of approximately 5:1. That is, each call to the data binding library (e.g., addInterface) encapsulates about 5 lines of DOM code.

4.2.3 Apigen

If the data binding library must be rewritten every time a schema is added to, changed, or removed from xADL, then the benefit of having it is diminished. Fortunately, the syntax information present in the ADL schemas is enough to generate the data binding library automatically. xADL's tool suite includes a tool called 'Apigen' (short for "API generator") [41] that can automatically generate the Java data binding library, described above, for xADL 2.0 and extension schemas. When an ADL's

schemas are changed, tool builders simply re-run Apigen over the modified set of schemas to generate a new data binding library. Of course, bindings for elements that did not change will remain the same, minimizing the impact on existing tools that use the library. Because of the complexity of the XML schema language, Apigen is not a generic data binding generator; it does not support the full gamut of constructs available in XML schemas. However, it supports a large set of schema features, and so far has been sufficient to generate data bindings for all the xADL 2.0 schemas as well as schemas written by third parties. A comprehensive list of supported and non-supported constructs in Apigen is included with the Apigen tool's documentation.

Apigen is one of a class of data binding generators for XML schemas, among them JAXB [158], EMF [27] and a proprietary generator included in recent versions of XML Spy. However, at the time xADL was developed, no XML schema data binding generator was in a sufficient state of completion to support the xADL schemas, and this necessitated our internal development of Apigen. With some effort, it would be possible to retire Apigen in favor of one of these later, off-the-shelf data binding generators, although these generators would likely output different interfaces to data binding library objects and data binding library callers (i.e., other tools) would have to be adapted accordingly.

Figure 34. Screenshot of the Apigen wizard interface.

One distinctive feature of Apigen as a data binding library is that it is driven by a relatively user-friendly graphical user interface (see Figure 34). Because this tool is intended for use by end-users, having a simple, straightforward interface is important.

This GUI is organized in a familiar ‘wizard’ manner, offering a sequence of pages containing form elements that the user must populate with configuration data. Apigen’s configuration data is straightforward: it includes the locations (URLs) of each schema, the set of schemas to process, and a destination directory on the local machine where the Java source files for the data binding library will be created. No additional information needs to be provided by the user, such as an additional binding schema. A command-line interface is also available so that Apigen can be integrated into automated build processes.

4.2.4 xArchADT

The data binding library provides a traditional object-oriented interface to edit xADL documents. This requires the library's callers to maintain many direct object references. In general, distributed and event-based systems assume that components do not share an address space, and therefore cannot maintain object references across components. Even in a single-process environment, it is often useful to retain direct references to data within a single component to improve modularity and separation of concerns. Because of this, using such a library as an independent component in a distributed or event-based system is difficult. To address this, xADL's tool suite includes a wrapper, called 'xArchADT,' for the data-binding library that provides a 'flattened' interface instead of an object-oriented one. This 'flattened' interface can easily be exposed over network-based middleware such as CORBA or an event-based interface, since each call passes only atoms (or sets of atoms) of serializable data. This kind of wrapper (wrapping a simple, one-level API around a complex, object-oriented API) can be seen as an instance of the Façade design pattern [59].

Data Binding Library Call

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

xArchADT Call

```
ObjRef componentRef = ...;
ObjRef interfaceRef = ...;

xArchADT.add(componentRef,
  "interface", interfaceRef);
```

Message-based Representation

```
ObjRef componentRef = ...;
ObjRef interfaceRef = ...;

Message{
  Name = "add";
  Source = componentRef;
  Type = "interface";
  Target = "interfaceRef";
}
```

Figure 35. Example of a data binding library call, the equivalent xArchADT call, and a message-based representation.

An example of how a data binding library call is translated into an xArchADT call and expressed as an event data structure is shown in Figure 35. It uses first-class indirect object references (called ObjRefs in xArchADT), rather than direct pointers, to refer to elements in xADL 2.0 documents. That is, xArchADT assigns identifiers to xADL 2.0 elements and those identifiers are used to refer to the elements. When the underlying architecture description is modified by one tool, xArchADT emits an event informing all listening tools of the change. This gives the data binding library the added property of loose coupling.

xArchADT, like Apigen and the data binding library, is reflective. It uses Java's built-in reflection capabilities to adapt to changes in the data binding library automatically. That is, if the library is regenerated by Apigen, xArchADT will work without modification. xArchADT's biggest contribution, however, is that it increases the range of contexts in which the data binding library can be used. The library alone is well-suited for use in tightly-coupled object-oriented system development, but the xArchADT wrapper gives it an interface suitable for remote access across process boundaries (facilitated by middleware) and inclusion in an event-based environment.

4.2.5 ArchEdit

The data binding library and xArchADT expose alternative programmatic interfaces for manipulating architecture descriptions. ArchStudio also includes a syntax-driven tool with a graphical user interface called 'ArchEdit.'

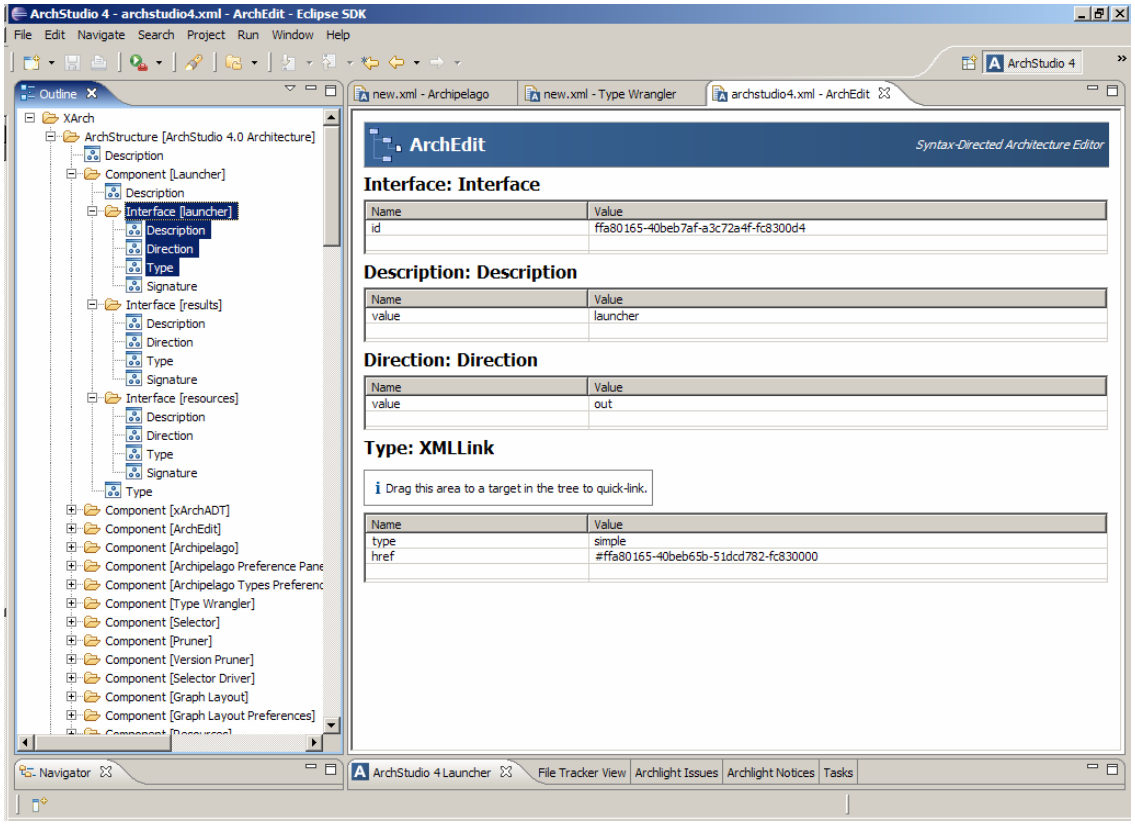


Figure 36. ArchEdit screenshot.

A screenshot of this ArchEdit is shown in Figure 36. ArchEdit depicts an architecture description graphically in a tree format, where each node can be expanded, collapsed, or edited. This is similar to many visual XML editors, except ArchEdit hides the XML details of the document from the user. The user interface is also similar to (and partially inspired by) the Microsoft Windows Registry Editor "RegEdit."

Admittedly, the tree-based interface of ArchEdit is not ideal for creating or editing complex architecture descriptions. However, ArchEdit does have one advantage over other editors: its user interface is guided by the underlying xADL schemas. The schemas direct the structure of the displayed tree view, making the structure of the XML document and the structure of the displayed tree identical. Editing options are also

guided by the schemas—right-clicking[3] on a tree node, for example, brings up a context menu containing options to add or remove elements based on the schemas. ArchEdit also understands xADL's use of internal XLinks, and provides user interface elements for managing them more easily. Because ArchEdit is reflective, it does not need to be changed or updated to provide support for new or changed schemas—the user interface will update itself automatically. With ArchEdit, a user defining a new schema will have a crude but functional low-level GUI-based editor available effectively for free.

There are two disadvantages to ArchEdit's reflectiveness. First, it does not enforce stylistic constraints or other rules on the architecture description that cannot be specified in XML. Second, ArchEdit cannot display the structure of the software architecture in an intuitive way—as a box-and-arrow diagram, for instance. These disadvantages are inherent in any syntax-based tool, but the benefits of having a 'free' editor for new ADL features outweighs these disadvantages. Other semantically-aware editors and analysis tools in ArchStudio address these deficiencies.

ArchEdit serves best as a low-level architecture editing and debugging tool. It gives architects direct access to architecture descriptions without abstracting away details. ArchEdit accesses architecture descriptions through xArchADT. Changes to the architecture description made via xArchADT by ArchEdit or other tools are immediately reflected in the ArchEdit user interface.

---

[3] The term 'right-click' will be used here to refer to the operation that brings up a context menu in an editor. Commonly this is done by clicking with the right mouse button, but the operation is different on some platforms.

4.3 xADL 2.0 Summary and Contributions

The xADL 2.0 architecture description language represents a fundamental rethinking of how software architecture description languages are defined and evolved. Its primary contribution is its modular design, with individual modeling features being separated into modules, written as XML schemas, that are then recomposed into a complete language. Although the set of xADL schemas is constantly evolving through improvements by its developers as well as outside contributions, the diversity and number of XML schemas available provides evidence that this is a suitable approach for the definition of architecture description languages.

The selection of schemas and features that comprise xADL 2.0 in its current state is not random or arbitrary. Each schema supports one or more real xADL stakeholders—researchers, practitioners, and ArchStudio developers (ArchStudio itself is modeled in xADL as described in Section 6.6). The core schemas, particularly the STRUCTURE & TYPES schema, support all users who need structural descriptions of architectures. The product-line schemas were initially developed specifically to support a research group's investigations into product-line modeling and tools, although they were later applied to modeling design alternatives in real systems (see, e.g., Section 6.1.3). The implementation mapping schemas were developed to support a number of different research efforts, allowing automated instantiation and architectural dynamism. The source code implementation schemas were developed for a project investigating the best way to manage the transition from architecture modeling to code-writing activities. The ability to specify statecharts was added to allow research investigations into

behavioral specification of components, but was also done to support a particular capability needed by an industrial modeling project (see Section 6.1.4).

xADL 2.0's accompanying tool suite addresses the challenges imposed in moving from monolithic to modular notations. Reflective tools such as the data binding library (through its generator Apigen), xArchADT, and ArchEdit automatically adapt to new and updated xADL schemas, providing the foundations for more semantically-aware tools. None of these tools do require complex configurations or custom code to process new schemas.

xADL 2.0's capabilities are well-matched to the needs of a stakeholder-centric architecture modeling environment. Users are allowed and encouraged to develop their own extensions, including divergent extensions. XML and XML schemas provide a meta-model for architecture descriptions that is generic and flexible; with the additional support of XLinks, xADL places few limits on what can be expressed in an architectural model. New features can be added to the language incrementally by gradually incorporating more schemas; syntax-directed tools will adapt automatically. Due to XML Schema's subtyping rules, extensions of a particular language still conform to the specifications of the un-extended language, which generally means that other tools will degrade gracefully. Because the time cost to add and compose features in xADL is low, it encourages end-user experimentation. Finally, because features are defined in modular schemas and dependency relationships between schemas are explicit, it is possible to reuse schemas from one project to another (as long as appropriate dependencies are satisfied).

4.4 Archipelago: ArchStudio's Extensible Graphical Editor

All architecture modeling notations offer at least one canonical visualization that depicts the information in the model in a particular way, chosen by its developers. When embodied in editor tools, these visualizations offer interfaces through which users can interact with those models. For xADL 2.0, that canonical visualization is textual: it is the raw XML form of the xADL model. XML files are specially formatted text files and can be edited using any text editing tool. However, as has been discussed, viewing or editing xADL documents in this form is cumbersome and error-prone: a large amount of 'housekeeping' data in the form of XML namespace data is present, and xADL makes extensive use of XLinks to allow one element to refer to another by means of a matching identifier. Syntax-directed editors such as ArchEdit alleviate this problem somewhat, hiding the housekeeping data and making it easier to traverse and maintain XLinks. However, the tree-based data structure offered by XML is not well-matched to the sort of mental models that software architects use when conceiving or discussing architectures.

Many architecture description languages, for example Darwin, Koala, and UML, provide or endorse a graphical visualization that captures some or all of the information in the architecture description in a graphical format. In general, these depictions can generally be described as being primarily composed of graphical symbols, annotated with textual labels and other textual decorations. Examples of such depictions are shown in Figure 2, Figure 10, and Figure 14. As personal computers have gotten increasingly powerful, editing tools for such graphical depictions have become commonplace—so much so, in fact, that certain user interface paradigms for editing

them are uniform even across competing applications. Examples of such editors include Microsoft PowerPoint [112], Microsoft Visio [111], OmniGraffle [166], and Rational Rose [131]. Some of these tools, such as PowerPoint, are solely drawing tools, and the diagrams they create have no connection to an underlying, semantically rigorous information model such as an architecture description. Some, like Rational Rose, are programmed with knowledge of a particular notation's constraints and their user interfaces help to guide users in adhering to that notation. Tools like Visio are somewhere in between—they can be used as a semantics-free drawing tool, or scripts/extension modules can be incorporated to add semantic knowledge and user-interface extensions.

Graphical editors are themselves complex software systems and, as such, are often expensive to develop, maintain, and extend. For this reason, notations can go for months or years without graphical editing support, and some notations never acquire graphical editing support at all. It is safe to assume that one reason the developers of AADL and SysML are interested in creating UML profiles for their notations is the ubiquity of industrial-quality UML-based graphical editing tools and the expense of developing their own. Efforts to integrate large commercial off-the-shelf diagram editors into architecture-centric development environments have been attempted—examples include the PowerPoint Design Editor [63] and Visio for xADL [134], although these met with limited success. Such environments offer a plethora of features (not all of which are supportive of architectural modeling) and industrial support, but offer limited means for extension by end-users. Limitations in these extension

mechanisms, in addition to the difficulty in keeping up with updates and changes to the underlying products, ultimately limited the effectiveness of these efforts.

Constructing a graphical editor for relatively static, monolithic underlying modeling notations is difficult enough. Building such an editor for a modular notation like xADL 2.0, in which end-users are expected to extend the notation in potentially unforeseen ways, adds another layer of difficulty. Here, the graphical editor must evolve modularly as well—in step with changes to the underlying notation—as well as provide guidance for end-users as to how to extend the environment. It should be extensible at multiple levels, allowing the addition of fundamentally novel features as well as extensions of existing features at lower cost. Ideally, it should allow developed editing features to be reused in different contexts at little or no cost, allowing the investment in developing editing support for features to be amortized. It must also support connection to an independent, back-end model containing the architectural model—in this case, to the xArchADT component.

In light of these concerns, as well as the lessons learned from previous graphical editor construction projects such as Visio for xADL and ArgoUML [2, 139], a new graphical editor was constructed for ArchStudio, called Archipelago[4]. The key insight behind the construction of Archipelago is that each aspect of the environment is

[4] The name "Archipelago" was chosen for several reasons. It starts with the prefix 'arch,' as with ArchStudio and its constituent tools. An archipelago is a chain or grouping of independent islands, and this was meant to be a metaphor for both the internal structure of Archipelago (independent but related modules) as well as for the kinds structural architecture diagrams that resemble 'islands' (components and connectors' loosely coupled by thin links.

constructed from small modules built upon a lightweight, flexible core. This core imposes few constraints on how the editor behaves or can be constructed. The constraints that are imposed are consistent with similar graphical editors—a tree on the left provides coarse-grained entrypoints into finer-grained editors in the main pane. Specific additional constraints are described below.

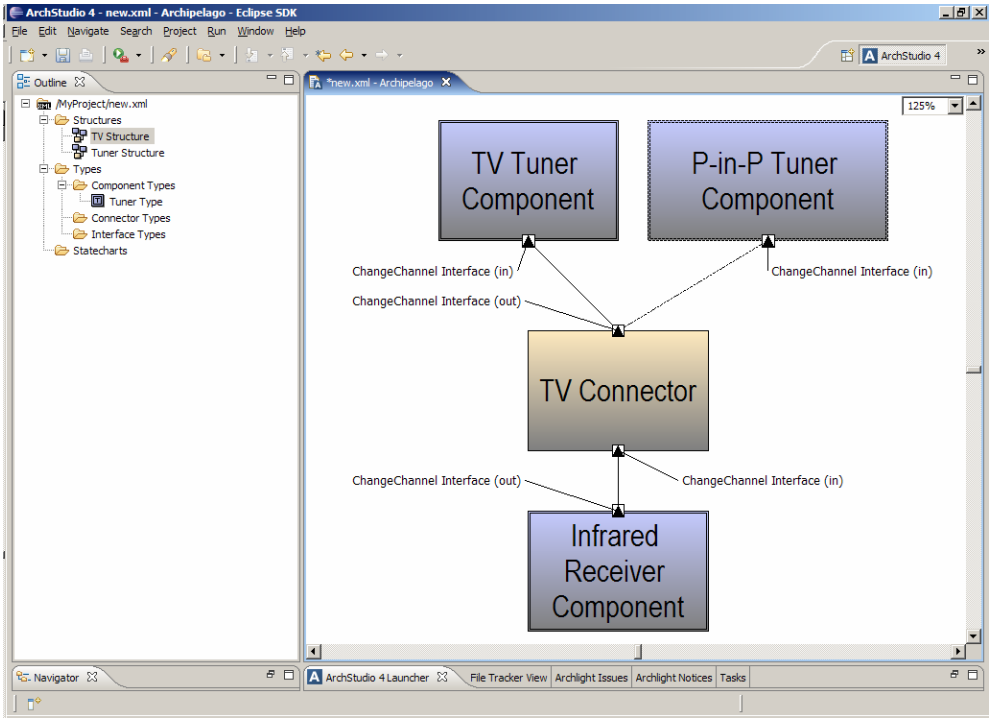


Figure 37. Archipelago screenshot.

Figure 37 shows a typical screenshot from Archipelago. Archipelago is divided into two primary user interface elements: the Archipelago tree on the left and the Archipelago editor pane on the right. The tree shows a subset of elements in the architecture—usually top-level elements such as structures and types, organized hierarchically. Visualization in Archipelago always begins from the tree: when a xADL document is opened in Archipelago, the tree is populated immediately, but the editor pane remains empty with only a placeholder. The user begins editing by interacting with

the tree. One way to do this is by right-clicking on a node in the tree and editing the tree nodes through context-menu options. More commonly, the user will double-click on a node in the tree. This generally causes Archipelago to open an appropriate editor in the editor pane on the right side.

4.4.1 Archipelago Internals Overview

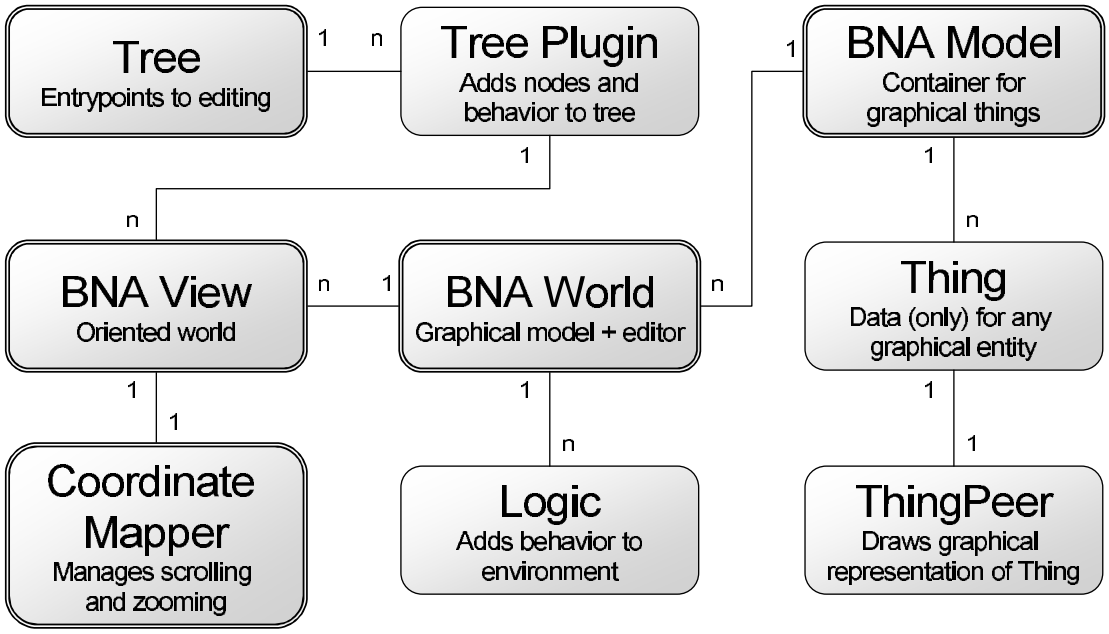


Figure 38. Entity-relationship diagram of Archipelago core elements.

Figure 38 shows an entity-relationship diagram depicting the elements of the Archipelago core and their relationships. In general, each element corresponds to one or more Java classes in the Archipelago implementation. Elements with a double-border have default implementations provided as part of Archipelago and are generally not changed or extended by ArchStudio users (although there is nothing preventing them from doing so). Elements with a single border are those that are intended to be extended

by users. As is obvious, the core is quite small—only nine kinds of elements, with only four of these being extension points. Each element of the core is described below.

4.4.2 The Archipelago Tree

As noted above, the Archipelago tree is the ‘starting point’ for using Archipelago—it is populated with data from the architecture when Archipelago starts, and the tree is used to open more detailed editors. The tree itself is simply an ordinary GUI tree widget, supporting all the expected features of such a widget: textual nodes, with descriptive icons, some of which are containers for child nodes that can be expanded or collapsed by pointing and clicking. Context menus are available by right-clicking on the nodes.

By default, the tree is not populated with any nodes or behaviors at all. Instead, all data and behavior of the tree is added by a set of Archipelago tree plug-ins, each of which has access to the entire tree.

```
public interface IArchipelagoTreePlugin{
      public IArchipelagoTreeContentProvider getContentProvider();
      public IArchipelagoLabelProvider getLabelProvider();
      public IArchipelagoTreeDoubleClickHandler getDoubleClickHandler();
      public IArchipelagoTreeContextMenuFiller[] getContextMenuFillers();
      public ICellModifier[] getCellModifiers();
      public XArchFlatListener getXArchFlatListener();
      public XArchFileListener getXArchFileListener();
      public IFileManagerListener getFileManagerListener();
      public IArchipelagoEditorFocuser getEditorFocuser();
      public DragSourceListener getDragSourceListener();
}
```

Figure 39. Archipelago tree plug-in interface.

The interface for an Archipelago tree plug-in is shown in Figure 39. This interface is primarily called by Archipelago when constructing the tree at startup. All methods in the interface are ‘get’ methods, each returning one or more objects that

provide a service to Archipelago. Each tree plug-in may provide as many or as few of these service objects as needed. They include:

- **Content Provider:** Provides data about new nodes and structure to the tree.
- **Label Provider:** Provides information on icons and textual representations of nodes reported by the content provider.
- **Double Click Handler:** Determines what happens when a user double clicks on a node in the tree.
- **Context Menu Fillers:** Determine what happens when a user right-clicks on a node in the tree.
- **Cell Modifiers:** Determine what happens when a user attempts to directly edit (e.g., by clicking and typing) a node in the tree.
- **Flat Listener:** Determines how the tree should react to changes in the underlying xADL model from xArchADT. Also passes these events to the open editing pane, if necessary.
- **File Listener:** Determines how the tree should react to changes in the file state of ArchStudio (files being opened, closed, or renamed, for example). Also passes these events to the open editing pane, if necessary.
- **File Manager Listener:** Determines how the tree should react when the user performs a save operation in the file manager.
- **Editor Focuser:** Determines how the tree should react when the user has requested Archipelago to focus on one particular element in the architecture.

- **Drag Source Listener:** Determines how the tree should react when the user begins a click-and-drag operation starting from the tree.

Roughly speaking, each xADL schema should be accompanied by a set of tree plug-ins that add data and functionality to the tree corresponding to what is expressed in the xADL schema. Not all schemas will have tree plug-ins (if they only contribute to the editor pane, for example), while some schemas may have multiple tree plug-ins. For example, the STRUCTURE & TYPES schema is associated with two tree plug-ins: one for adding structures to the tree and one for adding types.

4.4.3 BNA Internals Overview

Double-clicking on a tree node that represents, for example, a structure or a type, generally causes the Archipelago editor pane to display a graphical editor for the element that was clicked. Actual graphical editing responsibilities are handled by a subset of the Archipelago core: the seven elements in Figure 38 that are not the Tree or Tree Plug-ins make up this sub-framework, called BNA[5]. Functionally, BNA resembles many other graphical editing frameworks such as GEF [81] and JGraph [92]. Although it would have theoretically been possible to use one of these existing frameworks to build the graphical editing portion of Archipelago, the BNA framework offers advantages in terms of flexibility and extensibility. The BNA core makes very few assumptions about the type of graphical editing being implemented. It assumes:

[5] BNA stands for 'Boxes'N'Arrows,' a somewhat tongue-in-cheek way of referring to the kind of symbolic editors that resemble Archipelago.

- The editing context is a very large two-dimensional canvas that can be navigated by scrolling and zooming;
- The canvas itself is initially blank, then populated by a set of graphical entities;
- These graphical entities have data associated with them;
- These graphical entities may or may not have a visual representation on the canvas tied to their data;
- These entities have ordering relationships that determine the order in which they are drawn, allowing one entity to appear to be ‘stacked’ atop another, by virtue of it being drawn afterwards (and therefore on top of) other entities occupying the same space in the canvas;
- The user will interact with these entities and the canvas through the use of the mouse and keyboard.

These assumptions are in line with nearly any comparable graphical editing tool. BNA is not infinitely flexible: it would be a poor choice for a three-dimensional editor or a text editor. However, within its limits, the framework makes no assumptions about the kinds of entities, how they are represented, or how the editor responds to stimuli (from the user, from other applications, or from internal changes to entities). It is this feature that makes BNA extensible above and beyond alternatives.

4.4.4 BNA Things

The core data element in BNA is known as a Thing. A BNA Thing contains all the data needed to render and interact with a graphical element. Everything drawn on a BNA editing canvas—the boxes representing components and connectors, the splines

representing links, the text labels annotating other elements, the grid that is used to organize and align elements, and even the ‘dancing ants’ animated marquee box that appears when clicking and dragging to select multiple elements are captured internally as Things.

```
public interface IThing extends java.io.Serializable, Cloneable{

	//Indicates Thing’s peer class
	public Class<? extends IThingPeer> getPeerClass();

	//Duplicate a Thing
	public Object clone() throws CloneNotSupportedException;

	//Thing’s ID
	public void setID(String id);
	public String getID();

	//Used for accessing property data in a thread-safe way
	public Object getPropertyLockObject();

	//IThingListeners detect and react to changes to this Thing
	public void addThingListener(IThingListener tl);
	public void removeThingListener(IThingListener tl);

	//Set, query, and remove properties
	public void setProperty(String s, Object value);
	public <T> T getProperty(String name);
	public boolean hasProperty(String name);
	public void removeProperty(String name);

	//Manipulate properties where the value is actually a set
	public void addSetPropertyValue(String name, Object value);
	public void replaceSetPropertyValues(String name, Set s);
	public void removeSetPropertyValue(String name, Object value);
	public Set getSetProperty(String name);

	//Get all properties and values in a Map
	public Map getPropertyMap();

	public String[] getAllPropertyNames();
}
```

Figure 40. Interface for a BNA Thing.

A Thing can be any Java object that implements the `IThing` Java interface, shown in Figure 40. However, a default implementation of a Thing class is provided as part of the BNA framework; this can be used as-is or it can be extended through Java

inheritance. As should be obvious from the figure, the Thing data structure is straightforward: effectively a Thing consists of a unique string identifier and a property map. Each property maps a String name to an arbitrary object containing data. Typical properties that would be found on a BNA Thing might include a bounding box stored as a Rectangle data structure, a color stored as an RGB triple, or a label stored as a character string. Some convenience methods are also provided for dealing with properties that may have a set of values, but these effectively wrap the `setProperty` and `getProperty` methods.

```
public void setColor(RGB c){
      setProperty(COLOR_PROPERTY_NAME, c);
}

public RGB getColor(){
      return getProperty(COLOR_PROPERTY_NAME);
}
```

Figure 41. A typical getter-setter pair found in a subclass of BNA's default Thing implementation.

It is common for subclasses of the default Thing implementation to provide getter/setter pairs for specific properties to avoid callers having to remember the String name of each property; an example of this is shown in Figure 41. Note that the implementation for these methods is usually a one-line call to the generic `getProperty` or `setProperty` method. Additionally, since many Things may share a similar property (e.g., color, as in the example above), these getter/setter interface methods are often grouped into small interfaces implemented by all Things that have that property. This interface, `IHasColor` in this case, is known in BNA as a facet, and it allows behavioral Logics (described below) to reason about 'all Things that have a color' by indexing all Things that implement the `IHasColor` facet.

Thing objects are intended to act solely as data structures: they do not implement any application behavior. They do, however, also notify a list of Thing Listeners whenever any Thing property changes. In general, all Thing events are aggregated by the BNA Model (see below) and emitted as part of BNA Model Events. Thing objects are not intended to contain any other Thing objects (as property values, for example). Things can instead refer to other Things by including the target Thing's unique ID as a property value.

4.4.5 BNA Thing Peers

A thing is just a collection of data; it does not have an inherent visual representation on the BNA canvas. It may contain data about where and how it should be represented, but it is the job of the Thing's associated ThingPeer to actually draw the Thing on the canvas and report on its location. Each Thing is associated with exactly one peer. ThingPeer classes may be idempotent—that is, completely stateless. If all peers were stateless, all Things of the same type could use the same ThingPeer object. However, for efficiency reasons, ThingPeer objects often cache data such as results of previous computations, and as such a different ThingPeer is used for each Thing.

```
public interface IThingPeer{
      public void draw(IBNAView view, GC g);
      public boolean isInThing(IBNAView view, int worldX, int worldY);
}
```

Figure 42. The BNA ThingPeer interface.

The BNA ThingPeer interface, shown in Figure 42, is remarkably simple, consisting of only two methods. The first, `draw`, is called by Archipelago to draw the Thing on the canvas. The second, `isInThing`, queries a given point in the canvas' world (more on local and world coordinates below) to determine if that lies within the

boundaries of the Thing or not. The `isInThing` method is primarily used to determine whether a mouse click or action ‘hit’ a given Thing. As such, `isInThing` is specified such that peers should return ‘true’ if a user clicking a mouse on that point would reasonably expect to have hit the Thing. Points just outside the Thing’s drawing area may still return ‘true’ if the Thing has an especially small boundary, such as a spline with a thickness of one pixel.

The Thing-ThingPeer distinction clearly separates the internal graphical model from its representation—by replacing a Thing’s peer, the Thing’s graphical representation could be altered completely without changing the Thing at all.

4.4.6 BNA Models

Things existing on the same canvas are grouped into a BNA Model, a data structure that organizes Things. The primary purpose of the BNA model is to maintain the collection of Thing objects, and to organize the stacking (that is, drawing) order of Things.

```
public interface IBNAModel{

      //Model reading
      public IThing getThing(String id);
      public IThing getParentThing(IThing t);
      public IThing[] getAncestorThings(IThing t);
      public IThing[] getChildThings(IThing parent);
      public IThing[] getAllThings();
      public int getNumThings();

      public java.util.Iterator getThingIterator();
      public java.util.ListIterator getThingListIterator(int index);

      //Model updating
      public void addThing(IThing t);
      public void addThing(IThing t, IThing parentThing);
      public void removeThing(String id);
      public void removeThing(IThing t);
      public void removeThingAndChildren(IThing t);
```

```
        //Stacking order
        public void stackAbove(IThing upperThing, IThing lowerThing);
        public void bringToFront(IThing thing);
        public void sendToBack(IThing thing);

        //Model event listeners
        public void addBNAModelListener(IBNAModelListener l);
        public void removeBNAModelListener(IBNAModelListener l);
}
```

Figure 43. Selected methods from the BNA Model interface.

A portion of the BNA Model interface is shown in Figure 43. The interface includes a set of methods for querying the model and a set of methods for updating the model by adding or removing Things. A small set of methods is used for reordering the stacking order of Things, and methods are also provided for outside objects to register to be notified of any changes in the BNA Model. A BNA Model emits an event whenever a Thing is added or removed from the model, or whenever an internal Thing emits a change event.

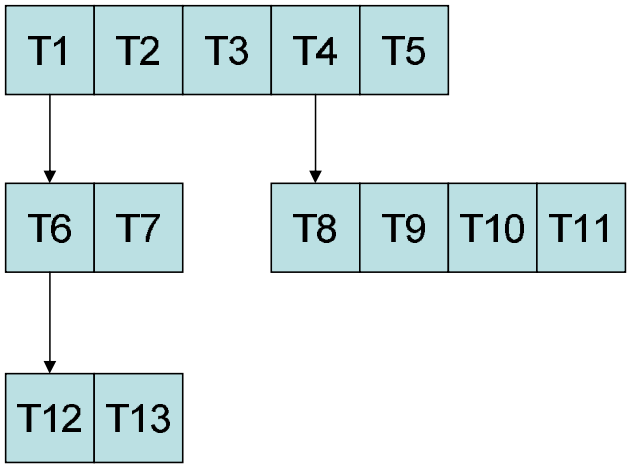


Figure 44. Organization of things in the BNA Model as a 'tree of lists.'

Things in the model are organized as a tree of lists of Things. Each list contains Things that are at the 'same level' in terms of stacking order; they are drawn in order from the front to the back of the list. Each list can be reordered using the

`stackAbove`, `bringToFront`, and `sendToBack` methods of the BNA Model. Additionally, each Thing may have a pointer to a child list, which is a list of additional Things intended to be drawn directly atop that Thing. This ensures that, for example, a box label is always drawn directly atop a box; it is not possible to stack the box in front of its own label. A hypothetical BNA Model structure is shown in Figure 44. Drawing is done in a depth-first manner. The drawing order for Things in this model would be: T1, T6, T12, T13, T7, T2, T3, T4, T8, T9, T10, T11, T5. Being drawn first, T1 is in back of all other Things; T5 is on top of all other Things. Obviously, if two things do not occupy the same pixels on the canvas, drawing order is irrelevant.

4.4.7 BNA Assemblies

Assemblies are not core classes in BNA, but they do tend to be pervasive in Archipelago. Assemblies are objects that can be used to index into subtrees of the BNA model. It is often the case that a single conceptual entity, such as a component, will be rendered by many individual Things.

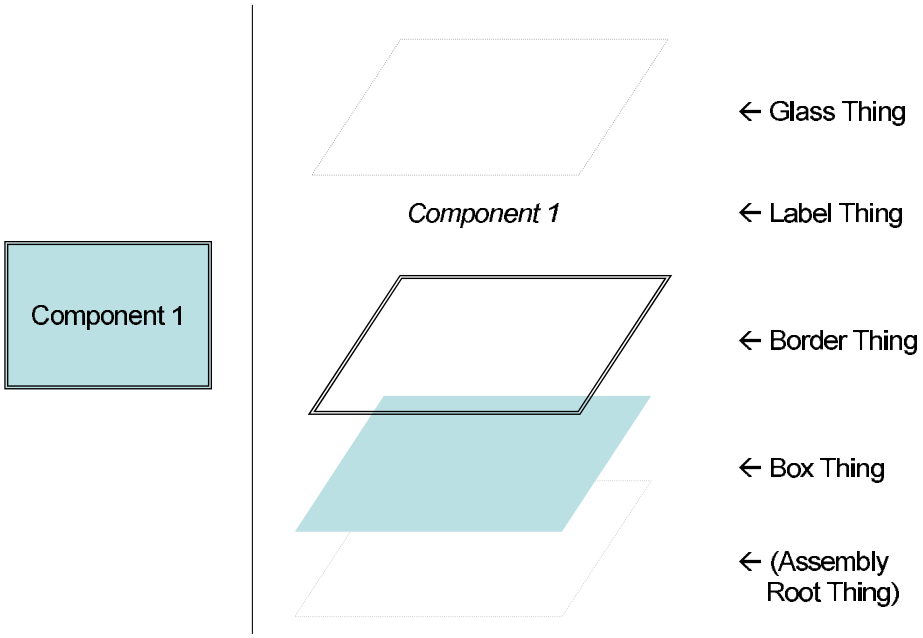


Figure 45. A component in Archipelago being rendered as a co-located stack of BNA things.

This situation is depicted in Figure 45. The left side of the figure shows a component as it might appear on the Archipelago canvas. Although this may look like it is rendered by a single Thing, in reality it is rendered by a stack of Things, as shown on the right. At the bottom of the stack is the assembly root Thing, a Thing that has no visual appearance (that is, its peer does nothing). It is a placeholder in the BNA Model Tree; all other Things in the stack are its children. Next in the stack is the box Thing, drawn as a rectangular field of color. Then comes a border Thing, which draws a decorative border around the field of color—for a component, a double line is used. Then comes the label Thing, which render's the component's descriptive text. On the top of the stack is the glass Thing. The glass Thing has no visual appearance, but it does respond positively to `isInThing` calls that occur within the component's border. The use of a glass Thing on top of all other Things ensures that mouse clicks anywhere on the component—the border, the label, or the colored field, are all caught by the same

Thing (the glass Thing) to be handled uniformly. The glass Thing also acts as a bounding box master for all the lower Things. Behavioral Logics, described below, ensure that when the glass Thing's bounding box is moved, all the lower Things' bounding boxes are moved in concert. This gives the illusion of the component being a single Thing, when in reality it is a connected stack of them.

This separation makes BNA Things (and associated peers) more extensible and reusable. For example, a label Thing does not necessarily have to label a box—it could also be used to label an ellipse or a spline. The same border Thing is used to render both box borders as well as the rubber-band marquee box used for click-and-drag group selection. If the label, border, and colored field were all grouped into one Thing, the code managing each part of the Thing would need to be duplicated in any other Thing that included either a border or a label.

Unfortunately, keeping track of all the related constituents of one conceptual element, such as a component, is tricky. To assist this process, BNA Assemblies act as indices into part of the BNA Model tree. Given the ID of the assembly root Thing, an Assembly will quickly locate and index all the other Things that make up the component. Assemblies can also be used to create the stack of Things that make up a conceptual element like a component.

The need for Assemblies in BNA is largely an artifact of BNA's implementation in Java, a programming language that does not natively support multiple inheritance or mixins [153]. In a programming language with mixin or multiple inheritance support, it would be trivial to compose each Thing class into a single ComponentThing using

inheritance—the ComponentThing would simply be the composition of all constituent classes.

4.4.8 BNA Logics

Things, ThingPeers, Assemblies, and Models provide the infrastructure necessary to describe a group of graphical entities and draw them on a canvas. However, this is just half of a visualization. Recall that the definition of visualization includes both *depiction* (provided by Things, ThingPeers, and Models) and *interaction.* Interaction refers to the 'user interface' through which users interact with depictions.

Behavioral aspects of a BNA-based editor like Archipelago are handled by a set of objects known as BNA Logics. A Logic is an object that responds to external stimuli. These stimuli can come in the form of mouse input, keyboard input, changes in application state (e.g., loss of window focus), messages from other components in the environment, internal changes within a Thing model, and so on. BNA Logics can respond to these stimuli in any way appropriate. Many will update Things in the BNA model (to move something, for example, by changing its bounding box), which will cause new model events to be emitted, which can be picked up by other Logics, and so on in a chain reaction. Others will pop up context menus, invoke external services, call xArchADT and update underlying xADL models, and so on.

```
public interface IThingLogic extends IBNAModelListener,
                                     ISynchronousBNAModelListener{

      //Install/uninstall this logic within a World
      public void setBNAWorld(IBNAWorld bnaWorld);
      public IBNAWorld getBNAWorld();

      //Query which event types are handled by this logic
      public int getHandledEventTypes();

      //Untyped events
```

```
        public void handleEvent(IBNAView view, Event event);

        //Mouse events
        public void mouseUp(IBNAView view, MouseEvent evt, IThing t,
                            int worldX, int worldY);
        public void mouseDown(IBNAView view, MouseEvent evt, IThing t,
                              int worldX, int worldY);
        public void mouseClick(IBNAView view, MouseEvent evt, IThing t,
                               int worldX, int worldY);
        public void mouseDoubleClick(IBNAView view, MouseEvent evt,
                                     IThing t, int worldX, int worldY);

        //Mouse move events
        public void mouseMove(IBNAView view, MouseEvent e, IThing t,
                              int worldX, int worldY);

        //Mouse track events
        public void mouseEnter(IBNAView view, MouseEvent e, IThing t,
                               int worldX, int worldY);
        public void mouseExit(IBNAView view, MouseEvent e, IThing t,
                              int worldX, int worldY);
        public void mouseHover(IBNAView view, MouseEvent e, IThing t,
                               int worldX, int worldY);

        //Key Events
        public void keyPressed(IBNAView view, KeyEvent e);
        public void keyReleased(IBNAView view, KeyEvent e);

        //Focus Events
        public void focusGained(IBNAView view, FocusEvent e);
        public void focusLost(IBNAView view, FocusEvent e);

        //Menu events
        public void fillMenu(IBNAView view, IMenuManager m,
                             int localX, int localY, IThing t,
                             int worldX, int worldY);

        //Drop events
        public void dragEnter(IBNAView view, DropTargetEvent event,
                              IThing t, int worldX, int worldY);
        public void dragOver(IBNAView view, DropTargetEvent event,
                             IThing t, int worldX, int worldY);
        public void dragOperationChanged(IBNAView view,
                                         DropTargetEvent event, IThing t,
                                         int worldX, int worldY);
        public void dragLeave(IBNAView view, DropTargetEvent event,
                              IThing t, int worldX, int worldY);
        public void dropAccept(IBNAView view, DropTargetEvent event,
                               IThing t, int worldX, int worldY);
        public void drop(IBNAView view, DropTargetEvent event, IThing t,
                         int worldX, int worldY);
}
```

Figure 46. BNA Logic interface.

Figure 46 shows the interface implemented by all BNA logics. Logics do not have to implement behavior for each method; most methods will be empty for any given Logic. The various external stimuli to which a logic can respond are obvious: mouse events, keyboard events, drag and drop events, requests to pop up a context menu and so on. This interface also extends `IBNAModelListener`, so it will receive model events as well. Individual Logics can also choose to add themselves as listeners to external events—those coming from xArchADT, for example.

Logics can store internal state, and can also announce state changes to ther logics. Archipelago includes several internal tracking logics that keep internal indexes on various Things. For example, one Logic keeps track of the state of all things that have their 'selected' property set to true. Whenever the set of selected Things changes, an event is emitted by the selection tracking Logic. Other logics can also query the selection tracking Logic for the current set of all selected Things and get a response quickly, since they do not have to iterate through the entire BNA Model looking for all selected Things. Another set of Logics keeps properties among related Things consistent. For example, as noted above, there is a logic whose purpose it is to watch for changes in a glass Thing's bounding box property. When the glass Thing's bounding box changes, the logic changes the associated border Thing, label Thing, and box Thing's bounding boxes to match.

Because Logics are reactive and event-based, the Logics themselves form a sort of internal implicit invocation architecture [60, 133] within Archipelago. Long and complex event chains are possible, and in fact routine. For example, consider the action of Archipelago when a user clicks on a component and drags it across the canvas one

pixel to the right. First, a `DragMovableLogic` catches the mouse-down event on the box's glass Thing. It then stores a bit of internal state indicating that the mouse is currently down on that particular glass Thing. Then, the user moves the mouse slightly. The `DragMovableLogic` receives this event as well, and from the previous event it knows that the topmost Thing currently under the mouse is the component's glass Thing. It then changes the component's glass Thing's bounding box and moves it one pixel to the right to match the mouse movement. Seeing this, a `MirrorBoundingBoxLogic` responds by looking in its pre-computed index of all Things whose bounding box master is the moved glass Thing. One by one, it updates the border, label, and box Things' bounding boxes to match. If the component stands alone in the architecture, then this is the end of the event chain. However, consider a component that has interfaces along its sides, as shown in Figure 24. These too are Assemblies, very similar to the component's Assembly—they have a box Thing, a border Thing, a label Thing (albeit showing triangular pointers instead of text), and a glass Thing. These must also be moved to keep them attached to the edges of the component. This is done by the `BoundingBoxRailLogic`. This logic only moves the glass Things, however. Another trip through the `MirrorBoundingBoxLogic` is needed—this time for the interfaces—to update all the constituent Things for each interface. Things become yet more complicated if the interface is the endpoint for an architectural link, which must also be updated in a further cascade. All these events happen very rapidly, and often tens or hundreds of events per second can occur. This requires logics to be written with efficiency and memory footprint in mind.

For all the complexity of the event chains that occur, the Logics themselves tend to be rather simple. They each have small, well-defined jobs, and usually act uniformly over a given type of Thing. For example, the selection tracking Logic acts on all Things with the `IHasSelected` facet, and the `MirrorBoundingBoxLogic` acts on all Things with the `IMirrorsBoundingBox` facet. The `MirrorBoundingBoxLogic` and the `BoundingBoxRailLogic` comprise only about 100 lines of code each.

The pervasive use of facets both in defining Things and implementing Logics is key to their small size, but is also key to the reusability of Logics. If a user wants to create a new kind of Thing that mirrors the bounding box of another Thing or stays attached to the edge of another Thing's bounding box, then the new Thing need only implement the `IMirrorsBoundingBox` or `IHasBoundingBoxRail` facet and the existing Logics will act on the new Thing appropriately, with no new Logic code written.

4.4.9 BNA Worlds, Views, and Coordinate Mappers

A BNA Model and a set of BNA Logics nearly comprise a graphical editor: both depiction (through Things and their associated peers) and interaction (through Logics) are present. These elements make up what is known as a BNA World. Some additional functionality is needed to render this world on a computer screen, however. Recall that one of the fundamental assumptions of BNA is that the editing context is a very large two-dimensional canvas that can be navigated by scrolling and zooming. BNA creates a 'virtual canvas' in arbitrary dimensions specified by the user—hundreds of thousands of pixels wide/tall. Of course, even on today's modern PCs, storing an image of that size

would take up too much memory. Instead, whenever the canvas needs to be painted onscreen, only the (relatively small) portion of the canvas actually visible in the window, set by the current scroll position and zoom level, is rendered. Coordinates stored in Thing data—bounding boxes, for example—are expressed with respect to the virtual canvas, irrespective of their location on the screen. Thus, scrolling or zooming around the canvas does not change the internal Things at all. Because these coordinates are relative to the internal BNA World, these coordinates are known as world coordinates. With knowledge of the current scroll position and zoom level, any world coordinate can be translated into a local coordinate (e.g., onscreen window coordinate) and back using simple math. This translation is done using an object called a Coordinate Mapper. Every ThingPeer has the responsibility, when drawing a Thing, to translate the Thing's world coordinates into local coordinates, and render the Thing in the appropriate place on the screen. Peers use the coordinate mapper to do this. The Coordinate Mapper is also used to turn local coordinates, such as those that occur within mouse events, into world coordinates. This can be done on-demand, but for efficiency is often done automatically and the calculated world coordinates are passed into Logics directly to save each Logic from having to make the same calls on the Coordinate Mapper over and over again.

One advantage of this approach is that individual peers are aware of element sizes and shapes in both local and world coordinates. This allows individual peers to adjust their own depiction to fit the current zoom level in ways that are not possible with simple bitmap scaling algorithms. For example, the peers for labels and internal diagrams will often choose not to render anything, or render an abstract depiction (such

as a set of horizontal lines) when the zoom level is too low to display meaningful data. These capabilities have been only initially explored in BNA and can likely be exploited further in the future.

A BNA World, plus an associated Coordinate Mapper to orient the world within the context of the computer screen, comprises an object called a BNA View. Generally, one BNA view is used to fill the entire editor pane of the Archipelago editor. However, the use of Views and Coordinate Mappers has a useful and elegant consequence: by changing the Coordinate Mapper's internal parameters, it is possible to render a world in part of a window, or to render one world within another. This is the mechanism used to render the nested diagrams shown in Figure 25. Moreover, the same World can be embedded in multiple Views simultaneously with no adverse effects. Thus, the same World can be rendered in two places at once, and changes to one are immediately reflected in the other. Thus, if a user edits the internal diagram within the left hand component in Figure 25, the diagram on the right will be changed in lock-step. Aside from some basic scaffolding to maintain the Coordinate Mappers' internal settings provided as part of the BNA framework, logics do not have any awareness that this is occurring. Thus, the same diagram, or parts of the same diagram, can be displayed in multiple windows simultaneously, as subdiagrams of two parts in the same window, and so on. This provides a powerful editing interface for hierarchical or nested structures that are common in software architecture descriptions.

4.5 Archipelago Summary and Contributions

Archipelago addresses the challenges of stakeholder-driven, multi-view architecture visualization using principles similar to those used in xADL 2.0—dividing

functionality into interdependent modules and recomposing those modules into a usable whole—in this case, a visualization environment for a xADL 2.0 architecture. As a side effect, Archipelago plug-ins can be modularized along the same lines as xADL 2.0 schemas, so it is possible to develop a set of Archipelago plug-ins that can be deployed alongside a xADL schema that provides visualization and graphical editing support for that particular schema.

Each aspect of the editor—how graphical elements are represented internally, depicted onscreen, and how the editor behaves—is defined using a composition of small modules. Adding or changing a few of those modules can drastically change the Archipelago environment. For example, a complete basic statechart editor was implemented inside Archipelago, reusing elements from the architectural structure editor, in less than 2000 lines of code.

No Archipelago plug-ins are privileged over any other. Every behavior in Archipelago, from mouse-wheel zooming to drag selection to context menu creation, is done entirely by a set of plug-ins that implement the same basic, stable Logic interface. This gives Archipelago users enormous freedom to tailor the depiction and editing of architecture descriptions to their own needs and mental models. Archipelago can be used to create modal and modeless, tool-based and context-menu-driven editors with equal facility.

Small plug-in modules inside Archipelago promote pervasive reuse. As noted above, Things such as labels and borders can be reused in many contexts. BNA Logics' use of facets to define behavior over classes of Things rather than individual Thing types adds an aspect-oriented [97] flavor to the environment—instead of implementing

a behavior such as drag-movability individually for ten or fifteen different kinds of objects, it is implemented once and works uniformly across all Things that are tagged with the drag-movable facet.

4.6 Archlight: ArchStudio's Extensible Analysis Framework

Unlike natural language descriptions or arbitrary symbolic diagrams (such as those created in Microsoft PowerPoint), xADL documents have a rigorous syntax and structure. Beyond this, elements in xADL have intended semantic interpretations and relationships. Syntactic verifiers such as XSV and other XML schema validators cannot verify these. Because xADL is rigorous and machine-readable, tools can be used to check some of these properties automatically. Many architecture description languages, such as Wright and AADL, offer such analysis tools as primary motivators for adopting them.

Providing analysis capabilities for a modular language like xADL 2.0 introduces challenges similar to those encountered by Archipelago: how can an environment provide analysis tools in a context where the underlying notation, including its syntax and associated semantics, may be changed by end-users. Archlight, ArchStudio's extensible analysis framework, uses a top-level strategy similar to Archipelago's: modularize the analysis capabilities in the same way that xADL itself is modularized.

Archlight is intended to support the class of analyses tools that operate in the mode of model checkers. Model checking tools take an architecture description as input, do some processing on that description, and output a set of results. Those results most often come in the form of issues—errors, warnings, informational messages, and so on—that were detected by the model checker. Not all analysis tools operate in this

mode, of course. For example, simulators and other dynamic tools require a more dynamic analysis environment, often with user intervention in the process. Additionally, some tools that could ostensibly be described as model checkers, such as Rapide's, output artifacts that cannot be represented as a set of issues (e.g., event trace graphs). These tools can be integrated into ArchStudio as independent simulators or analysis tools, but would not be appropriate within the Archlight framework.

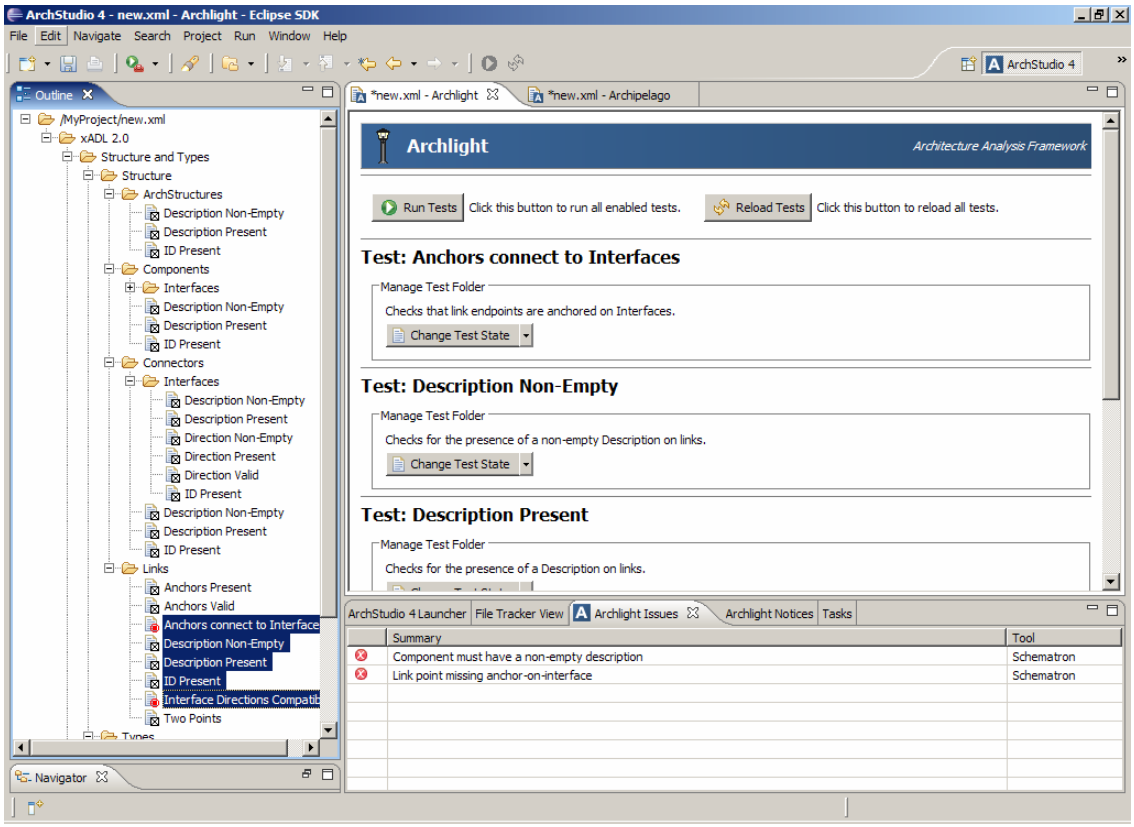


Figure 47. Archlight screenshot.

A screenshot of Archlight is shown in Figure 47. Like Archipelago and ArchEdit, Archlight shows a view of information in a xADL document. Unlike, for example, Archipelago, which shows views such as structural and statechart views, Archlight shows an analysis-centric view of an architecture description. Archlight's

user interface is broken up into three areas. The left pane contains the Archlight tree. The Archlight tree is populated by Archlight tests, organized into folders that group the tests. Each test is accompanied by an icon indicating its state—applied, disabled, or unapplied. While the selection of available tests is provided by the Archlight framework itself, the state of each test is stored in the xADL document. Tests are applied, disabled, or unapplied at a document level for several reasons. First, a document may or may not be intended to pass all tests. Some of the tests may check for mutually exclusive properties—owing to differing semantic interpretations of xADL, different target architectural styles, and so on—and thus cannot all be passed at once. Some documents may not use all of xADL's schemas, and therefore will not pass tests intended to check for properties present in an unused schema. Second, as architecture descriptions are written and evolved, they will be in various states of completeness and correctness. A half-complete architecture description may pass only a subset of available tests, with increasing test coverage as the description is fleshed out. Archlight's document-level test application allows stakeholders to customize their architecture testing strategy to their own needs.

The upper-right pane displays details about each selected test—what the test does, for example. The user can enable or disable tests in this pane, or in the tree. This pane also contains activation buttons for both running applied tests on the current document, and for reloading tests. Reloading tests is necessary when users add or change tests while the environment is running.

The lower-right pane displays all issues found during testing. A basic description of each issue, along with its severity, is displayed in this pane. A user can

right-click on any found issue and view additional detail in a pop-up window, or tell ArchStudio to navigate to the source of the issue in any ArchStudio editor (e.g., ArchEdit or Archipelago). Each editor has the opportunity to respond to such a focus request in the way most natural for that editor: ArchEdit highlights the appropriate tree node, and Archipelago brings up a graphical depiction of the architecture, highlights the appropriate element, and 'flies' the user's view to focus on that element.

4.6.1 Archlight Internals

Internally, Archlight has three core concepts: engines, tests, and issues. Analysis engines are software tools that incorporate algorithms for running tests on an architecture description and providing a set of issues as output. Each analysis engine provides one or more tests to the framework. A test is a condition, constraint, or program that can be applied to an architecture description, producing zero or more architecture issues as a result. Issues are errors, warnings, or simply informational messages. They are data structures that include a short description, a detailed description, and references to elements that indicate what parts of the architecture description were responsible for the issue's creation. Each test is run by a single analysis engine, but distinctions between analysis engines are not evident in the Archlight tree. To an end-user, the actual underlying engine performing a test is effectively unimportant—the test itself is what matters. Issues are displayed in the Archlight issue list.

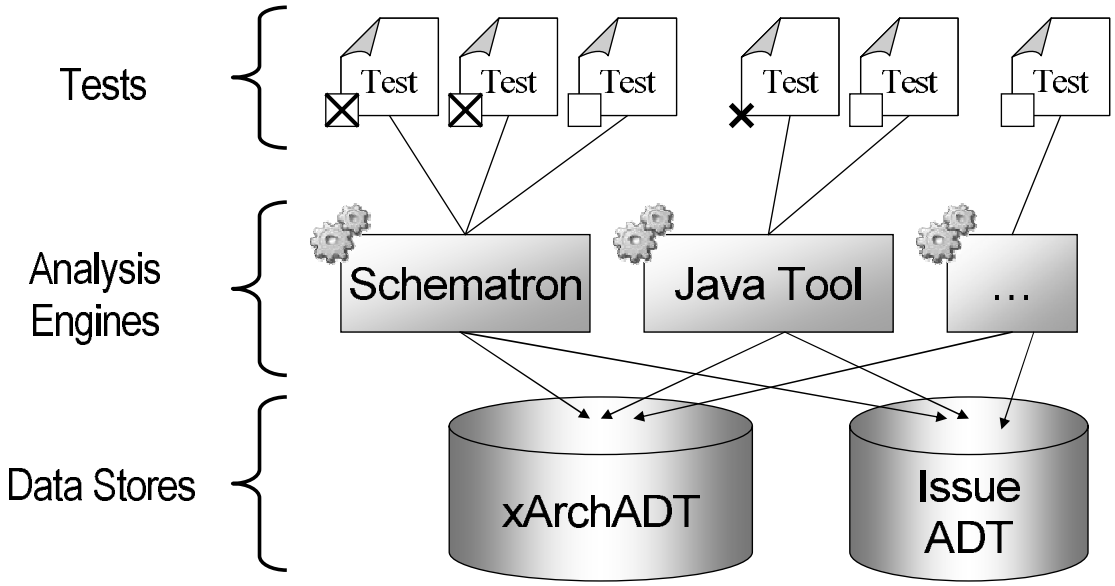


Figure 48. Diagram showing the relationship between data stores, analysis engines, and tests.

The relationships between data stores (xArchADT and Archlight's internal issue store called the Issue ADT), analysis engines, and tests are shown in Figure 48. Each testing engine provides a set of tests, which may or may not be enabled. When testing commences, if the engine is responsible for any applied tests, the engine accesses the architecture description via xArchADT, runs the tests, and stores an updated issue list in the Issue ADT. The Archlight issue list in the user interface is simply a GUI view of the data in the Issue ADT.

Archlight tests are be grouped in hierarchically organized 'folders,' enabling test writers to group related tests. Using this structure, top-level folders can correspond to xADL schemas. In this way, 'packages' of tests corresponding to a particular schema can be developed and deployed along with that schema to perform consistency checks or analyses relevant to that schema.

As noted above Archlight's mode of operation is most similar to that of model checking tools. This was a deliberate choice over alternatives such as, for example, a

"design critics" approach [136, 139]. Users must explicitly invoke testing, and analysis can take seconds or minutes depending on the size of the architecture description and the complexity of the test. This particular interface was selected because of the broad availability of generic off-the-shelf model checking tools available that work in this fashion. These include constraint checkers like Schematron [90] and xlinkit [117, 118], as well as more complex and dynamic model checkers such as Alloy [87] and SPIN [78]. Using an off-the-shelf generic model checker to analyze architectural properties is not a novel idea—Wright uses the off-the-shelf CSP analyzer FDR [57] for its analysis duties. Doing so is not necessarily straightforward, however—no generic modeling tool takes an architecture description as input directly. To integrate an off-the-shelf model checker as an architectural analysis tool, it must be wrapped.

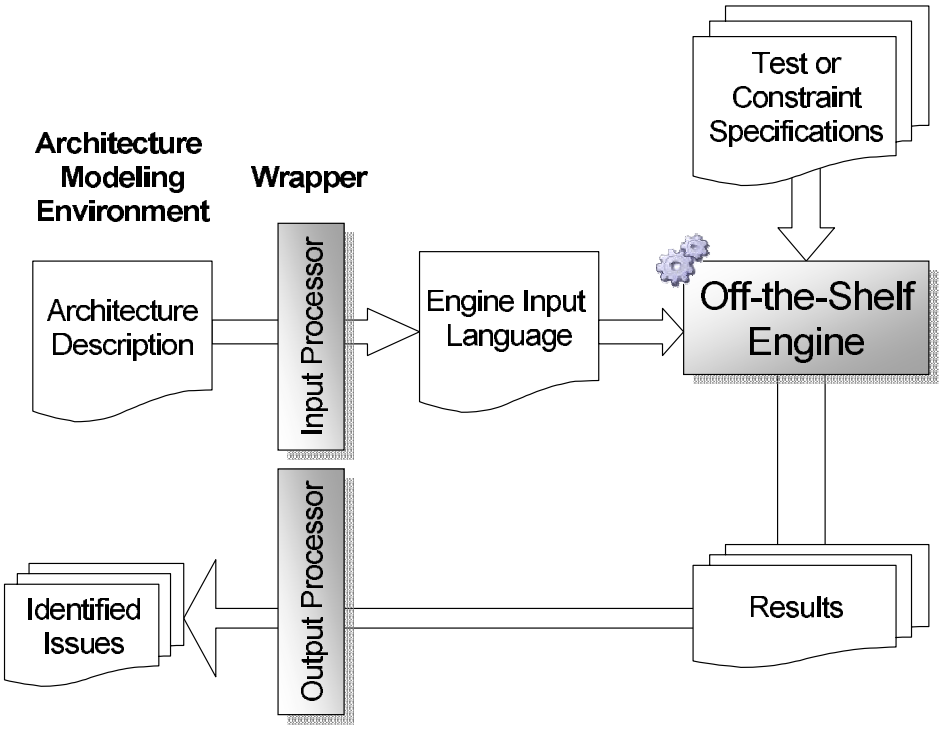


Figure 49. Using a wrapper to adapt an off-the-shelf model checker for architectural analysis.

Figure 49 shows what such wrappers look like. Architecure descriptions are transformed by an input processor into the input language for the off-the-shelf analysis engine. These are then combined with tests or constraint specifications, which are often expressed directly in the test/constraint language of the engine. The engine runs the tests or evaluates the constraints, and outputs a set of results in a format propriety to the engine. These results may indicate test failures or constraint violations. The results are translated by an output processor into identified issues; in ArchStudio's case, they are translated into Archlight issue data structures.

4.6.2 Schematron

This integration strategy is used in ArchStudio to integrate the Schematron [90] analysis approach into Archlight. Schematron provides the analysis capability for most of the current Archlight tests. Schematron itself is not an analysis engine by the strictest definition. Schematron is a set of XSLT templates [32] that are used, along with an XSLT processor such as Apache Xalan [165] to check constraints within an XML document. Effectively, Schematron is a clever approach that turns a generic document translation engine (the XSLT processor) into a constraint checker for XML documents.

All told, within ArchStudio the approach works in the same manner as shown in Figure 49. The input language for Schematron is an XML document. In ArchStudio's case, the input processor is simply a call to xArchADT to serialize a xADL model to XML. Tests are written in a testing language created specifically for Schematron.

```
<pattern id="test.80c3147f.102d2871711.ff3a20b004329533.2b2"
         name="xADL 2.0/Structure and
               Types/Types/Components/Interfaces/Interface Type =
               Signature Type"

               description="Checks that the type of a component's
```

```
                                interface matches its signature's type.">

    <rule context="instance:xArch/types:archStructure/types:component/
                   types:interface">
      <let name="href1" value="./types:type/@xlink:href"/>
      <let name="sigid1"
           value="substring-after(./types:signature/@xlink:href, #')"/>
      <let name="href2"
           value="/instance:xArch/types:archTypes/types:componentType/
                  types:signature[@types:id=string($sigid1)]/types:type/
                  @xlink:href"/>
      <assert test="$href1=$href2">

        id0=<value-of select="@types:id"/> |*|
        iddesc0=Interface |*|
        id1=<value-of select="../@types:id"/> |*|
        iddesc1=Component |*|

        text=Component interface
             <value-of select="../types:description"/> should have the
             same interface type as its signature |*|

        detail=The interface type of
               Interface <value-of select="../types:description"/> on
               Component <value-of select="../../types:description"/>
               must be the same as the interface type of its signature.
      </assert>
    </rule>
  </pattern>
```

Figure 50. Schematron test determining whether interface and signature types are compatible.

A sample Schematron test is shown in Figure 50. This particular test is a type-checking test: it determines whether the type of an interface and its signature match. This test contains all the information Archlight needs to make the determination of whether the test passed or not, create an issue if it failed, and link that issue to the specific interface and signature with mismatched types. The test is contained in a ‘pattern’ element, which has the test’s unique ID. This identifier is always associated with this test (even if the test contents change—to enhance or refine the test, for example), and is how xADL documents reference the test. The name attribute contains both this test’s short name and its location in the Archlight tree. Any analysis tool can insert a test anywhere in the Archlight tree that is appropriate. The description attribute

specifies the long description that will appear in the Archlight UI. The test condition itself is specified in the ‘rule’ and ‘assert’ elements. The ‘rule’ element includes a ‘context’ attribute that specifies a root in the xADL document from which the test condition will be evaluated. An optional series of ‘let’ elements follows: The ‘let’ elements can be used to break up a complex assertion by evaluating parts of the assertion and storing those evaluations in variables. The assertion here is rather simple: it compares the value of two of the variables evaluated by the let elements above. The body of the assertion is included in the Schematron output if the test fails. This data includes a short error message, a detailed error message, and a list of identifiers within the xADL document indicating the problematic elements. These identifiers are used by the Archlight user interface to focus other editors (such as ArchEdit and Archipelago) on those elements automatically.

The important thing to notice about this test is its length: only about 35 lines of XML. Writing the equivalent test as a custom Java module, for example, would require a hundred lines or more. The compactness of test specifications is one important reason why off-the-shelf model checkers are useful as architecture analysis tools, and why Archlight is geared toward integrating such tools. Users can write their own tests, such as the one above, in ordinary text files, and ArchStudio will incorporate those tests directly into the Archlight tree as part of the environment.

Schematron can be configured to output test results in many different formats: plain text, HTML, XML, and so on. For ArchStudio, Schematron is configured to output results in an XML format because this format is easily parsed and processed afterward. An output processor, part of the Archlight Schematron wrapper, reads this

resulting XML document using DOM and processes it, resulting in a set of Archlight issue data structures.

4.7 Archlight Summary and Contributions

Archlight gives stakeholders fine-grained control over architecture analysis and consistency checking. It lets them specify, on a document-by-document level, what constraints or tests are expected to pass. It can run those tests automatically and provide error reports using a combination of analysis engines. New consistency tests can be written to take advantage of existing analysis engines such as Schematron. Leveraging the compact size of the tests for these models makes incorporating tests into the environment more efficient. User-incorporated tests work and act exactly like default ArchStudio tests, because ArchStudio tests are integrated into the environment with the same mechanisms. When existing analysis engines are not sufficient, entirely new engines can be integrated seamlessly through the use of wrappers.

Archlight is well-suited as an analysis framework for a stakeholder-driven architecture modeling environment. It provides users with incremental ways of integrating their own tests and test engines, and thus to tailor the environment to their own notions of correctness and consistency. It makes no overt assumptions about what kinds of tests can be run, although it makes some assumptions with respect to the mode of operation of analysis engines that are consistent with most off-the-shelf model checkers.

4.8 ArchStudio as a Tool Integration Environment

ArchStudio provides stakeholder-driven extensibility for architectural modeling in the three key areas of notations (with xADL 2.0), visualization (with Archipelago) and analysis and consistency checking (with Archlight). However, a complete modeling environment requires the integration of additional tools that do not fit within these categories. These include tools to instantiate and run architecture-based software systems, tools to manage product-lines, additional non-symbolic editors, and so on.

As an environment, ArchStudio integrates all these ancillary tools in addition to components described above such as xArchADT, Archipelago, and Archlight, into a cohesive whole. The problem of tool integration in software development environments [169] has been addressed by many previous approaches, such as the UNIX operating system [12, 172], Marvel [95], Interlisp [163], and countless others. ArchStudio uses two primary strategies for supporting tool integration. The first is its use of its own architecture-centric development techniques internally, and the second is its integration within the Eclipse development environment [51].

4.8.1 The Construction of ArchStudio

ArchStudio is constructed using an architecture-centric approach. Its functionality is broken up into independent components. These components expose well-specified provided interfaces, through which other components can call upon their services. Components also expose required interfaces, indicating the services that they need to implement their functionality.

4.8.2 The Myx Architectural Style

The construction of ArchStudio is guided by an architectural style called Myx [40]. Myx is an architectural style created specifically for the construction of ArchStudio, taking into account lessons learned from the construction and maintenance of earlier versions of ArchStudio constructed in C2. Myx is a hybrid style, drawing primarily from the C2 [161] and Weaves [65] architectural styles. The rules of the Myx style are as follows:

- Components are used as the loci of computation;
- Connectors are used as the loci of communication;
- Components communicate only through well-defined provided and required interfaces;
- Components and connectors have two ‘faces’ (also called ‘domains,’ in the C2 literature), ‘top’ and ‘bottom’;
- Components interact through three distinct patterns:
    - Synchronous bottom-to-top procedure call;
    - Asynchronous top-to-bottom (notification) messaging; and
    - Asynchronous bottom-to-top (request) messaging;
- Components may only make assumptions about the services provided above them, and may make no assumptions about the services provided below them.

In Myx, concurrency and threading are primarily dealt with by the application’s connectors—a feature inspired by a similar constraint in Weaves. Applications have at least one main thread of control. Additional threads may be created by components as

necessary. However, the primary source of asynchrony and concurrency comes through the connectors: each asynchronous connector has its own thread. Synchronous component-to-component invocations are made asynchronous by the use of these connectors: a synchronous invocation on the connector is passed to a separate thread of control. The calling component gets control of the calling thread back immediately, while the connector's thread shepherds the invocation concurrently through the called component. Asynchronous interactions are fundamentally one-way; the intermediate synchronous calls do not return data.

Components are not prohibited from assuming that they share memory with other components, which is a difference between Myx and C2. Making this assumption greatly increases coupling between components, although it can simplify implementations and increase performance. Because the tighter coupling increases implicit dependencies and reduces component reusability, allowing components to share memory is strongly discouraged—if it is used, 'islands' of components that share memory only among themselves can be designated as a compromise.

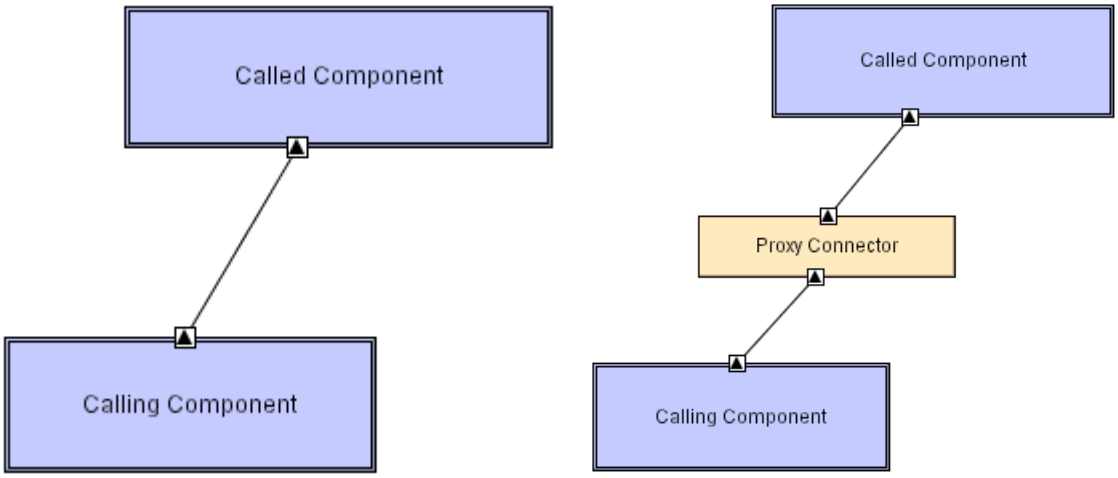


Figure 51. Synchronous interaction patterns in Myx.

As noted above, Myx does support synchronous calls in the ‘upward’ direction through the architecture. Figure 51 shows the two most common patterns of synchronous invocation in Myx: direct component-to-component calls, and calls via a proxy connector. Other, similar patterns are also allowed. For example, ArchStudio uses a synchronous multi-way connector, where the same invocation is made by the connector on many components that implement the same interface. The results are aggregated by and made available to the caller via a separate provided interface on the connector.

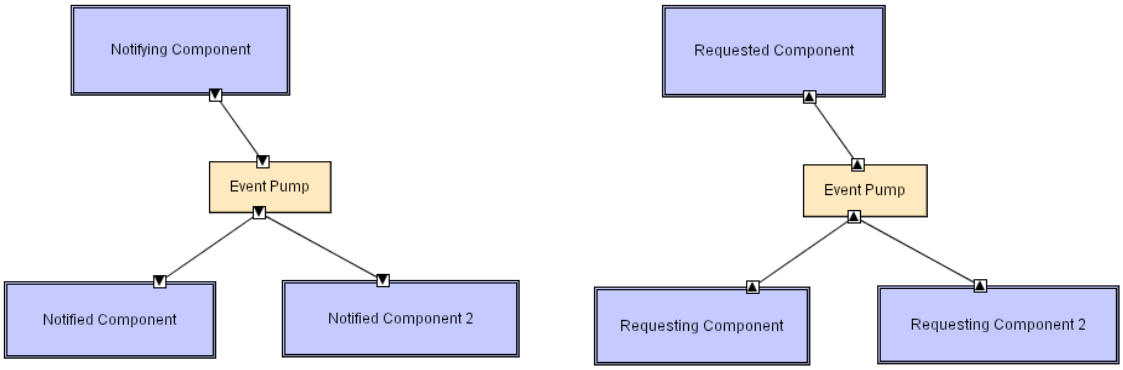


Figure 52. Asynchronous interaction patterns in Myx.

Figure 52 shows the most common asynchronous interaction patterns in Myx. In general, asynchrony in Myx is handled through event pump connectors that contain their own threads of control. Other variants of these patterns are possible as well: a two-way event pump would strongly resemble a C2-style bus connector. Thus, if all other constraints are obeyed, C2 can be seen as a subset of the Myx style.

As a style for an environment of loosely-coupled, integrated tools like ArchStudio, Myx is an excellent fit. It permits the integration of components that operate in both a synchronous and asynchronous fashion. It enforces loose coupling and maximizes reusability through the required use of explicit interfaces, the property of substrate independence, and the admonishment to construct components that do not use shared memory to communicate.

The Myx architectural style is programming-language independent. However, an architecture implementation framework for building Myx-based systems in Java is available, called myx.fw. This framework provides abstract base classes for components and connectors. It also includes a runtime that can be called to instantiate, connect, and manage the lifecycle of Myx components. This Myx runtime class is itself encapsulated

in a Myx component, making it possible to embed Myx architectures inside one another.

4.8.3 The ArchStudio Bootstrapping Process

ArchStudio, being an environment composed of loosely-coupled components, needs to be accompanied by a configuration that indicates how these components are supposed to be instantiated and connected. In ArchStudio, this configuration is provided by ArchStudio's xADL 2.0 architecture description, which is deployed with ArchStudio as part of its implementation.

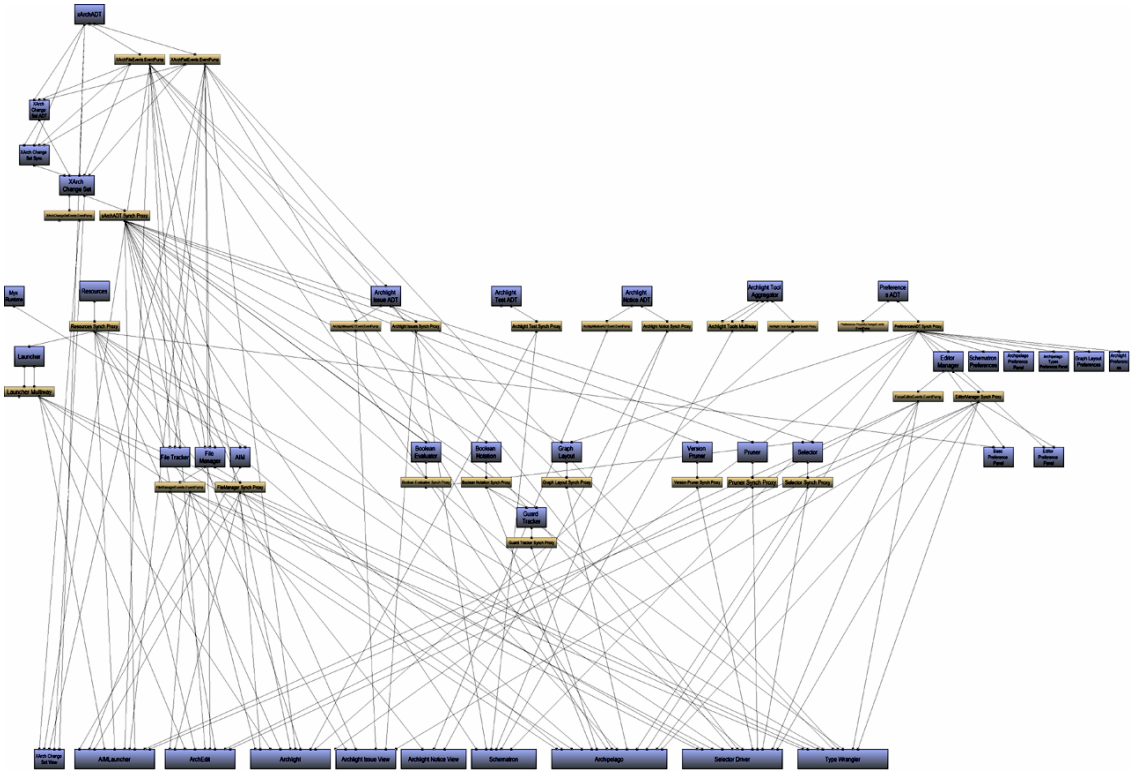


Figure 53. An Archipelago depiction of ArchStudio 4's architecture.

The Archipelago depiction of ArchStudio 4's (build 4.0.5) architecture is shown in Figure 53. Obviously, the elements in the diagram are too small to make out individually due to the size of the diagram. However, several things are nonetheless

evident from this visualization. First, the layered aspect of the Myx style is visible. Second, substrate independence is also evident: components on the bottom of the diagram depend on those above them, but not vice-versa.

The upper-left part of the diagram is clearly important, as a large number of links are connected in and out of this set of components. These are the components that manage the open xADL descriptions, chief among them xArchADT in the upper-left. Other top-layer components tend to be data stores as well—for storing ArchStudio preferences or Archlight issues, for example. Components further down in the architecture provide intermediate layers of functionality—tools that leverage the data structures and provide some computation as well. All components on the very bottom layer represent ArchStudio tools that have user interfaces—ArchEdit, Archipelago, Archlight, and so on.

Obviously, the diagram is complex; this is a side effect of the architecture of ArchStudio itself being complex. No dependencies or interconnections are elided here: this diagram represents the true configuration of ArchStudio. Several strategies might be applied to improve the visual appearance of the diagram. Elements could be hidden or removed, showing only aspects of the system. Perhaps only connections to ADTs, or only showing upward connections and not asynchronous back-channels would provide a clearer, but incomplete view. The architecture has already been visually reorganized to an extent through the inclusion of synchronous proxy connectors between components to centralize ‘hubs’ of links going to the same place.

Each component and connector in ArchStudio’s xADL description is linked to an implementation. These implementations are done in Java and each contains a root

class that serves as the Myx runtime's entrypoint to accessing the component or connector in the myx.fw framework.

Each time ArchStudio starts, a very small system is created using a subset of the ArchStudio components. These include xArchADT, a Myx Runtime component, and a bootstrapper component. The configuration of this micro-ArchStudio is static and hardcoded into the bootstrap launcher. The bootstrapper component communicates with xArchADT to load the ArchStudio xADL description into memory. It then iterates through the description, getting the implementation mapping for each component and connector. It makes calls on the Myx runtime to instantiate these components and connectors. When this is done, the bootstrapper reads through the links in the xADL document and connects the Myx elements according to those links. Therefore, the running copy of ArchStudio is always consistent with its architecture description, since it was created using the architecture description directly.

As a side effect, the architecture of ArchStudio can often be evolved simply by updating its architectural description. Inserting a new component into the environment simply means coding that component's implementation and adding it to the architecture description. If it uses services already provided by existing ArchStudio components, then it can simply be linked with those components in the xADL model. No recoding of any other components is required.

4.8.4 ArchStudio as an Eclipse Application

The first three versions of ArchStudio were stand-alone applications, operating outside the context of any larger tool-integration environment except the operating system. These versions of ArchStudio provided basic functionality such as file

management, user interface elements, preferences, and so on by themselves. ArchStudio 4 is fully integrated within the Eclipse integrated development environment.

The history of integrated development environments goes back to the earliest days of computing, with Interlisp [163] as a classic early example. Later exemplars such as Microsoft Visual Studio [110] and JBuilder [35] took additional advantage of increased computing power and better graphical user interfaces. These environments integrate many tools used in the software development process—compilers, linkers, debuggers, 'smart' code editors, configuration management clients, and so on. In general, all these tools deal with code as the primary development artifact. However, many of the same functions—editing, configuration management, file and preference management, and so on—are all equally applicable in an environment like ArchStudio where architecture descriptions are the primary artifact.

In the mid-to-late 1990s, several IDE vendors had offered products in the marketplace with similar functionality—Visual Studio, JBuilder, and others among them. IBM was also offering a similar tool suite called VisualAge [29]. In the late 1990s, IBM began development of a next-generation IDE called Eclipse, constructed in Java, that was based on the earlier VisualAge environments as well as experiences IBM had developing environments for Smalltalk development. They also made the crucial decision to release the environment as an open-source project, meaning that the public at large would not only be able to get and use the environment for free, but also improve and extend the environment by integrating their own tools, bug fixes, and enhancements.

Originally, Eclipse was most useful as a development environment for Java systems, this was bolstered by the powerful Java development toolkit (JDT) integrated within Eclipse. However, the JDT was simply a set of tools integrated using a tool integration framework that would later become known as the rich client platform (RCP). The RCP itself and its constituent services provide a general platform for integrating tools into an integrated development environment.

Integrating an environment such as ArchStudio into Eclipse confers many advantages. It allows ArchStudio to leverage all of the basic Eclipse services such a file and preference management, instead of handling them itself. It provides a consistent user interface that will be familiar to many users, especially Eclipse users. It offers the ability to provide a more cohesive user experience for architectural implementation—users can switch from architecture modeling to code-writing within the same application. It also lets ArchStudio leverage the plethora of additional tools that are being integrated into Eclipse. For example, by combining ArchStudio with subclipse [36], it becomes trivial to manage the evolution of architectural model documents in a Subversion [37] configuration management system. It also opens up the possibility that ArchStudio could be tightly integrated with other modeling tools such as Rational XDE [80] that are being integrated into Eclipse as well. From a social point of view, being integrated with Eclipse makes ArchStudio a member of a larger community of Eclipse extensions, making it more attractive to a wider user base.

Integrating ArchStudio into Eclipse is not a trivial task. Eclipse places many restrictions on how extensions to the environment must be implemented. These restrictions form what can be thought of as an architectural style for Eclipse. In Eclipse,

extensions to the environment are encapsulated into *plug-ins*. Eclipse plug-ins and their use were originally a proprietary mechanism that was specific to Eclipse; however, later versions of Eclipse are moving toward the use of the OSGi framework [66]. Plug-ins can interact in a variety of ways. The most common mechanism is for one plug-in to call another. Each plug-in can specify an explicit set of dependencies on other plug-ins. Each plug-in can expose a subset of its Java packages for access by other plug-ins. A plug-in may load and use any Java classes from the packages exposed by the plug-ins upon which it is dependent. A second mechanism of integration, called *extension points*, allows a plug-in to make callbacks to plug-ins that depend on it. Eclipse itself defines many extension points, and other plug-ins use them extensively. For example, to add a new tear-off window (called an 'Eclipse view') or editor pane to Eclipse, a plug-in adds itself as an extension to an existing Eclipse extension point. Eclipse then loads the target plug-in when the user invokes a user-interface option that causes the view or editor pane to be brought into the main window.

ArchStudio is constructed in the Myx architectural style. However, this was not a necessary condition—an environment like ArchStudio could have been constructed in a wide variety of architectural styles, each with different imbued qualities and tradeoffs. The choice to implement ArchStudio in the Myx style was made concurrently with the decision to integrate it with Eclipse, so the natural question arises: why not simply use the architectural style of Eclipse and Eclipse plug-ins to implement ArchStudio? Indeed, this was a possibility. Eclipse's architectural style was not used to implement ArchStudio because the qualities of the Myx framework were perceived to be a better fit for the needs of ArchStudio than Eclipse's. For example, with Eclipse plug-ins, it is

difficult to control or understand the precise dependencies between plug-ins. One plug-in can make use of any exposed class in another plug-in, instead of calling that plug-in through explicit interfaces. That is, one plug-in can instantiate and manipulate any class in a dependent plug-in, calling any public method and possibly updating its fields directly. This increases and confuses coupling between components since it is not always clear what constitutes a plug-in's 'interface,' and objects of classes from each plug-in intermingle in memory and share data in unconstrained ways. Second, fidelity to the architectural model is easier to maintain in Myx than in Eclipse. With Eclipse plug-ins, there is no strict notion of explicit provided and required interfaces—only plug-in dependencies. Third, additional Myx restrictions open up new possibilities in future ArchStudio versions, particularly with respect to dynamism and distribution. Myx and myx.fw allow components to be connected and disconnected at runtime. Components that obey the no-shared-memory admonishment can also be moved or offloaded to another processor. Eclipse plug-ins have limitations in both of these areas: for example, plug-in unloading is just now becoming supported in Eclipse. It is clear that the architectural style embodied by Eclipse did not anticipate these needs.

Implementing ArchStudio in Eclipse *and* using Myx at the same time is possible, but requires making some compromises to overcome architectural mismatches. The most significant of these mismatches is a common one, colloquially known as the *master of the universe problem*. This occurs when integrating two tools or frameworks only to find that one or both of them make the assumption that they alone are in control of things like system startup, determining control flow, and so on. In this specific case, Eclipse assumes that it is solely responsible for loading and instantiating

plug-ins. ArchStudio primarily handles this by being itself implemented as an Eclipse plug-in, whose internal architecture is built atop myx.fw. When Eclipse instantiates the ArchStudio plug-in, the bootstrapper reads ArchStudio's xADL description and instantiates the internal components, as described above. However, some elements of ArchStudio cannot be instantiated this way. Specifically, those parts that contribute views, editors, and other user interface elements (menu additions, preference panels) to the Eclipse user interface.

Eclipse's user interface elements are provided by Eclipse plug-ins that extend predefined extension points in the Eclipse framework. Plug-ins contribute elements in two ways. Primarily, Eclipse calls the plug-ins and code in the plug-ins populates the user-interface. This is not always the case, however: to manage memory footprint and improve startup times, Eclipse uses a 'lazy-loading' mechanism when dealing with its plug-ins. That is, plug-ins are not actually loaded and executed until they are actually activated by the user. Eclipse furthers this by allowing plug-ins to contribute some elements to the user interface without actually being loaded at all. For example, a plug-in can provide an additional menu option or view-activation button and have this appear in the user-interface without the plug-in actually being loaded. The plug-in is only loaded when the user finally clicks on the menu option. Eclipse gets the data needed to add these elements to the user interface from specially formatted files in each plug-in called 'plug-in manifests.'

This presents a conflict with ArchStudio, where it is the xADL specification and the ArchStudio components that are intended to dictate what elements are present in the user interface. In an ideal world, Eclipse would provide hooks such that ArchStudio

code, rather than a plug-in manifest, could provide the appropriate data. However, Eclipse provides no such hooks. ArchStudio is thus forced to ship with a separately-maintained plug-in manifest that specifies what user-interface elements ArchStudio contributes to Eclipse.

Worse yet, when Eclipse *does* instantiate these user-interface elements, it does so by loading and calling on the user-interface extension classes directly. It gives no opportunity for ArchStudio to intercept those calls and invoke the appropriate components to create the user-interface elements. The compromise in ArchStudio is to use a two-level approach. Each ArchStudio component that contributes to the Eclipse user interface is managed by the Myx framework. However, it also wears a "hat"—a set of classes that are instantiated and controlled by the Eclipse framework. These connect to one another via a shared registry class that is independent of Myx or Eclipse. All communication between components occurs via Myx.

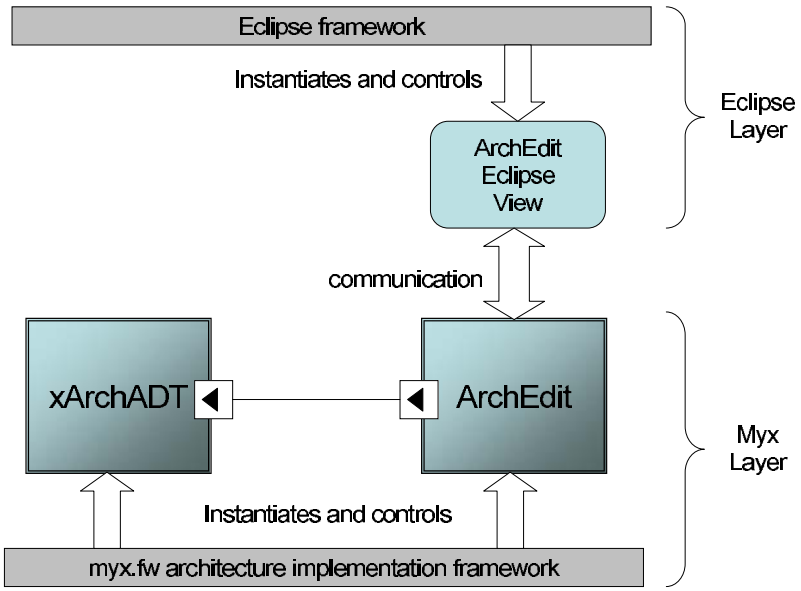


Figure 54. Managing architectural mismatch between Eclipse and Myx.

The two-layer strategy employed by ArchStudio and Eclipse is shown in Figure 54. The Eclipse portion of ArchEdit is loaded and managed by Eclipse; the remainder is managed by myx.fw. From a Myx perspective, the Eclipse portions of ArchEdit are simply part of the internals of the ArchEdit Myx component. Likewise, from an Eclipse perspective, the Myx infrastructure is simply a group of utility classes used and called by the Eclipse elements.

With these mismatches resolved, ArchStudio, Eclipse, and Myx can coexist in relative harmony. ArchStudio is, to the extent possible, a 'good Eclipse citizen,' using Eclipse services in their prescribed manner pervasively. So, instead of providing its own preference or file management system, for example, ArchStudio uses the Eclipse systems. This maintains a consistent user-interface look, feel, and operation between Eclipse, ArchStudio, and other Eclipse-based tools that a user might be running.

4.9 Additional ArchStudio Tools

xArchADT, ArchEdit, Archipelago, and Archlight form the core set of ArchStudio tools. However, ArchStudio's tool integration mechanisms through Myx are used to integrate additional tools that provide functionality apart from these. Two examples of such tools are the Type Wrangler and the Product-Line Selector.

4.9.1 The Type Wrangler

ArchEdit provides a tree-based view of xADL documents, and Archipelago visualizes them as interconnected graphs of symbols. These are not necessarily the only two visualizations that are applicable in an architecture modeling environment. From

time to time, additional visualizations that are optimized for dealing with a particular aspect of architectures are useful.

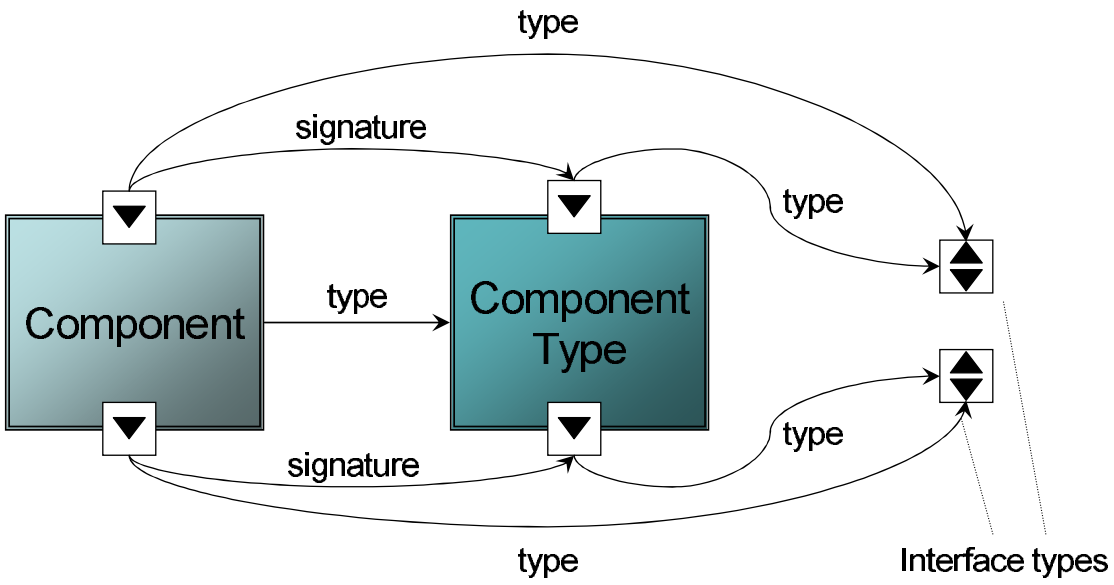


Figure 55. A type-consistent component in xADL.

Recall that in xADL, XLinks are used to connect components, connectors, and interfaces to their types. These links are used to maintain type consistency. The full set of links is shown in Figure 55. A component links to its types; each interface on the component links to its signature, each interface and signature links to its interface type (and the interface and associated signature are intended to link to the same interface type).

These links and relationships are hard to visualize symbolically, and the link structure is not readily apparent in ArchEdit. For this reason, ArchStudio includes a purpose-specific visualization called the Type Wrangler that helps users to understand, maintain, and repair these links. The Type Wrangler is an independent ArchStudio component and interacts with other components primarily by reading and making changes to the model stored in xArchADT.

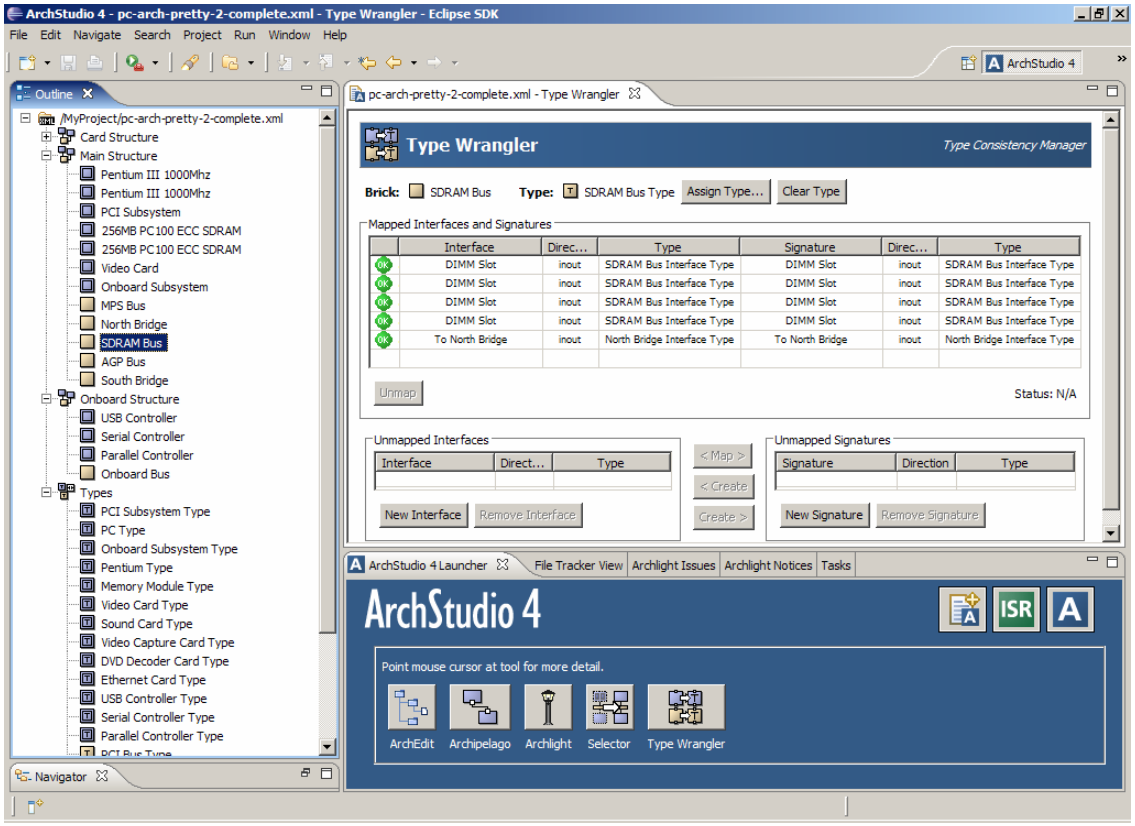


Figure 56. Type Wrangler screenshot.

A screenshot of the Type Wrangler component is shown in Figure 56. As with ArchEdit and Archipelago, a tree on the left shows document structure—in this case components, connectors, and types. By clicking on any of these, the main editor pane, on the upper right, is populated. This pane contains four main areas. The header shows the currently selected component or connector and its type (if a type is selected, only the type is shown). Buttons allow types to be cleared or reassigned here. The lower-left quadrant contains a table of ‘unmapped interfaces.’ These are interfaces that exist on the selected component or connector, but are not mapped to signatures. The lower-right quadrant contains a table of ‘unmapped signatures.’ These are signatures that exist on

the selected component or connector type, but are not mapped to interfaces on the selected instance. The top pane shows mapped interface-signature pairs.

The goal of the Type Wrangler is to move all interfaces and signatures into the top pane and make them consistent. An interface and signature are considered consistent if they have the same direction and the same type. User-interface elements in the type wrangler allow the user to assign directions and types to any interface or signature to help establish these relationships. Additionally, buttons between the two ‘unmapped’ areas at the bottom help to match up interfaces with signatures. One button automatically creates a compatible signature given an interface; the other creates a compatible interface given a signature. If both the appropriate signature and interface already exist, then they can both be selected and the ‘map’ button will connect the two with an XLink. Because the Type Wrangler stores all its data in xArchADT, any changes made here will show up in other editors such as ArchEdit and Archipelago immediately, and vice-versa.

4.9.2 The Product-Line Selector

Some architecture tools cannot be classified as editors, visualizations, or analysis tools at all—they perform functions pertaining to other parts of the system development lifecycle. The Product-line Selector is one such tool.

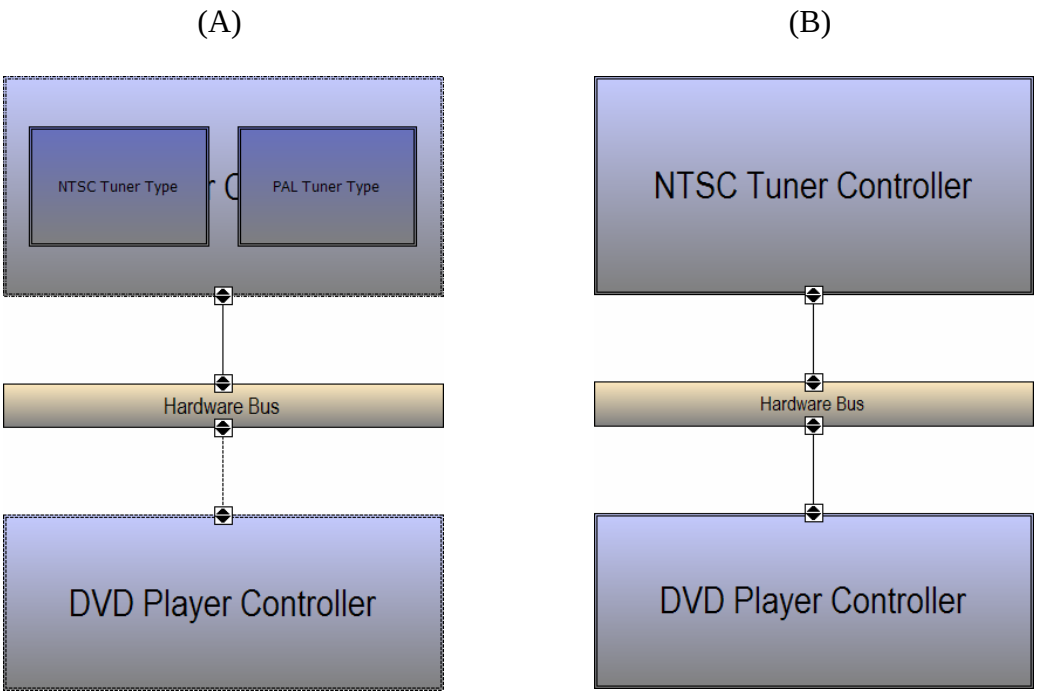


Figure 57. Simple product-line architecture before (a) and after (b) selection.

Figure 57 (a) shows a simple product-line architecture for a television set like those described earlier in this chapter. It contains both variant and optional elements. The tuner is variant, and may be selected as an NTSC or PAL tuner. The DVD player controller and its connection to the bus are optional—the TV may or may not have an integrated DVD player. (B) shows one product from that product-line, with an NTSC tuner controller and a DVD player included. With appropriate guard conditions in place for each optional or variant element, the actual selection of a single product from the product line can be done automatically. ArchStudio's tool for this purpose is the Product-line Selector.

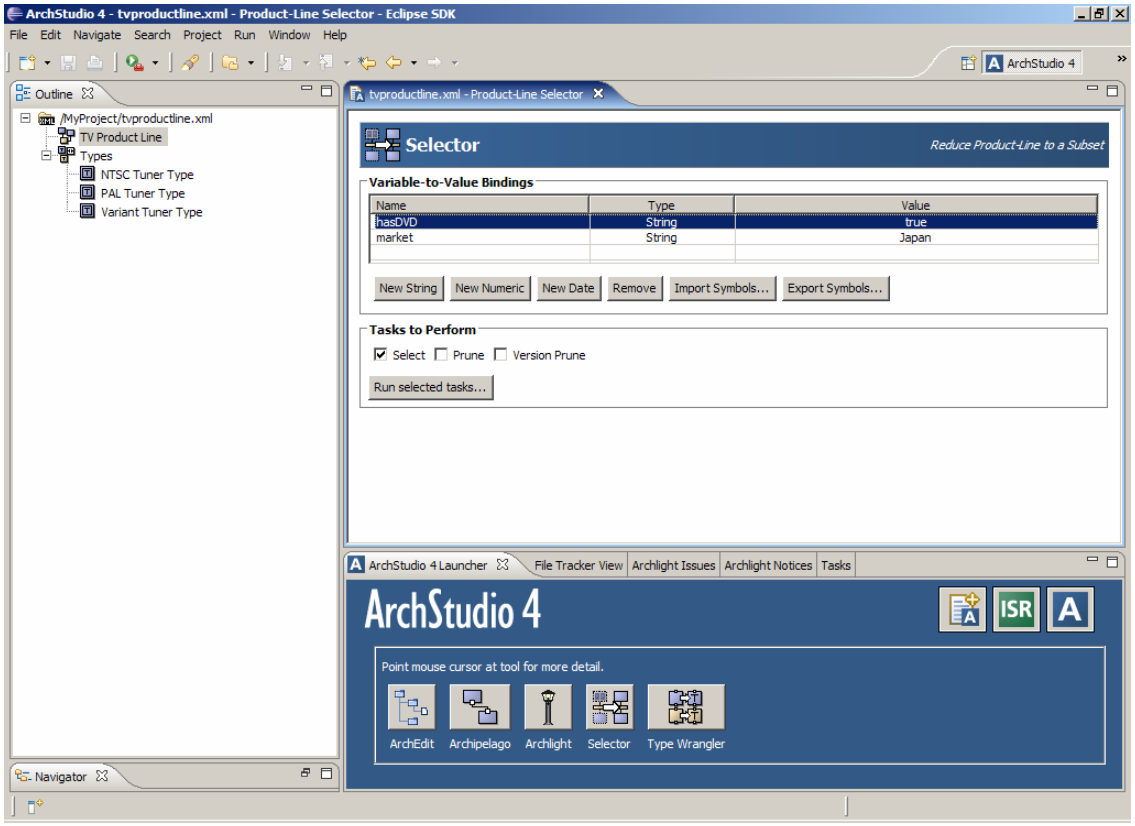


Figure 58. Product-line Selector screenshot.

A screenshot of the Product-line Selector is shown in Figure 58. The tree on the left shows xADL structures and types; the selected structure or type serves as the starting point for selection in a hierarchical architecture (it is generally the topmost structure or the outermost enclosing type). The editor pane on the right contains form fields for assigning values to variables. The variables that appear here are those found in the product-line guards. Once all values are assigned to variables, the user has the option of running any combination of three operations on the product-line: selection, which reduces a product line by resolving points of variation, pruning, which removes unused types that may result from selection, and version pruning, which removes unnecessary versioning information that may result from selection.

The result of running the selector on a xADL document is the creation of a new xADL document with the selected transformations (selection, pruning, or version pruning) applied.

4.10 The Complete Modular Picture

ArchStudio supports stakeholder-driven, multi-view software architecture modeling through a combination of an extensible notation—xADL 2.0, an extensible visualization environment—Archipelago, an extensible analysis framework—Archlight, and extensible tool integration with Myx and Eclipse.

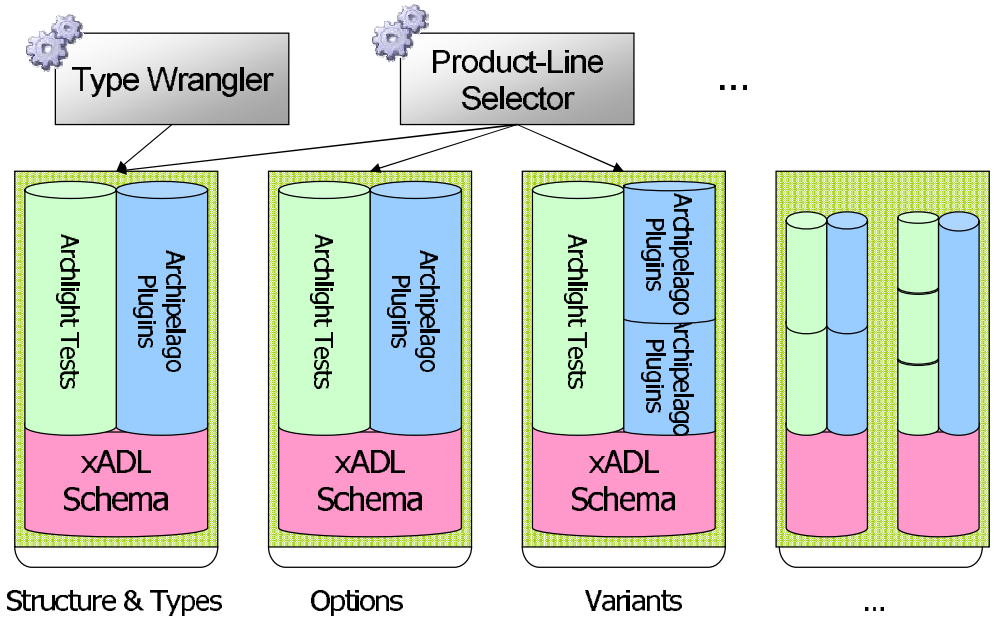


Figure 59. The complete modular picture in ArchStudio.

New concerns can be bundled into extension 'packages' and integrated into the environment, as shown in Figure 59. Modeling a new concern can be done by adding a new xADL schema (or schemas), along with Archipelago plug-ins to add visualizations support for the schema, and Archlight tests (and possibly new analysis engines to run

them). Additional special purpose tools and visualizations can be integrated that coordinate with all three of these.

ArchStudio fulfills the first two hypotheses of this research. Recall that they are:

H1. An environment can be constructed that supports stakeholder-driven multi-view software architecture modeling, addressing modeling, visualization, and consistency checking; and

H2. Within such an environment, architectural concerns can be supported by interdependent, reusable modules corresponding to concerns that can be composed into modeling support for a particular project or need.

With respect to **H1**, ArchStudio addresses all these concerns; this has been detailed extensively in this chapter. With respect to **H2**, the modularization of concerns such as structure, types, instances, options, variants, versions, implementations, and so on, integrated modularly as shown in Figure 59, provides clear and convincing evidence that a wide variety of architectural concerns are well-supported by this approach.

5 Design Principles

The preceding chapter provides a detailed description of how the ArchStudio environment is constructed. In the process of constructing ArchStudio, many design decisions were made for reasons that may or may not be obvious. It is also unclear, for example, whether some design decisions are flexible—whether some part of ArchStudio could be done differently and still retain some or all of the target qualities. This chapter will provide a summary of the design principles that guide ArchStudio, in an effort to make the design rationale behind ArchStudio clearer, separate essence from accident [26], and provide insights and guidance for developers wishing to create their own environment.

With respect to the definition of architecture presented in Section 3.2, these design principles can also be considered design decisions. They are termed ‘principles’ here primarily because they are presented as applicable to stakeholder-centric architecture modeling environments in general, and are not simply decisions applied to one particular environment—ArchStudio. Some, but not all of the principles are captured in ArchStudio’s own xADL description; the rationale behind this is provided in Section 5.8, below.

The principles presented in this chapter are broken up into six groups. The first group contains core principles: those that pervade every aspect of the environment. The second, third, and fourth sets are principles governing the construction of stakeholder-driven notations, visualizations, and analysis frameworks. The fifth set contains principles governing the development of tool integration strategies and environment configurations. The final set governs the design of xADL modules.

5.1 The Source of the Design Principles

The source of the design principles presented in this chapter is primarily experience. If these design principles were extracted from only a single experience—a single version of ArchStudio, for example—their usefulness might be called into question. However, these design principles are extracted from a long history of many versions of ArchStudio. ArchStudio 4, the version of ArchStudio described in Chapter 4, is the fourth major iteration of ArchStudio development.

ArchStudio 1 and 2 were initial research prototype environments developed in the mid-to-late 1990s [96, 127]. These versions of ArchStudio were focused primarily on supporting software development in the C2 architectural style. They themselves were also constructed in C2, wrapping custom-built or off-the-shelf components and using a C2 architecture implementation framework to support tool-integration. Structurally, these early environments have several properties in common with later ArchStudios. For example, access to architectural models was stored in an independent component and used as a point of integration for other tools. The elements of notations, visualizations, and analysis were all there as well. In ArchStudio 1, the underlying notation was C2SAD(E)L [105], an architecture description language developed explicitly for capturing C2-style architectures. This was replaced by xADL 1.0 [96] in ArchStudio 2. xADL 1.0 was an early XML-based architecture description language. It was somewhat more general-purpose than C2SAD(E)L, but clearly inherited many properties that made it amenable to specifying C2-style architectures. Despite being defined in XML, neither xADL 1.0 nor its toolset were extensible in any principled way. The primary advantage of using XML in this case was the set of off-the-shelf tools

for parsing, editing, and serializing these documents—operations that had to be implemented manually for C2SAD(E)L. Both ArchStudio 1 and 2 included visualization tools. A primary visualization tool, ArchShell, provided a 'command-line interface' for exploring and editing architecture descriptions. Commands were hard-wired into ArchShell and included basic operations such as 'add component' or 'weld brick1 brick2.' All input and response were done with text. A symbolic editor similar to Archipelago was also integrated called Argo/C2 [137], which is a cousin of the Argo/UML open-source UML editing environment. ArchStudio 2 also included a tool called DRADEL, which provided architecture analysis. DRADEL [105] checked a set of typing and subtyping relationships for C2 architectures, as well as certain kinds of interface compatibility.

ArchStudio 3 was a major redesign effort. It was the first version of the environment to focus explicitly on stakeholder-driven modeling. It used xADL 2.0 as its underlying modeling notation and included xArchADT in a form very close to its present one. Versions of Archipelago and Archlight (then called TRON) were included as well as the first versions of the Type Wrangler, Selector, and other tools. The main differences between ArchStudio 3 and ArchStudio 4 are:

Target architectural style: ArchStudio 3, like its predecessors, was built in the C2 architectural style. A new framework, called c2.fw, was constructed specifically for this purpose. c2.fw was somewhat more flexible than previous frameworks, and it also incorporated a mechanism by which synchronous procedure calls could be emulated over asynchronous events. This was critical for accessing data store components such as xArchADT. Issues with performance, latency, complexity, and user interface limitations

in implementing ArchStudio in C2 led to the development of the Myx style for ArchStudio 4.

Implementation Platform: ArchStudio 3, as noted above, was implemented as a stand-alone Java application. Its user interface was constructed using the Java Swing toolkit [174]. At the time ArchStudio 3 was constructed, Eclipse existed, but it was not yet the clear dominant player in the IDE/tool integration environment community that it became. Other environments, such as NetBeans [91] offered similar services, and we were reticent to throw our hat in the ring with any one of them. By the time ArchStudio 4 was under consideration, Eclipse had clearly become the dominant tool integration environment, and so the decision was made to port ArchStudio 4 to Eclipse. The reasoning behind this decision is explained in Chapter 4. Eclipse does not use Java Swing, it uses a different GUI framework known as SWT [119]. SWT provides similar functionality, but instead of implementing all widgets in pure Java, each widget in the platform is mapped to a native operating system widget. Thus, SWT applications, including Eclipse and ArchStudio 4, have 'native-looking' and often faster-performing user interfaces.

BNA Design: The BNA framework underlying Archipelago underwent a major redesign during ArchStudio 4 development. The most obvious change was the move from the Swing toolkit to SWT. Because BNA encapsulates all of its drawing functionality in a small number of classes (primarily the Thing Peers), it would have been possible to retain the Things and many of the Logics from a Swing implementation intact. However, during the Archipelago implementation for ArchStudio 3, it was discovered that certain design choices made in the BNA implementation were not ideal.

BNA 3 used object-oriented inheritance or copy-and-paste coding to combine features (e.g., borders, fields, labels, and so on) into individual Things. This made certain Things (especially generic Things like boxes) extremely large and difficult to extend or customize. BNA 4 replaced this mechanism with the current composition-based mechanism using assemblies. Additionally, BNA 3's core structure made it difficult to embed BNA views inside one another to create multi-view or hierarchical editors. BNA 4 introduced the concepts of BNA Views and BNA Worlds to rectify this.

5.2 Core Design Principles

Looking across the whole of ArchStudio, certain core principles emerge that are seen expressed in nearly every facet of the environment. These are high-level principles and are often refined into more specific principles that govern some individual aspect of the environment.

5.2.1 No Architectural Concern should be Privileged Over Any Other

This design principle is the core of ArchStudio's design, and is the principle that imbues stakeholder-centricity throughout the system. The principle advises to make as few assumptions about what can be modeled and how it can be modeled as possible. If almost fifteen years of software architecture research have revealed any insight, it is that there will never be a one-size-fits-all solution for architecture modeling. Each domain and project will invariably have its own concerns that cannot necessarily be anticipated by an environment developer. The only way to account for these concerns is to construct environments in which no particular concern is favored over any other.

Nearly every previous architecture modeling approach has some notion of privileged design decisions: Darwin's structures, Acme's core elements, UML's thirteen diagram types. There is generally a clear distinction between which concerns are modeled as a native part of the approach and which are added by stakeholders through extension mechanisms.

It is certainly reasonable to make assumptions about common or generally useful concerns (structural modeling, for example) and integrate those into the environment. ArchStudio does this extensively. However, it integrates structural modeling, visualization, and analysis using the same extension and integration mechanisms available to every other concern. These mechanisms provide neither special support nor bias toward integrating the structural modeling concern.

To support truly stakeholder-centric modeling, stakeholders must not only have the ability to extend the modeling environment (through access to the source code, for example) but also guidance and mechanisms to do so. These mechanisms must not only be present, but they must be exposed to and used by the environment's developers as if they were end-users of the environment itself.

5.2.2 Modeling Support for each Concern should be Captured in Reusable, Composable Modules

Composable modularity is a key design principal seen throughout ArchStudio. Keeping modeling support for each concern independent (or at least interdependent with known dependencies) is critical to keep the environment from evolving into a monolithic, tightly coupled, approach-driven modeling solutions. Besides separating each concern into support modules, a composition mechanism must be established that

allows these modules to be easily recomposed into specific modeling solutions. If the composition mechanism is onerous or requires extensive post-composition customization, the cost to extend the environment goes up rapidly.

Keeping the modules in(ter)dependent can also result in the creation of a library of reusable modules that support particular concerns. Although some modules will undoubtedly be so specific that they are only useful to a particular project, it is more often the case that modules will have some applicability in other projects, particularly those in the same application domain. By allowing modules to be reused across projects, the cost of developing each module is amortized. There are some specific strategies that can be employed to maximize module reusability, and these are described below. The general principle to maximize module reusability, however, is to develop modules for more generic concepts and then refine those into specific instances of those concepts. This is seen in ArchStudio, for example, in the progression of xADL schemas from ABSTRACT IMPLEMENTATION to JAVA SOURCE CODE IMPLEMENTATION to ECLIPSE JAVA SOURCE CODE IMPLEMENTATION—each schema adding yet-more specific details to the last. Even if the ECLIPSE JAVA SOURCE IMPLEMENTATION schema and supporting tools cannot be reused, the JAVA SOURCE CODE IMPLEMENTATION schema might be.

5.2.3 Modules should Group Related Extensions to Notation, Visualization, Analysis, and Other Tool Support

This strategy, captured best in Figure 59, ensures that each new concern has support for all critical aspects of the modeling problem: notation, visualization, analysis, and so on. Within such groupings, dependencies among extensions should mirror one another. For example, if notation extension B depends on notation element A, then the

visualization and analysis extensions for B may also depend on those for A. However, they may not import or depend on unrelated notation element C. This strategy maximizes reuse across aspects of the modeling problem and keeps the environment as a whole from evolving into a monolith.

5.2.4 Keep Cores Small and Layer Functionality

Small functional cores are perhaps the key driver of reusability and modularity in ArchStudio. These cores are generally the frameworks and data structures that underlie the major subsystems: xArchADT, the Archipelago tree and BNA, the Archlight scaffolding for tool integration, and so on. These are perhaps the most critical elements of the environment, and they are also the smallest and least complex. In general, their size is dwarfed by the size and complexity of the extensions implemented atop them.

When integrating a new feature, the option to integrate new functionality—especially cross-cutting functionality—into a framework core is tempting. In general, this temptation must be resisted. Every new feature integrated into a core framework increases the number of implicit assumptions made by that framework and has an effect on all extensions to the core. The core should generally only be modified as a last resort if it is infeasible to add such behavior in an extension.

Two anecdotes from the development of ArchStudio reflect this principle, both regarding integrating new features into the BNA framework. In the first case, the BNA framework had been engineered to avoid drawing a Thing if that Thing did not appear within the window as characterized by the current zoom and scroll position. This was a drawing optimization and applied to every imaginable Thing in BNA, so this

optimization was integrated directly into the BNA core. However, as more and more Things were developed, problems arose with this approach. It did not deal with all types of Things well—particularly those without bounding boxes that could be easily determined from looking at a Thing's properties. It could not deal with Things such as large enclosing borders, where the current window's bounds were contained entirely within the border. This border need not be drawn, since all parts of it lay outside the current window, but the framework-level optimization had no way of knowing this. It eventually became clear that this simple drawing optimization was not well-placed within the BNA core, and was eventually moved out into the Thing Peers (which are the elements that are responsible for drawing Things anyway). This resulted in Thing Peers being slightly more complex, but in the entire framework being far more flexible as to what kinds of Things were supported.

In the second case, early attempts were made to integrate multiple BNA views into the BNA framework, for developing hierarchical editors, dual-pane or dual-window editors, and so on. Doing this entirely within extensions proved to be infeasible, the concept of an 'editor' or a 'view' was present only in the core of the framework and could not be easily reused within the framework reflectively. For this reason, the BNA View and World concepts were integrated directly into the BNA framework. This was fortunately done in such a way that other elements (Things, Peers, and Logics) generally did not need to be modified, so the addition of these concepts changed few of the assumptions binding the implementation of these elements.

Keeping cores small is easy to espouse as a design principle but hard to implement in practice. Doing so successfully means providing substantive functionality

to users without over-constraining extensions. As noted in the above two anecdotes about BNA, it is not always easy to get the design of a core correct the first time (and only experience using the core can increase confident that it is, in fact, well-designed). Small core design is characterized by minimalism: reducing the problem to its fundamental and essential parts, and then understanding the nature of those parts.

From this discussion, it would be easy to assume that small core design equates with low functionality. Indeed, a small core will often provide fewer services to users than a large, complex one. The remedy for this is to provide additional services in layers of extensions above the core. For example, in BNA, a set of Logics exist whose sole purpose is to track different properties of Things. To the end-user, these extensions provide no value; however, they serve as a substrate to many other extensions that do. The same small-core design principles apply to these extension layers as well.

5.2.5 Support Incremental Adoption

Using a completely architecture-centric approach pervades system development activities. Not all projects or developers will be comfortable moving from existing processes (which may be inferior and ad-hoc) to a completely architecture-centric approach right away. This can be expensive, and returns on investment are not always obvious. For this reason, it is better for an environment to allow incremental feature adoption than to require users to integrate the environment into their development processes whole. The best way to achieve this is by establishing and maintaining functional layers and dependencies within an environment such that the environment's features can be 'cut' at any point and the target project can incorporate just those features below the 'cut point.'

ArchStudio, built in a layered fashion, supports incremental technology adoption. The lowest level of adoption is to incorporate the xADL language into a project. This may involve incorporating just the xADL schemas, or the data binding library. Within xADL, users do not have to use or incorporate all schemas—users can move from structural modeling to implementation mappings to product-lines a bit at a time. Beyond this, visualization and analysis technologies and techniques can be incorporated as well. A deep integration would also involve the binding of a system's architecture description to its implementation atop a well-defined architectural style and framework, as ArchStudio does with Myx and myx.fw.

5.2.6 Grow Extension Costs with Complexity

In a stakeholder-centric architecture modeling environment, end-users will eventually need to extend the environment for their own purposes. The cost to construct these extensions should grow with the complexity of the extension: simple extensions should come at low-cost, complex extensions can cost more. To the extent possible, there should be a continuous spectrum between simplicity and complexity. This property is critical for developing an environment that is stakeholder-centric, and does not just expose selective extension mechanisms or is simply open-source. Consider UML environments, for example. UML profiles provide a low-cost way of developing simple extensions, but more complex extensions (adding entirely new views, for example) become infeasibly expensive—consider the experience of the SysML partners. Achieving this property requires careful consideration of the design of modeling, visualization, analysis, and tool integration mechanisms in the environment,

and the guidance needed to achieve this property in those areas is found in the later sections of this chapter.

ArchStudio applies the principles described in this chapter to achieve this scaling of complexity and cost. With respect to modeling, new XML schemas are relatively easy to write no matter how complex they are, although extension schemas that only add a small amount of detail to the existing xADL capabilities may be as small as 50-100 lines of XML. Generators create all basic language tool support for free. Visualization tools scale as well: a simple, syntax directed editor (ArchEdit) will work with any new schema for free, with no additional coding. Archipelago is extended to support new schemas using new plug-ins. Schemas that add only small details or semantic extensions to existing ArchStudio capabilities can be added with only a few Archipelago plug-ins; complex schemas require more effort. With respect to analysis, Archlight allows users to incorporate new tests and analysis engines. New tests are relatively easy to write; most Schematron tests are less than 25 lines of XML code. New analysis engines require new code—scaffolding to wrap the engines. Incorporating a fundamentally new kind of analysis (i.e., something other than model checking such as simulation or testing using a prototype or runtime implementation) is also possible, although this would require the development of an entirely new analysis framework. Tool integration scales similarly: a new tool that uses services provided by existing tools can simply use Myx links to connect to the existing services, with all new code confined to the added components. Tools that need new services can be created as well, although the components providing these services need to be coded as well.

5.2.7 Avoid Over-Engineering

This design principle is intended as a limiting factor. Stakeholder-driven modeling environments consider aspects of the modeling problem at the meta-level. They seek to maximize user flexibility and extensibility. These characteristics make such environments a prime target for over-engineering: building functionality or features into the product anticipating future changes that never actually occur. The problem with over-engineering is that these features often add additional complexity to the system, or have negative effects on other software qualities—particularly performance.

An anecdote from ArchStudio's development exemplifies this. As noted earlier, ArchStudio 3 was constructed in the C2 architectural style. C2 imposes many constraints on applications: explicit dependencies, no shared memory among components, communication exclusively via asynchronous events, and so on. The benefits of the C2 style are manifold: in particular, these constraints make it relatively straightforward to add dynamism and distribution to a system. Dynamism is the property in which a system's architecture can be modified while the application is running—to add or replace components, for example—without shutting the system down. Distribution would involve moving a subset of the components to a different host. These qualities are nontrivial to achieve, and the C2 style makes this substantially easier.

This turned out to be a case of over-engineering. ArchStudio, intended primarily as a single-user tool that can be restarted when re-architected if needed, did not take advantage of two primary benefits of the C2 style. Following all the C2 constraints

strictly resulted in many undesirable and user-visible consequences. Foremost, since each component and connector ran in its own thread of control, a message traversing many layers of the architecture would have to pass through 7-10 threads of control, each way, on average. The most common interactions occurred between xArchADT, at the top of the architecture, and editors, at the bottom of the architecture. Thus, a substantial portion of ArchStudio messages traversed most of the layers in the architecture. Although the Java runtime has high-performance thread context-switching, these additional context switches made each call from an editor to xArchADT an order of magnitude or more slower than a direct call.

Certain ArchStudio components, particularly data store components like xArchADT and the Archlight Issue Store, are most naturally accessed synchronously. Synchronous access to data structures is helpful for two reasons: first, queries are most naturally synchronous: the calling component requests data and can proceed only when that data is available. Second, update operations must often be finished before a caller can proceed with processing, especially if that data is retrieved subsequently. C2 does not permit such synchronous interactions by default, and extensive functionality had to be developed in the c2.fw framework to support them. Effectively, this functionality simulated a synchronous procedure call over asynchronous messages, which were built atop a programming language that supported synchronous procedure calls as its primary mode of communication. This further reduced the performance of the system.

The no-shared-memory requirement caused interesting problems in the user interface. Since no two components were allowed to share memory, each component that provided a user interface did so in a separate window (integrating multiple user

interfaces into a single-window environment in Swing requires that all the elements run in the same memory space). The result was that ArchStudio 3 used a UI paradigm unfamiliar to most users, where multiple tools each ran in their own independent windows. This caused confusion with users, since similar environments (Eclipse, Visual Studio) used a single-window UI.

Avoiding over-engineering is not always a straightforward or trivial process. It is a balancing act between retaining maintainability and expandability of the environment in the future and importing too many additional constraints for unused features. ArchStudio 4 used a number of strategies in this regard. First, new features or changes to ArchStudio were evaluated with emphasis on the perspective of the end-user. If a proposed feature would potentially increase maintainability, but would only result in a negative impact on the end-user experience, then the feature was rejected. Second, ArchStudio uses small framework cores, as described above. Small cores tend to be more stable and exclude, rather than include, extraneous features. Third, a general philosophy for ArchStudio was to perform “Extreme Programming-like” [15] refactoring. In ‘Extreme Programming,’ only immediately necessary features are constructed. However, aggressive design refactoring is used when new features are needed or better designs are discovered.

5.3 Modeling Design Principles

The design principles in this section apply to the construction of modeling (that is, notational) support for stakeholder-centric architecture modeling.

5.3.1 Choose a Suitably Expressive Meta-Model

When developing an architecture modeling environment, many meta-models are available. In general, meta-models that can express one-to-one, one-to-many, and many-to-many relationships are suitable for architectural modeling. Such meta-models include entity-relationship (e.g., UML's MOF), relational (e.g., as expressed through tables in a relational database), object-oriented (as stored in an object-oriented programming language or database) or hierarchical (such as XML). Meta-models to avoid include flat structures (e.g., name-value property pairs) or meta-models where some structure is present in a core set of constructs, but extensions are flat. Fixed-structure meta-models suffer from limitations to extensibility: there is a clear delineation between 'core' concepts and extensions, and this violates the primary design principle of not privileging any particular concept over another. Flat structures make it difficult to express one-to-many and many-to-many relationships in data that are common in almost any architectural model. When including structured data in a flat meta-model, users tend to resort to creating ad-hoc, often incompatible structures within the flat model.

A second critical aspect of the meta-model is the need for it to support modular extension. That is, extensions should be specified in interdependent modules and then composed into modeling notations. This allows for feature reuse and divergent extension, as different organizations and stakeholders construct modeling features that fit their own needs. Not all meta-models support this naturally. For example, XML DTDs permit languages to be partially defined in modules, but composing them requires creating and maintaining an additional "master DTD" that depends on all

included modules. This strategy was used in the Modularization of XHTML standard [8]. In XML schemas, on the other hand, a language can be extended by simply adding a new module, without changing the definition of the base language at all.

ArchStudio uses a hierarchical meta-model based on XML schemas, and uses XLinks extensively to link elements that are not connected by parent-child relationships. Other meta-models could have been leveraged as well—for example, Telelogic System Architect [130] uses a relational meta-model to store its architecture models. The relational meta-model can generally be augmented by extensions through the addition of new tables without restructuring the existing data.

5.3.2 Encapsulate Models in a Separate Repository that is the Center of Coordination for All Tools

Architectural models are the most important artifact in the environment. Nearly every component and tool in an architecture modeling environment will need direct and simultaneous access to the architectural model. For this reason, it is best to encapsulate access to architectural models in a single component. All persistent architectural data should be stored in this component. The architecture model repository can then serve as a point of data integration between other tools in the environment. In this way, the repository serves very much as a blackboard through which tools coordinate and exchange data. ArchStudio's central model repository is the xArchADT component.

The alternatives to using a central data repository include, for example, maintaining multiple coordinated models in different tools, or building the environment such that only one tool is operating at a time and the architectural model is passed from tool to tool. The former approach introduces enormous complexity into the

environment, since maintaining coordinated independent models is difficult at best, and the problem worsens as more tools are brought into the picture. The latter approach requires the user to explicitly switch between tools and precludes using multiple tools simultaneously and represents a significantly worse user experience.

5.3.3 All Changes to Architectural Models should be made Explicitly Available through Events

If the architectural model is accessed (both read and written) concurrently by many tools, it is vital that all tools know what changes other tools are making. Tools often maintain their own internal state—for user interface or efficiency reasons, for example. This internal state must be kept up-to-date with the underlying model. If the underlying model changes, but internal tool state does not, then tool interfaces will become out-of-sync with the underlying repository. This can lead to user confusion or internal tool errors.

The most effective way to keep tools synchronized is for the architectural model repository to emit asynchronous events whenever any tool makes a change to a model. These events should include information about the type of change (addition, modification, removal), what was changed, where the change occurred in the architecture, the state of the architecture before the change, and the new state of the architecture after the change. Each individual tool that maintains internal state is responsible for listening to these events and updating internal state accordingly.

In ArchStudio, xArchADT emits such events every time a change is made to one of the models in ArchStudio. These events are multicast, via an event-pump connector, to all interested components.

5.3.4 The Model Repository should Provide Access to Documents as Structured by their Meta-Model

Once a meta-model is chosen, the model repository should at least provide access to document structure based on that meta-model. For example, an environment using a relational meta-model should allow access to the related tables, and an environment using a hierarchical structure should allow access to the tree structure. This gives tools maximal low-level control over the manipulation of architecture descriptions. Accessing architectural models at this level is not always the most convenient way of doing so—APIs with understanding of architectural semantics can simplify tool development greatly. These “convenience APIs” that provide more semantically-aware interfaces can be built on lower-level structural APIs.

In ArchStudio, xArchADT’s basic operations allow exploration and manipulation of xADL in its native tree structure. However, additional “convenience APIs” are provided by other ArchStudio components as well. Indexing components track element identifiers and XLink targets in each xADL document and make it possible to quickly find an element by its ID or trace an XLink. Another component provides a simpler API for reading and writing product-line Boolean guards, where each guard is presented as a traditional infix expression (e.g., foo == “value”) rather than in a tree structure.

5.3.5 The Model Repository Must Evolve with the Underlying Notation

In a stakeholder-driven modeling environment, the underlying modeling notation will evolve based on stakeholder needs, often in unexpected ways. Mechanisms must be provided such that the model repository can adapt to these changes with a minimum of recoding or hassle. Two strategies can be employed for this purpose. First, if the model repository's interface can be developed based on the notation's meta-model, as suggested in Section 5.3.4, then the interface to the model repository should only change when the meta-model changes—which should not be often. Second, if the model repository interface is syntax-directed, generative tools can be used to generate that interface when the underlying notation changes. ArchStudio applies this strategy by using Apigen to generate the data binding library.

5.3.6 The Model Repository should Support Reflection

Building syntax-directed tools requires not just an architectural model, but also a separate model of the underlying notation. To build such tools effectively, the model repository must include reflection as part of its interface. Reflection allows callers to explore the architecture meta-model just as they explore architectural models.

ArchStudio uses model reflection, particularly in the ArchEdit syntax-directed editor. ArchStudio 3 used a form of model reflection based on Java reflection mechanisms. Specifically, it explored the class structure of the data binding library as a source of information about the xADL language. If the class used to encode an architecture's structure had a `getAllComponents(…)` method, for example, ArchEdit could infer that structures could have zero or more component elements as

children. As the data binding library interfaces became more complex, this solution became less and less elegant and special cases had to be encoded in ArchEdit to deal with situations when ArchEdit would, for example, erroneously interpret a new `get(…)` method as a new child attribute. In ArchStudio 4, explicit metadata interfaces were added to the data binding library, generated by Apigen. ArchStudio 4's ArchEdit uses these metadata interfaces to generate its user interface, and the ArchEdit code is substantially simpler.

5.3.7 Make it Possible to Index into the Model

The model repository should make it possible to uniquely identify any element within the model. Being able to index into the model in this way is critical for many operations, for example, indicating the location of an error found during architectural analysis. There are different strategies for indexing into a model. The first is to tag each element with a unique, persistent identifier that can be used to identify the element. The second is to provide other location indicators—path-based, structure-based, or (at worst) line number.

Early versions of ArchStudio 3 relied on xADL elements having unique identifiers. However, not all xADL elements have unique identifiers—only those that are intended to be XLink targets. A component, for example, has a unique identifier, but its description does not. This made it difficult to build tools that referenced these less-important elements. A second mechanism was added called XArchPaths to rectify this. An XArchPath refers to an element via a path to that element, starting at the root of the XML tree and working its way down through descendants of the root until the target element is reached. This mechanism is similar to the XPath [33] mechanism.

5.3.8 Optimize Speed of Access to the Model Repository

As noted above, the model repository is the most-often accessed component in an architecture modeling environment. Tools are constantly reading from and writing to architectural models as users make changes. As such, the performance of the environment itself is largely dependent on the performance of the model repository. Given a choice between increasing speed and using additional memory, speed should be given preference in nearly every circumstance. Additionally, read operations are much more prevalent than write operations, so if needed strategies such as caching can be employed to improve read performance.

In ArchStudio 3, the importance of xArchADT's performance to the overall performance of the environment was not initially well-understood. The use of the C2 style greatly increased latency between editor components and the architectural model. This caused certain operations, such as opening a model for the first time in Archipelago, to be slow—taking up to several minutes on large models. Later builds of ArchStudio 3 partially addressed this issue using a bulk query mechanism, in which editors could request subsets of a xADL document all at once instead of piece-by-piece. Because this substantially reduced the amount of round-trips to xArchADT, the performance of some operations increased up to tenfold. However, components wishing to use this mechanism had to be recoded to call the bulk query interface instead of the ordinary xArchADT interface. In ArchStudio 4, the use of the Myx architectural style allows direct, low-latency synchronous calls on the xArchADT component, and the bulk query mechanism became unnecessary.

5.3.9 The Interface to the Model Repository Component should be Remotable

This constraint is concerned with the coupling between the model repository component and other components. In designing the interface to the model repository, one concern should be to make that interface *remotable*. A remotable interface is an interface that is callable across process or host boundaries. In effect, this means that the model repository cannot make the assumption that it shares memory with other components (since it may be running on a different host entirely), and all data passed through the interface must be serializable—that is, turned into a byte stream for transmission across a network or other interprocess communication mechanism. Making the interface for the model repository component remotable accomplishes two things. First, it helps to ensure that the only access to the repository is through the defined interface, and not through accidental side-effects that can occur in a system where memory-sharing is allowed. Second, it allows the repository to run in a separate process or host than the other components. If the environment moves from single-user to a multi-user, the ability to have multiple users in different processes access the repository simultaneously over a network will be critical.

xArchADT exposes a completely remotable interface. Although ArchStudio is a single-user environment, inter-process communication has been used in ArchStudio's history to access xADL documents via xArchADT. For example, an early integration of ArchStudio 3 with Eclipse ran ArchStudio and Eclipse in separate processes, and communication was facilitated by the use of xArchADT's remotable interface.

5.3.10 Models Must Be Allowed to be Partially Incorrect

Any architecture modeling notation will be laden with assumptions about syntactic and semantic correctness. Tools are especially easy to build if there is a guarantee that all these assumptions are met, so it is tempting to enforce these constraints constantly in the tools or in the model repository itself. By never letting the model enter an incorrect or inconsistent state, potential problems can be avoided. However, doing so often has a serious negative effect on usability. Such requirements often force a user to make decisions only in a particular order, and make many decisions prematurely.

Consider if this were the case for ArchStudio and xADL. No element could be created without a description, for example. No component or connector could be created without creating its type first. No link could be created without its endpoints being defined first. These limitations would undermine the usability of the environment substantially. The need for consistency must be balanced against allowing exploratory and partial designs. xArchADT requires that the basic xADL syntax be followed (it is not possible to add completely arbitrary elements to the xADL model), but things like cardinality restrictions (e.g., zero-or-more, exactly one) and XLink constraints (e.g., must include a type link) are relaxed to allow incomplete models. Syntactic and semantic constraints can be verified later using Archlight.

5.4 Visualization Design Principles

The design principles in this section apply to the construction of visualization support for stakeholder-centric architecture modeling.

5.4.1 Conform to Users Expectations

Users never come to a visualization as *tabula rasa*—blank slates. Every user of an architecture modeling environment will have had extensive experience with visualizations, depictions, and interaction methods used in similar tools. This experience establishes a set of expectations as to how applications should behave. Some of these expectations are guided by platform user interface and depiction standards; some of them are simply de-facto behaviors implemented by many similar tools. User expectations help them to work productively in unfamiliar tools. Encountering a tool that does not fit those expectations is often a jarring or disturbing experience. To the extent possible, visualizations should conform to known expectations. Established user interface patterns and conventions should be followed. If the environment is built atop a substrate or framework, then the environment should also obey the constraints of that framework.

ArchStudio attempts to conform to established expectations throughout its user interfaces and depictions. It uses platform-based user interface widgets to normalize data depiction and entry. Archipelago emulates the user-interface techniques of many similar software systems such as PowerPoint and Visio—the way you resize a box in all these applications is effectively identical. This is not to say that environments should never innovate: Archipelago's use of the mouse wheel to scroll is nonstandard but extremely intuitive; its spline editing features in which two close points coalesce into one and double-clicking on the spline creates a new break are unusual, but natural. ArchStudio also attempts to be a good 'Eclipse citizen,' following established Eclipse conventions for tool integration. ArchStudio preferences appear as part of the Eclipse

main preference panel. ArchStudio tools appear in Eclipse's context menus when a user is manipulating an XML document. Services such as file management are left entirely to Eclipse and ArchStudio simply hooks them.

5.4.2 The Visualization Should be Kept Synchronized with the Underlying Architectural Model

The visualization must be a true and accurate reflection of the underlying architectural model, especially in an environment where multiple tools or visualizations can access the model. If this is not the case, it can lead to user confusion, or worse—error states. Imagine if a component is removed from an architecture, but not its visualization. What happens when the user attempts to modify that component? Keeping the visualization and the model consistent avoids these problems. The user should not have to intervene in keeping them consistent, for example, by invoking a "refresh" or "resynchronize" operation—the changes should be live and in real-time. Consistency must be maintained both ways: if the visualization is changed, the underlying model should be changed to match. Likewise, if the underlying model is changed from the current visualization or some other tool, the visualization must be updated. If the model repository emits events whenever the underlying model changes, as suggested in Section 5.3.3, these events can be used to update the visualization. Care must be taken to avoid an infinite loop, however, in which the visualization is changed, that changes the model, the model emits an event to reflect this, the original visualization receives that event and updates itself, which makes another change to the model and so on. This is best rectified by having the visualization check each event

from the model repository. If the visualization is already consistent with the change contained in the event, the event is ignored.

In ArchStudio, all visualizations (ArchEdit, Archipelago, Archlight, the Product Line Selector, and so on) make changes to xArchADT and listen to xArchADT events. Whenever the user makes a change in any of these visualizations, the change is immediately recorded in xArchADT. When an event is received, the visualization is updated to match. The strategy used to update the visualization is generally to use the event to determine where in the architecture the change occurred and do a refresh of only that part of the architecture. A refresh may perform redundant operations. For example, if Archipelago receives a message indicating that a component's description changed, the entire depiction of the component (color, border, label, and so on) are all updated—even though only the description changed. This localization of changes helps reduce the amount of code and complexity required to process events, at the cost of an insignificant amount of performance.

5.4.3 A Visualization Must not Assume it is the Only Visualization

In an environment that integrates multiple tools and visualizations, none can afford to assume that it is the only visualization. Making this assumption might take on several forms. The visualization may assume that it has exclusive access to the architectural model, or that it is the only visualization making changes to the model, violating the principle described above in Section 5.4.1. The visualization may assume that it can store architectural changes as internal state and commit them to the model at a later time, denying other visualizations access to these changes. The visualization may be modal or have significant modal elements, preventing the user from switching to

other visualizations. Although all these behaviors can make it easier to implement a visualization, they are all undesirable. The goal of a multi-visualization environment should be to provide users with a fluid experience, allowing them to switch among visualizations freely and make changes in any one of them. This is a reification of the principle of “reflection in action” [147], in which designers are continually informed of the effects of their actions so that they may judge those effects and make subsequent decisions based on them.

In ArchStudio, all visualizations are constructed to send any user-initiated changes to the architectural model immediately. Each visualization also responds to change events from the architecture by updating its own internal representation and depictions immediately. Modal interfaces are eschewed (more on this below in Section 5.4.11).

5.4.4 Use Syntax-Directed and Reflective Visualizations to take advantage of Model Structure

With knowledge of the architectural meta-model, visualizations can be created that interact with architectures based on the structure of the model itself. These visualizations are limited, because they *only* have structural knowledge to use—additional semantics must generally be supported in other tooling. However, these syntax-driven visualizations can do something that semantically-aware visualizations cannot: they can adapt automatically to changes in the underlying structure or notation. This is a key advantage in a stakeholder-driven modeling environment, where end-users can change or evolve the underlying notation in myriad ways.

ArchStudio includes the syntax-directed editor, ArchEdit, which bases its user interface entirely on the structure of xADL as expressed in the xADL metadata classes in the data binding library. Although ArchEdit's interface is crude compared to, say, Archipelago's, no additional coding is required to get it to work with new schemas. This helps to encourage end users to create new schemas, since they will have some measure of editing support available immediately, without having to write new code.

5.4.5 Consider Using a Separate, Local Visualization-Centric Data Model for Performance

Visualizations are often one of the most performance-critical parts of an architecture modeling environment. The user directly interacts with them and expects real-time performance: when the user clicks on something, they expect a result immediately thereafter. For this reason, storing all visualization data in the architectural model is not always a good solution, especially if the latency between the visualization and the architectural model is high (as occurred in ArchStudio 3, for example, or might occur in a distributed environment where the model is available on a different host). For these reasons, an internal visualization-centric data model can be employed within the visualization component(s) that stores additional information needed by the visualization. The cost of this is that the visualization information will not be available to other tools as part of the architectural model, so a careful tradeoff analysis must be conducted to balance these two concerns.

In Archipelago, the xADL model of the architecture and the BNA model, which contains the graphical depiction of the xADL model, are kept separate. The xADL model resides in xArchADT and the BNA model resides in Archipelago itself. The two

are kept consistent through BNA logics. Solely visual changes to an architecture made in Archipelago, such as moving a box representing a component or re-routing a link, are not immediately reflected in xADL, since these are simply visual representations and are not semantically important. Since visual representations still need to be stored across sessions, some visual details are written to xADL in the form of "rendering hints" whenever the file is saved or closed. Common rendering hints include box positions, spline routing, colors, fonts, and so on. Because Archipelago is not constantly interacting with xADL and translating its internal BNA model objects into xADL elements and attributes, this approach helps maintain good performance in Archipelago. The drawback is that other editors do not have access to this visual information in real-time. To date, no other ArchStudio editor has needed this hint data, but if one were developed the decision of when and how to store rendering hints would need to be revisited.

5.4.6 Keep Visualization Models Independent from Rendering

Any visualization tool will have data that needs to be visualized—from the architectural model, from its own internal visualization model, or both. This data should be kept as separate as possible from the functionality that renders the data. This gives a visualization tool maximum flexibility to change how it depicts the same models, and also insulates tools from changes in underlying graphical APIs and frameworks.

Archipelago's data comes from both the xADL model and the internal BNA model in the form of Things. This data is kept largely separate from the functionality in Archipelago that actually draws and manipulates the Things—this functionality resides primarily in the Thing Peers. When it came time to port ArchStudio 3, developed atop

the Swing framework, to ArchStudio 4, based on SWT, it was possible to reuse a substantial amount of Thing code while spending the bulk of the effort on rewriting peers.

5.4.7 Persist All Relevant Visualization Data in the Architectural Model

A visualization environment will often contain visual information about how an architecture should be depicted, as well as data such as user preferences that do not semantically affect the architecture. This data should be stored in the architectural model anyway. One alternative is simply to discard this data between sessions, which will mean that the visual appearance of an architecture will be lost every time the environment restarts. This is obviously unacceptable from a usability standpoint. A second alternative is to store the visualization data in a separate file from the architecture description. This is problematic for several reasons. First, users must now remember to move, copy, and manage the architectural model and visualization data file in unison. Second, if users edit the architecture description outside the target editor, there is a good chance that the architecture description and its visualization information will get disconnected or out-of-sync.

An early ArchStudio project, Visio for xADL [134], was an effort to adapt Microsoft Visio to work as ArchStudio's visual editor (Archipelago had not yet been developed). Visio for xADL stored architectural model data in xADL files, but stored the visual representations of that data in Visio files. It was discovered that keeping these files in sync was very difficult, and editing the xADL files in another editor, such as ArchEdit, without Visio being open to track the changes and update its own file, caused

errors and other problems to occur. Archipelago stores all its persistent visual data in xADL rendering hints whenever the xADL file is saved.

5.4.8 Develop Visualization Behaviors based on Aspects, not Elements

In architecture visualizations, particularly symbolic visualizations like those used in Archipelago, the same kinds of operations are often applied to different types of objects. For example, components, connectors, links, and architectural states and transitions can all be removed from an architecture description. Each one can have its textual description edited. Each can have its visual stacking order changed. Each can be moved around the canvas via click-and-drag. Components, connectors, and states, all represented as rectangles (or rounded rectangles) are all resized by using 8 compass-point resize handles displayed on the edge of the box that are clicked and dragged. Splines such as links and transitions are also edited by dragging resize handles, although they perform a slightly different function. The implication from these cross-cutting commonalities is that the traditional object-oriented view of tying behaviors to particular kinds of elements may result in a significant amount of duplicate functionality. This can be overcome by implementing global, cross-cutting behaviors that operate on classes of elements, rather than embedding behaviors in the elements themselves.

For example, in Archipelago, Logics tend to operate on classes of symbols, rather than specifically identified symbols. A Thing element can identify itself as “bounding-box resizable” by implementing a facet interface. A BNA logic knows that the result of clicking on this Thing should be to display resize handles at the compass

points of the bounding box, and change the Thing's bounding box when these resize handles are dragged. Any new Thing that implements the "bounding box resizable" facet gets this behavior automatically—the logic does not need to be changed at all. This reduces redundancy and improves the reusability of many behavioral logics.

5.4.9 Provide Visualization Embedding and Coordinated Visualizations

Architecture descriptions are complex artifacts. Often, even within a single architectural concern (i.e., view), it is easy to get lost among the amount of information presented. Some of these concerns may be hierarchical, where different layers of information are displayed embedded within one another. Users need the ability to visualize multiple levels of the architecture simultaneously, and also to create split or duplicate windows of the same visualization, each focused on a different part of the architecture. This is generally accomplished by developing visualizations following the model-view-controller pattern [132], wherein the model refers to both the architectural model in the model repository and any local data stored by the editor.

The version of Archipelago included in ArchStudio 3 did not provide these capabilities: users could view the internal architecture of a component, but would have to switch to an 'internal visualization' to edit. Users were also prevented from opening up multiple Archipelago windows on the same architecture description, since no BNA mechanism existed to coordinate the BNA model data. These features were the source of much consternation for ArchStudio 3's users. Archipelago introduced the concepts of BNA views and worlds, which allow both hierarchical editing and multi-window editing. Other potential applications for this technology are the display of thumbnails

depicting the entire architecture, or mirroring the Archipelago view of an architecture on a secondary display—perhaps one the size of a wall composed of multiple monitors.

5.4.10 Context-Sensitive > Global

A new editing feature can be added to an architecture visualization in one of two primary ways: as a global feature or as a context-sensitive feature. The most common place this arises is on menus, for example: the feature can be exposed from a global menu available at the level of the application, on a ‘right-click’ context menu, or both. In a stakeholder-driven modeling environment, context sensitive features are almost always preferable to global ones. Because users can extend the environment in unforeseen ways, global features tend to pile up quickly. Menu bars, global toolbars, and feature lists become cluttered and unwieldy, and the usability of the environment drops. Context-sensitive features mitigate this effect by presenting the feature only in the context in which it is actually used. An option applied to a component, for example, will not be available to a user working with an interface.

Archipelago, ArchEdit, and ArchStudio itself make nearly exclusive use of context-sensitive feature integration. For example, ArchStudio does not add any top-level menu items to Eclipse at all; ArchStudio’s contributions to the global Eclipse environment are found limited to the Eclipse preference subsystem, for example. Most operations in ArchEdit and Archipelago occur through context popup menus (‘right-click’ menus). As a side effect, users do not have to move the mouse as much since context-sensitive user interface elements always pop up at the current mouse location.

5.4.11 Modeless > Modal

Architecture visualization features can work in one of two primary ways: modeless and modal. ‘Modal’ is somewhat of an overloading term; here it is used to indicate any feature that creates or facilitates the creation of a set of (mutually-exclusive) visualization *modes*. For example, the text editor ‘vi’ is a modal editor: the user is either working in text-entry mode or editing mode. Most visual editors that use a tool palette also tend to be modal: each tool puts the editor in a different mode (selection mode, text editing mode, symbol reshaping mode, and so on). A second form of modality occurs when a tool takes exclusive control of the user interface and bars the user from using any other part of the tool until the user completes some operation. Many applications, for example, will throw up modal dialog boxes that prevent the user from doing anything in the application until the dialog is dismissed. In a stakeholder-driven modeling environment, modality should generally be avoided as it negatively impacts the usability and extensibility of the environment. Usability is reduced because every user action becomes a two-step process: put the visualization in the correct mode, and then perform the desired action. Modal dialogs also negatively impact usability, especially in an environment that incorporates many tools. Whenever a modal dialog comes up, the user is prohibited from switching to other tools—even if only gathering information to populate the modal dialog. Maintainability is more difficult because each operation’s implementation must now consider not just the operation performed, but the mode the visualization was in when the operation occurred. Modeless user interfaces, where the user can choose any valid operation at any time and switch between tools freely are preferable in nearly every case.

ArchStudio goes to great lengths to use modeless behavior wherever possible. For example, in Archipelago, when the user edits the description of an element, a text field is placed directly on the element in the BNA canvas that stays open until the user clicks off the text field to dismiss it. The text field is not modal, and the user can switch to other editors and come back without it disappearing. Multiple text fields can be open at any given time and the user can, for example, copy from one to the other easily. Archipelago is one of a very few symbolic editing tools that does not use a tool palette. The most common operations are done by default, and other operations are available through context menus, keyboard commands, or even mouse gestures. For example, selection is the default behavior when clicking on an element in Archipelago; label editing is available through a context menu.

5.4.12 Composition > Inheritance

In constructing extensible visualizations, many features will often be combined into a single element. There are generally two ways to achieve this combination: through *inheritance*, in which an element derives itself from one or more base features, which may themselves be combinations of extensions, or through *composition*, in which independent or interdependent extensions are composed into a target element. The distinction here is one of an *is-a* versus a *has-a* relationship. For example, with inheritance, the visual representation of an architectural component as a rectangle *is-an* element with a colored field, and it *is-an* element with a border, and it *is-an* element with a label, and so on. With composition, the representation *has-a* colored field, it *has-a* border, and it *has-a* label.

In general, composition is the more flexible of the two approaches. This is especially true in contexts where only single-inheritance is possible, such as in Java programs. Multiple inheritance using mixins [153] has a tendency to resemble composition rather than inheritance. The tension between composition and inheritance in visualization framework design is often referred to as the difference between monolithic (inheritance-based) and polylithic (composition-based) designs [16].

Using composition makes it easier to define independent features, and thus reuse those features. For example, BNA 3 primarily used inheritance to extend complex objects. So, a simple box Thing would be extended to form component Things or connector Things, and so on. This worked well enough until some other element—perhaps a state Thing representing a state in a statechart—needed a property already defined in the component Thing. Here, the state Thing could inherit from the component Thing, but this relationship is false: an architectural state is not a component. The inheritance is only used to steal features from the component Thing. BNA 4 uses a much more composition-centric approach. Common features are incorporated into facets, which act like aspects and can be applied to any Thing. Beyond this, individual graphical elements on the BNA canvas are often implemented as a stack of related things, connected via a BNA Assembly, as shown in Figure 45. Because these elements are split out in this manner, they can be more easily reused in other contexts.

5.5 Analysis Design Principles

The design principles in this section apply to the construction of analysis support for stakeholder-centric architecture modeling.

5.5.1 Plan to Integrate Multiple Analysis Engines

The kinds of tests and checks that can be run on an architecture description are diverse and complex. Invariably, different testing engines will need to be combined to provide the necessary breadth of analysis. Any architecture analysis framework or environment must take into account that more than one engine will be providing testing services and results. Therefore, the framework data and control structures need to be designed in a way that is independent of any given engine. Furthermore, the user will generally not care what engine is responsible for which particular test, so engine-specific details should be hidden if possible, or relegated to engine-specific preference pages if necessary.

Archlight provides simple, well-defined interfaces for integrating new testing engines into ArchStudio. Engines must provide a set of tests that they can run, be invoked by Archlight, and update issues they found in the Archlight Issue ADT. This simple, generic interface makes it possible to integrate a variety of model checkers in ArchStudio.

5.5.2 Leverage Off-the-Shelf Analysis Engines

Architectural analysis is rarely trivial or simple. At best, it involves checking assertions over the entire architecture description. At worst, it involves exploring huge state-spaces or developing and running complex simulations. The reward for performing these complex analyses is that they often yield non-obvious results that provide insight about a given architecture description and its quality. Because of the complexity, it is infeasible to develop the analysis tools for an environment oneself. Fortunately, there are a plethora of off-the-shelf, generic analysis tools available for

analyzing various properties of software and other models. These include constraint checkers like xlinkit [117, 118] and Schematron [90] and formal-methods-based model checkers such as Alloy [87], SPIN [78], and FDR [57]. Maximizing the effectiveness of architectural analysis and getting the highest quality results at the lowest possible cost will invariably involve integrating one or more of these off-the-shelf analysis tools. An architecture analysis framework, therefore, should provide mechanisms through which such tools can be wrapped to participate in the framework. One drawback of these off-the-shelf analysis tools is that they tend to consume many system resources, run in batch mode and do not provide incremental results. Because of the complexity of what they are doing, these are often inevitable tradeoffs for the analysis power offered by the tools.

Early versions of ArchStudio 3 provided analysis services through a framework that used design critics [136, 139, 140] as the primary analysis tools. Each design critic was a small Java program, perhaps ~200 lines of code, that checked some architectural property. These versions shipped with a small number of design critics (5-6) that checked basic architectural constraints. Later, a team of undergraduates implemented a set of additional design critics (8-10) that checked properties of product-line architectures; this took approximately 10 weeks. These experiences revealed that handwritten Java design critics, while still only a few hundred lines of code, were still relatively expensive to write and maintain given their limited functionality. Later versions of ArchStudio incorporated the TRON and Archlight frameworks, in which constraints were primarily checked with the Schematron off-the-shelf XML constraint checking engine. With Schematron, tests that took 100-200 lines of Java could be done

in 5-10 lines of XML code. In this case, the use of an off-the-shelf testing engine allowed ArchStudio to incorporate many more tests at a much lower cost.

5.5.3 Give Users Control Over what is Tested, and When

Architecture descriptions are rarely 100% complete and correct. As they are created and evolved, the descriptions ideally become moreso, often one property or constraint at a time. Some constraints will never be achieved—constraints for architectural styles that the architecture is not using, for example. Architecture analysis frameworks need to recognize this and provide mechanisms for users to control, on a per-document basis, which tests are run and which are expected to pass. Optionally, users might be able to customize each test with something like a default severity, so failing a test that would normally be critical is temporarily reported as a warning until the architecture description has been sufficiently developed so as to pass the target test. These tests, or at least references to them, should be stored with the architecture description to ensure that they persist across sessions.

Archlight provides users with this kind of fine-grained control, as explained in Section 4.6. Each test provided by any Archlight engine can be applied, disabled, or unapplied on an architecture. The state of each test is stored directly in the xADL document.

5.5.4 Ensure that the Analysis Framework is Well-Integrated with Other Tools

The ability to find issues and errors in an architecture analysis framework is only part of the framework's value. Significant value is added to a framework that

incorporates tight integration with other tools in the environment. For example, it is critical that users be able to navigate from a test result to the element that caused that result to occur—not just the test that passed or failed, but the specific elements that were evaluated to obtain the result. Such capabilities put the provided analyses in context.

In ArchStudio, tests, test results, and causes are all linked through two mechanisms. The first is xArchADT, the component through which all data integration occurs. The second is the editor-focusing mechanism, which allows an ArchStudio tool or editor to request that another editor focus on a particular element. Tests in Archlight are specifically designed to capture the elements that caused a test to fail, and the identifiers of these elements are used by the editor focusing mechanism to navigate to them.

5.5.5 Make it Easy for Users to Define New Tests

Users should be able to perform three primary activities within an architecture analysis framework: enabling, disabling, and running tests, writing and incorporating new tests, and incorporating new analysis engines. Enabling, disabling and running tests should be trivial to do through the framework user interface. Incorporating entirely analysis engines is expected to be complicated—wrapper code will be needed, deployment and licensing issues will have to be worked out, and so on. Writing new tests that leverage existing engines, however, should be made relatively easy if possible. It is often the case that the constraint languages for model checkers are compact and understandable, even if the models upon which they operate are not. The environment should provide an easy mechanism by which users can write their own tests and

incorporate them in the environment as if they were default tests that ship with the environment itself. Users should ideally be able to integrate and refine these tests without restarting the environment every time, as well.

In ArchStudio, users can easily add new Schematron tests to the environment. A user preference panel allows the user to select a set of directories containing Schematron tests on the local machine. Those tests are then incorporated directly into the Archlight user interface as if they were part of the environment itself. These tests can be reloaded dynamically as ArchStudio runs, giving users an opportunity to debug and refine tests that they write.

5.5.6 Be Prepared for Fundamentally Different Styles of Analysis

The approach used by ArchStudio and described in this section is biased toward a style of analysis based on model-checking. The architecture description is transformed into a model-checker's input language, checked against a set of constraints or propositions, and the results, if any, are turned into issues. Model-checking is not the only analysis paradigm available, of course. Dynamic analysis and simulations, for example, work differently. For vastly different kinds of analysis, a model-checking-based analysis framework may become less and less useful. For this reason, it is important to separate out services like issue display and issue management from test execution, and also to provide tool integration mechanisms outside the analysis framework to incorporate different types of tools.

Archlight's issue management is mostly handled by the Archlight Issue ADT, which is connected to Archlight and its analysis engines. However, it is possible to hook directly into this component without being part of Archlight, in which case the test

results from a non-Archlight tool can still be displayed in Archlight with other identified issues. Secure xADL [135] used this approach to take results of a dynamic simulation and incorporate them “live” into the Archlight UI.

5.6 Environment Design Principles

The design principles in this section apply to the construction of stakeholder-centric architecture modeling environments in general, as well as the tool-integration mechanisms used therein.

5.6.1 Construct the Environment Using an Architecture-Centric Approach

Using an architecture-centric approach can confer many benefits in the construction of an environment in which tool integration is a primary goal. Certainly, not all architecture-centric approaches are equal. However, experience has shown that architecture-centric development, particularly the selection and use of an appropriate architectural style, can confer known qualities on systems designed in that style. Because the environment under construction is also an architecture modeling environment, the environment should ideally be modeled in its own approach. This allows the environment to act as a testbed for itself, and helps to validate the environment’s effectiveness. Of course, depending on the modeling approach embodied by the environment, this may not be appropriate: a modeling environment focused on embedded systems or some specific domain may not use an approach that is suitable for self-application.

ArchStudio is developed using an architecture-centric approach throughout—its own. It is modeled in xADL 2.0, visualized in Archipelago, and checked in Archlight. It is implemented in the Myx architectural style and its components are implemented in the myx.fw architecture implementation framework. The xADL description of ArchStudio ships as part of the implemented system, and is used by the ArchStudio bootstrapper to instantiate and configure the system. This approach improves ArchStudio in many ways. First, its architecture is always in sync with its implementation, since the architecture is part of the implementation. Second, the Myx style facilitates the integration of heterogeneous, off-the-shelf tools with custom components. Myx and myx.fw help to increase reusability of components by forcing them to interact through well-defined provided and required interfaces, and to avoid sharing memory.

5.6.2 Keep Components Loosely Coupled

Loosely-coupled components are more reusable and flexible than tightly-coupled components. Loose coupling can be achieved through a number of strategies. A primary strategy is the use of explicit dependencies and interfaces for communication. By defining components in terms of the services they provide and require it is much easier to reconfigure them, interpose proxies and other connectors, and so on. Enforcing a no-shared-memory rule among components further reduces coupling, and allows components to be distributed across process and machine boundaries.

ArchStudio components and connectors communicate exclusively through Myx interfaces: any component providing a particular service can be swapped out for another component providing the same service without recoding other components. Due to

constraints imposed by Eclipse, not all ArchStudio components act as if they run in separate memory spaces, however this constraint is obeyed to the extent possible.

5.6.3 Leverage Off-the-Shelf Environment and Tool Integration Technologies

Using an already-available tool-integration platform such as Eclipse, or even the UNIX command-line, as the basis for an architecture-centric development environment can save development time and costs, increase the quality of the resulting environment, and help new users acclimatize to the environment more easily. It saves time and costs by providing common services such as file and preference management, as well as by providing user interface scaffolding. It increases environment quality because off-the-shelf frameworks are often developed by organizations with substantial resources to test and ensure product quality. They also have established user-bases that provide feedback. This user-base will also come with experience with the platform, and so users that come to a new environment based on that platform will likely have an easier time using it. Additionally, if a platform has substantial support from other tool developers, it will make it easier to integrate a new environment with these other tools.

ArchStudio 3 was constructed as a stand-alone application, while ArchStudio is built atop the Eclipse platform. ArchStudio 4 implemented the same functionality as ArchStudio 3 in less code, because fundamental services provided *de novo* by ArchStudio 3 were provided by Eclipse in ArchStudio 4. Eclipse has a large open-source development and user community, and ArchStudio gets to take advantage of bug fixes and feature enhancements to Eclipse at little or no cost. Eclipse also allows

ArchStudio 4 users to quickly switch between architecture modeling and implementation, since a complete Java development toolkit is included in Eclipse.

5.6.4 Consider using Platform-Independent Implementation Technologies

Programming language and runtime developments, particularly in the Java programming language, have greatly increased the ability of applications developed once to be deployed and used on multiple operating system and hardware platforms. Architecture modeling activities are often platform-neutral, and increasing the number of platforms upon which a modeling environment runs can increase the environment's userbase. One caveat, however, is that platform-independent technologies often come at a cost. Application performance of a platform-independent application may be worse than that of a native application. This gap is closing due to improvements in just-in-time (JIT) compilers [156]. Additionally, platform-independent user-interface toolkits can look and perform differently than native user-interface toolkits. Such is the case with Swing, a pure Java UI framework. An alternative framework, SWT, uses Java wrappers to provide Java access to native widgets.

ArchStudio and Eclipse are both implemented in Java, and both run on most major platforms. While most ArchStudio users run it on Windows, a few use it on Mac OS X, Linux, and UNIX operating systems. Because Java is used, few changes or special cases are needed for each platform within the ArchStudio code. ArchStudio 3 used Swing, while ArchStudio 4 (like Eclipse) uses SWT for its user-interface toolkit. SWT's user interfaces are nearly indistinguishable from native platform applications; this was one of the reasons the Eclipse port of ArchStudio was done. However, SWT

has more vagaries and cross-platform bugs than Swing—certain minor features do not work correctly on the Macintosh platform due to these bugs, for example. SWT developers continue to rectify these issues as they are identified.

5.6.5 Separate Data from Computation from User Interface

Functions in an architecture modeling environment can be broken up into three rough groups: data storage and access, computation, and user interfaces. Maintaining this separation at the component level improves the flexibility of the environment. Data is stored in data store components, which provide access to complex structures (including architectural models themselves). Computation components implement the algorithms that create and manipulate the data. User interface components allow the user to configure and execute the algorithms provided by the computation components. This three-layer strategy allows multiple algorithms and services to operate over the same data, and multiple user interfaces to access those algorithms.

ArchStudio demonstrates this separation throughout. xArchADT is the primary data store component, although there are others such as the Preference Store and the Archlight Issue ADT. The Product-Line Selector and Archlight testing engine components represent different algorithms that can be run over architecture descriptions stored in xArchADT. Neither of these components has a user interface, however—the user interfaces are provided by the Product-Line Selector and Archlight user interface components, respectively. This allows, for example, the Product-Line Selector to be invoked from multiple sources—its own user interface, or an Archipelago plug-in as occurs in the Stateline project (see below in Section 6.1.4).

5.7 Schema Design Principles

The design principles in this section apply to the design of new architecture modeling schemas, as applied in the context of xADL 2.0.

5.7.1 Start with the Empty Set

In designing a composite modeling language, the obvious first question is where to start. The best place to start is with the empty set: a basic schema that adds no features to the language except perhaps a container in which to add arbitrary new features. If needed, basic generic elements such as identifiers and descriptions can also be defined. By starting with no assumptions about what features will be part of the language and only focusing on a container for them, the modeling environment begins from the most logically flexible point.

xADL 2.0 defines a top-level XML element, `<xArch>`, which exists at the root of all xADL documents. That element is allowed to contain zero or more additional elements of any type. The children of this element tend to be first-class concepts in xADL: architectural structures, architectural types, analysis data, statecharts, and so on.

5.7.2 Practice Bottom-Up Design

All architecture modeling approaches and concerns have a set of core concepts or elements that are used as the building blocks of reasoning within that approach or concern. For most first-generation ADLs, these core concepts are components, connectors, interfaces, and links. For statechart modeling, these core elements are states and transitions. These core concepts are described in schemas that extend the empty set. Core schemas should add elements with as little detail as possible to give those

elements maximum reusability. Additional detail is then added in extension schemas that specialize and add on to these core elements in a layered fashion. This supports divergent interpretation and extension: saying as little as possible about what constitutes a component, for example, maximizes the kinds of components that can be created in the modeling language through extensions.

xADL schemas follow this pattern throughout. For example, the basic definition of a component requires that it only have a unique identifier, a description, and a set of zero or more interfaces (which are also defined at this level to have only an identifier and description). This minimal definition is then augmented with type pointers, which refer to component types, which can then be extended with abstract implementations that are then specialized as, e.g., Java source code implementations, which are further refined as Eclipse Java source code implementations, and so on. Modelers wishing to capture components can import as much or as little of this detail as they want.

5.7.3 Consider Adding New Concepts as First-Class Entities

When adding a new concept to a modeling language, a primary consideration is ‘where’ in the language to add it. In general, new elements can be added as subordinate to existing elements, or they can be added as independent, first-class entities. Consider the case of architectural links. Links connect interfaces on components and connectors. Link data can be embedded in the interface descriptions of components and connectors—this is the strategy taken by, for example, HTML. Link data can also be externalized and stored as a first-class element independent of components or connectors. Each approach has advantages and disadvantages. Embedded links are, to an extent, modified along with their endpoints—when a component interface is

removed, links to that interface go with it, for example. First-class links, on the other hand, can connect things other than interfaces, and can be specialized in more ways (to create multi-way links, for example). When faced with this choice, first-class entities are generally more flexible, but are also somewhat harder to manage. This is a trade-off that must be evaluated on a case-by-case basis, but the preference in schema design should generally be given to flexibility. Tools that maintain indexes of first-class entities can help with the management of relationships.

In xADL 2.0, a combination of first-class and subordinate elements is used. In general, subordinate elements are only used when there is a clear containment (i.e., 'is-part-of') relationship between two elements. An interface is part of a component, so a subordinate relationship is used there. However, links are not parts of components or connectors, so links are implemented as first-class objects. A component has a type, but the type is not part of a component—so a component type is a first-class element.

5.7.4 Be a Minimalist—Avoid Redundancy

Within a modeling language, redundant data can make interpretation or processing of models much easier. With regard to the design principle above—it is possible to include a language feature as both a first-class *and* subordinate element: the data is simply duplicated in both places. While this seems like a boon from a processing perspective, it greatly increases maintenance and consistency management costs. What if the two representations got out-of-sync, and the first-class element asserted something different from the subordinate element? In general, schemas that add new modeling features should add the minimal amount of data needed, even if it makes processing less

convenient. Wherever a relationship can be inferred by tools, the relationship should not be explicitly specified in the language.

The xADL VERSIONS schema describes how to model a version graph. Version graphs are directed acyclic graphs, with explicit parent-child relationships among nodes. Graphs can also interconnect with one another, meaning that multiple graph roots may connect the same nodes. xADL models the nodes as first-class entities. Three possible choices for implementing the relationships between these nodes: capture parent relationships, capture child relationships, or both. xADL captures only the parent relationships: a node refers to its set of parents, but not its children. To find the children of a particular node, a tool must find all other nodes that have the target node as one of their parents. This makes traversing the version graph more difficult than if both parent and child relationships were stored in xADL. However, storing both relationships would require keeping them consistent: what would it mean if node A was the parent of B, but B was not noted as the child of A? With only parent links, the relationship between A and B is clear. Child links could also be used, but this makes it slightly harder to 'slice' a version graph. By using parent links only, a new node can be inserted in the graph without modifying any other node. Also, any valid subgraph of a version graph can be taken by selecting a set of leaf nodes and their ancestors and omitting other nodes.

5.7.5 Create Permissive Syntax, Enforce Constraints in Tools

In keeping with the above principle, it is best not to try to specify every semantic and syntactic constraint in the language definition itself. While this clearly specifies one's intent as a schema designer, it leaves little room for differing interpretations that might be useful in different project or domain-specific contexts. When there is little

automated analysis tool support for a schema beyond a basic syntactic validator, then syntactic constraints are often useful. When paired with a flexible analysis framework like Archlight, however, the need for syntactic constraints diminishes. Tests in the analysis framework can test for canonical or even user-defined interpretations and constraints on elements.

In xADL 2.0, this property is seen throughout the language, particularly in the core modules. For example, while it is intended that architectural links connect interfaces on components and connectors, nothing in the language syntax prevents them from connecting the components and connectors themselves, or even completely arbitrary elements. A suite of Archlight tests, however, ensures that they do indeed connect interfaces. Users have the option of running these tests or not, of course, and a user that wanted to use links to connect something other than interfaces could easily add new Archlight tests that conformed to his/her use of links.

5.7.6 Composition > Subtyping

There are, in general, two ways to mix in a new property or decoration on an existing element in a modular language. How these mechanisms work is generally dependent on the underlying meta-language. The first mechanism is through inheritance: a base type (e.g., the definition of a ‘component’) acquires some new data through subtyping (e.g., it becomes an ‘optional-component’). An optional component is a component, but it also has some additional data that describes whether it is optional or not in a product-line. The second mechanism is through composition: a component has, as children, an unbounded set of ‘component extensions.’ Each extension can add detail to the component: to make it optional, to add implementation data, and so on. To

best support divergent extension, composition is a much more flexible solution than extension. If the underlying meta-language only allows single inheritance (as XML Schema does) then this is especially true. Single inheritance forces module designers to include artificial dependences between conceptually independent or orthogonal elements.

xADL 2.0 primarily uses inheritance to add detail to elements. Orthogonal extensions are added using artificial dependencies, as described in Section 4.1.1. This was an artifact of an early design choice, when the implications of subtyping vs. composition for divergent extension were not clear. Additionally, at the time it was considered likely that future versions of XML Schema would support multiple inheritance, but in retrospect this no longer seems likely. A future version of xADL will likely replace the existing subtyping relationships with composition relationships, and provide a translator to convert old instance documents to use the new extension format.

5.7.7 Provide Unique Identifiers for all (Important) Elements

Often, tools find it necessary to indicate or point to a specific element in an architectural model in a way that is persistent across sessions and document edits. In order to do so, elements need some persistent unique identifier. The nature of this identifier is not important. One way to generate these identifiers is to use an autonumbering strategy like those commonly used in relational database tables as primary keys. Another is to use globally-unique identifiers (GUIDs) [84], long digit sequences that are generated with procedures involving seeds such as the current time and a random number that make it statistically very unlikely to have a collision. GUIDs have the advantage that the IDs will not only be unique within a document, but unique

across documents. GUIDs are not strictly necessary; however, document-unique IDs suffice to create indices into documents.

xADL applies document-unique IDs to many elements for two reasons. First, to allow indexing into a document as described above. Second, because XLinks between parts of a xADL document require a unique identifier to identify the target of a link. The criteria for deciding whether to assign a unique ID to a xADL element is whether the element could possibly be a target of such an XLink. If there is any doubt, an ID is usually assigned just in case. Although xADL does not specify the format of the ID, most ArchStudio tools use some form of GUID. The xADL strategy has met with mixed results. The use of GUIDs simplifies ID generation somewhat (it is not necessary, for example, to keep a running "last ID generated" to generate the next ID). However, GUIDs are long (128 bits or more) and therefore increase the size of xADL documents significantly. Additionally, some projects, such as EASEL (described in Section 6.5.2) would be substantially easier to implement if each element in the xADL document had a unique ID. Future revisions of xADL may provide a unique ID on each element.

5.7.8 Consider Giving Elements Minimum Cardinality Zero

Often, designing language modules will result in several elements being defined in the same place. For example, in xADL, an architectural structure is defined to contain components, connectors, and links. It is possible to break up each case like this into separate extensions (e.g., one for components, one for connectors, and one for links) but this kind of extreme modularity can be detrimental to schema cohesion and understandability. A happy medium can be achieved by giving such elements minimum cardinality zero—that is, the document expects to include 0-1 or 0+ of the target

element. In this way, schema users that do not want to use a particular element can simply omit it without violating the schema syntax. This gives language modules increased flexibility without the added complexity of modularizing every new element.

xADL 2.0 does this profusely. In the above example, components, connectors, and links all have cardinality 0+, making it possible to define an architecture just by enumerating components, for example. Even components are optional, because one could imagine the structure of a middleware package consisting solely of connectors.

5.7.9 Avoid Unconstrained Extension through Weak Syntax

One temptation in language module design is to avoid using the rigorous syntax specification mechanisms available in the meta-language, and just allow unconstrained extension of the language. The classic example of this is providing an extension that allows users to decorate elements with arbitrary name-value pair property lists. This sort of extension simplifies language design greatly: users can annotate elements with any simple properties they want, in any format they want. It is up to language users to manage the property names and their formats. New data bindings almost never need to be regenerated, because new properties can be added without changing the language. This is a poor design choice, and undermines the reason for having a well-defined language with data bindings in the first place. It is one step away from using a completely generic meta-model (e.g., XML without schemas), and shifts the burden entirely to tools to agree upon and maintain consistent architecture representations. It also makes it very difficult for newcomers to understand the syntax of the architecture description language.

xADL schemas avoid the use of such syntactic elements to the extent possible. Despite many requests for a generic ‘properties’ schema from various xADL users, there remains no such schema. One minor exception is the rendering hints schema, which is used to encode visual and layout data for Archipelago. This schema established a very broad syntax for encoding hints, and did not specialize hints by type (e.g., bounding box hint, color hint, font hint, etc.) This was done to allow BNA model data to be written to xADL easily, even if BNA properties for a Thing changed. It was also assumed that this data would be used exclusively by Archipelago. Later projects (see Section 6.1) turned out to have a need to access these hints as well, however, invalidating this assumption. Future versions of xADL will likely include a more syntactically rigorous hints schema.

5.7.10 Write Schemas such that Readings are Still Valid with Extensions Stripped

In an extensible language, it is often the case that some particular tool will encounter an unfamiliar extension. When this occurs, the default behavior of most tools is to ignore the extension, and process the existing data if possible. Language extensions should be written with this in mind. That is, if possible, a document should have a valid or reasonable interpretation with the extension stripped off. Obviously, this will not always be possible, as some extensions add semantic detail that fundamentally alters the interpretation or meaning of the changed elements.

For example, consider xADL 2.0’s OPTIONS schema, which adds data about optionality to components and connectors. If this data is ignored, the result is an architectural description in which every component and connector is included in the

architecture. This may not be an architecturally valid interpretation (some choices may be mutually exclusive in practice) but it is still a reasonable one that can be processed by tools. For example, xADL's type consistency rules apply even in a situation where two conflicting components are included in the same architecture.

5.8 The Design Principles as Architecture

These design principles, as embodied in ArchStudio, can be seen to comprise architectural design decisions for ArchStudio. A question naturally arises as to how ArchStudio's xADL model embodies or captures these design decisions. Many of these design decisions can be seen in ArchStudio's xADL model. For example, the separation of data, computation, and user interface (Section 5.6.5) is seen across the ArchStudio description. The sequestering of architectural models in a single component (Section 5.3.2) that emits events upon changes (Section 5.3.3) is also evident. Many of the design principals are not immediately evident in the xADL model. This occurs for many reasons. First, not all design principals are expressed at an appropriate level of abstraction to be encoded in a xADL representation. Recall Figure 17, which expresses the spectrum of design decisions and notations: xADL is best suited to capturing design decisions that are rigorous and concrete—those on the right side of the diagram. It would be possible to capture a design principal such as "avoid over-engineering" (Section 5.2.7) in xADL, but it would be equivalent to a natural language expression. Because this decision is abstract, non-rigorous, and to an extent aesthetic, the value of expressing it in xADL is limited. It cannot be automatically verified, for example, nor can it be directly visualized.

A primary motivating factor behind the selection of ArchStudio design decisions that are modeled in xADL hearkens back to the motivation behind architecture modeling itself. As stated earlier in Section 1, *in order to make modeling worthwhile, the models themselves must provide more value than they cost*. In ArchStudio, the existing xADL model provides substantial value. It helps to visualize the relationships and dependencies between components in ArchStudio. More importantly, it is mapped to the implementations of each component, connector, and interface, and is part of the environment's implementation. It is used to define the configuration of ArchStudio itself, and ArchStudio can be easily reconfigured using Archipelago. Configuration changes can be quickly checked with Archlight. Modeling these aspects of ArchStudio has clearly increased the maintainability and understandability of ArchStudio enough that the costs of modeling have been recouped.

ArchStudio's evolution to date has been guided by a small and closely-connected group of developers, for whom maintaining more detailed or extensive ArchStudio model would provide limited returns. However, as evidenced below in Chapter 6, the community of users interested in ArchStudio and xADL is growing. Many individuals and organizations, for their own needs, have extended either xADL itself or ArchStudio, and this increases the value of rigorously capturing more of these design principles in ArchStudio's xADL description. Simply capturing these design principles is not enough—as noted above, the benefit must exceed the cost. This will happen, over time, in two ways. First, the cost of capturing some design principles will go down due to improved tool support in ArchStudio for capturing new types of design decisions—Archipelago extensions, new Archlight tests, and so on. Second, the benefits

of capturing design decisions will go up by encouraging more users (especially external users) to contribute directly to ArchStudio's development, and lowering the barriers to entry to create ArchStudio add-ons. For example, as ArchStudio documentation improves, the benefit of creating traceability between ArchStudio's xADL description and other ArchStudio artifacts will increase, which will require new xADL extensions. The costs of maintaining traceability will go down with the inclusion of supporting tools within ArchStudio. This is a virtuous cycle and reflects the same kinds of modeling decisions that must be made in any large software project.

In sum, not all of these design principles are expressed directly in ArchStudio's current xADL description. The effects of some decisions are indeed seen. Others are too abstract to be appropriately specified in xADL, and others do not yet achieve the cost-benefit balance that motivates their modeling. One thing that is sure is that as ArchStudio, its capabilities, and its community continue to evolve, the ArchStudio xADL description will continue to be evolved and adapted as well.

5.9 Summary of Design Principles

The design principles presented in this chapter address all key aspects of environment development. Each design principle is cataloged with rationale for why that principle is applicable, as well as how it impacted From a scientific perspective, these principles help to make the ArchStudio experiment repeatable. If someone wanted to design their own instance of a stakeholder-driven, multi-view architecture modeling environment, they could do so with guidance at all levels. The results in this chapter fulfill Hypothesis 3 of this research:

H3. These capabilities can be enabled by employing a particular set of design principles (which are identified) in the construction of the environment.

The many versions of ArchStudio that have been built and the applications of ArchStudio (described in the next chapter) increase confidence that these design principles are indeed effective. Additionally, as has been detailed above, many of these design principles have evolved along with ArchStudio. Here, alternatives have not only been considered, but implemented and deployed. In a small number of cases, the existing ArchStudio environment does not embody the principle completely, and part of the future work of this research is to refine ArchStudio further.

6 Evaluation and Application

The existence of ArchStudio and its implementation of extensible frameworks for constructing notations, visualizations, analyses, and tool integrations implies that it implements the functionality necessary to support stakeholder-driven architecture modeling. However, evaluating its effectiveness in this regard requires additional application. Throughout its development, ArchStudio and its constituent technologies have been applied in a wide variety of domains to solve a diverse array of architecture modeling problems. This chapter describes a selection of those application experiences.

The chapter focuses primarily on four experiences in which ArchStudio was applied in industrial settings: with The Boeing Company on a project involving software defined radios, with Lockheed-Martin on a project involving the U.S. Air Force AWACS aircraft, and with NASA's Jet Propulsion Laboratory on two projects related to mission data systems and space data system modeling. ArchStudio and its technologies have also been used in at least 10 other published projects. These projects and their use of ArchStudio technology are described briefly as well.

6.1 Software-Defined Radios

A software defined radio (SDR) [49] is a device that can communicate with other devices by means of radio waves. Unlike traditional radio devices, software-defined radios can be configured to communicate on many frequencies, with many different types of modulation by means of a reconfigurable combination of hardware and software elements. In an SDR, much of the signal processing is performed by general-purpose computing processors (GPPs), although some high-performance signal

processing is also done by reconfigurable processors—i.e., field-programmable gate arrays (FPGAs). Different software applications, known as waveforms, can be deployed on the radio to cause it to communicate in a particular frequency band, using a particular modulation, for a particular purpose. Because software-defined radios are so flexible, they can be used in far more contexts than traditional single-purpose radios. They can also be upgraded as needed to take advantage of new technologies or interoperate with new systems.

The Boeing Company has been contracted by the U.S. Department of Defense to construct a software-defined radio system. A joint research project with Boeing resulted in ArchStudio being used to model various aspects of this SDR's architecture. ArchStudio helped to provide novel views and capabilities not available to Boeing via other state-of-the-practice tools.

6.1.1 Software Defined Radio Background

The term "software-defined radio" applies to a large class of devices constructed in many different ways. Although these designs often share common elements (the use of FPGAs and GPPs, for example), there are still differences from system to system. An amateur SDR, for example, will be designed differently than a military SDR. The notes in this section refer to the class of software-defined radios that Boeing is contracted to construct.

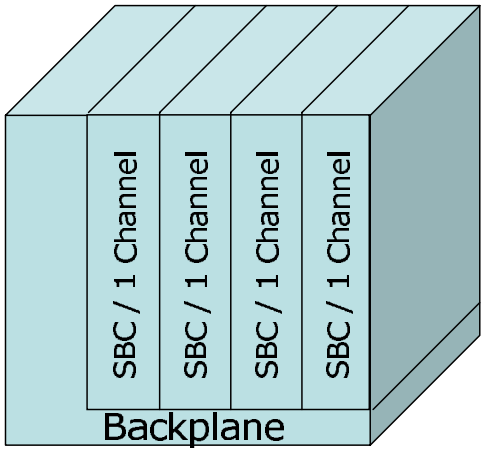


Figure 60. The hardware platform for a software-defined radio.

The basic hardware platform for a software-defined radio consists of a backplane containing one or more single-board computers. Each single-board computer is, in effect, a radio that can communicate over a single channel. Different configurations of backplanes and single-board computers are used for different applications. These SDRs are expected to be deployed in a wide variety of ways—from small radios carried by personnel to large radios installed on vehicles, aircraft, and ships. Depending on the application, a radio may have as few as two boards/channels or as many as eight. Figure 60 shows a four-board/four-channel configuration.

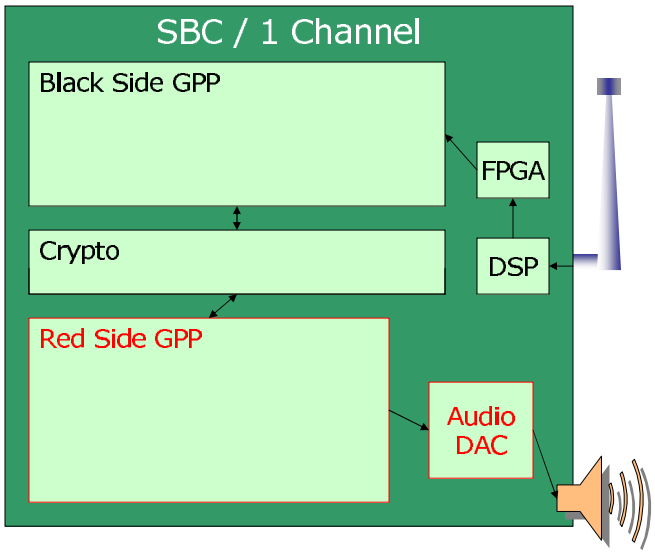


Figure 61. Hardware elements on an SDR's single-board computer.

Examining any one of these boards reveals interconnected hardware processing elements. Figure 61 shows one hypothetical configuration: an analog signal coming in through an antenna is converted to a digital stream by a DAC/DSP chip. This signal is sent to an FPGA for further high-performance signal processing. This signal is assumed to be encrypted, so the resulting digital stream of packets is sent to a "black-side" GPP, where processing is done on the encrypted stream. It passes through a separate cryptographic processor that decodes the stream and sends the decoded stream to a "red-side" GPP, which processes the unencrypted data. This processor may or may communicate further with an audio device that allows part of the signal to be output as sound to a speaker, performing much the same function as a personal computer's sound card.

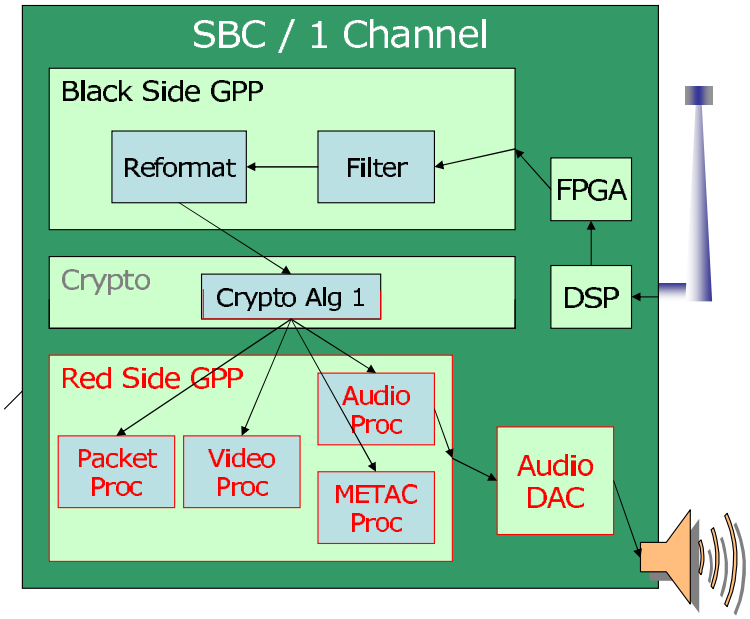


Figure 62. Software elements deployed on the hardware elements of an SDR.

Figure 62 shows one additional level of detail. On these general-purpose processing devices, various software components are deployed. Here, another hypothetical example is shown: basic filtering and reformatting are done on the black side, cryptographic algorithms decode the stream on the crypto processor, and various stream processing components act on the decrypted data on the red side. Managing the configurations and interconnections of these software components within the context of the distributed processing environment is a critical activity when constructing an SDR.

The construction of Boeing's software-defined radio is governed by a standard called the Software Communications Architecture (SCA) [114]. The SCA describes the architecture of an SDR primarily in terms of two technologies: POSIX [82] and CORBA [121]. SCA-compliant SDRs run POSIX (e.g., UNIX-like) operating systems, and thus have available all the usual POSIX services—file storage, processes and scheduling, and so on. SDR components, such as those shown in Figure 62, are

implemented as CORBA components. Each component exposes a set of provided services that can be called by other components. Some components are part of the SDR's "operating environment" (OE). These components serve as device drivers for different hardware devices on the radio (such as the audio device shown above), as well as wrappers for typical POSIX services such as file management. These components are instantiated during radio startup and generally persist throughout the radio's operation. The operating environment provides a layer of service to the application, or waveform, components that are deployed on the radio to configure it for a particular purpose. All components, whether on the same processor or different processors, communicate via CORBA, and CORBA's support for location transparency masks the actual location of each component.

The SCA specification goes into great detail about the types of components that can be used to construct an SDR and what their (CORBA IDL) interfaces should look like. However, it offers surprisingly little advice on how to actually compose these components into a working radio. Essentially, the SCA prescribes a catalog of reusable parts, but not the blueprints for assembling them. The assembly is up to the SDR developers themselves.

An effort called the SCA reference implementation (SCARI) project [1] implemented a simple demonstrative SCA-compliant software-defined radio that works not over a radio link, but over a network with two personal computers. As such, it provides an example of how a radio might be implemented using the SCA standard, but does not represent an adequate reference architecture for building a real radio. Instead, the SCARI radio implementation can be thought of as the "hello world" of software-

defined radios. All code and documents for this project are publicly available. This makes it an excellent testbed for developing SCA-related tools. In this section, all examples shown will be drawn from the SCA open-source radio, rather than Boeing's proprietary radio since that data is not public.

Developing an SCA-compliant radio is not a trivial endeavor. Working with Boeing, we were able to provide several novel architectural views that were not being provided by existing tools. These included:

1. An end-to-end structural view, showing components and connectors as actually deployed and connected on the radio;
2. Product-line views, showing potential alternative deployments of components on different hosts; and
3. A view that integrated architectural states with deployments, detailing the architectural changes that occur during radio operating environment startup.

6.1.2 End-to-End Structural Modeling

The first ArchStudio view created for the SDR project was an end-to-end structural model depicting all the radio's software components, running on their hosts, with interconnections (both intra- and inter-host) modeled as well. As noted above, the SCA does not provide this specification, nor does it provide a reference architecture. Appendix D to the SCA specification [114] does, however, provide an XML format for deployment descriptors. These deployment descriptors are configuration files that are part of the SDR implementation. They specify the components to be deployed on the radio, either as part of the operating environment or a particular waveform. They also define both provided and required interfaces on each component, and interconnections

among them. These artifacts can be seen as weak architecture descriptions for SDRs. At the time we first encountered them, these files were being written manually, in XML, by SDR engineers. Beyond syntactic validation against the SCA's DTDs, there was no way to assess the quality or correctness of these XML files.

Using the SCA specifications, a translator was constructed to convert SCA "Appendix D" deployment descriptors into xADL 2.0 documents. This translator used an open-source SCA toolkit from the SCARI project to parse the deployment descriptors, and the xADL data binding library to generate the xADL descriptions. The operation of the translator was theoretically straightforward, however an additional complication arose as it was applied to actual SDR deployment descriptors. These descriptors were found to be incomplete (links connecting to nonexistent interfaces, elements with duplicate identifiers, and so on). In this case, we were restricted to a "look but don't touch" mode of operating—we were allowed to examine the SDR descriptors, but not modify them. As such, the translator was enhanced to deal with several exceptional conditions we found in the real SDR deployment descriptors and report them when they occurred. The translator itself works in a manner similar to how the SDR itself operates. First, the operating environment deployment descriptors are parsed and translated into xADL. Then, a set of waveform deployment descriptors can also be parsed and mixed in to the xADL, just as waveform components are deployed onto SDRs.

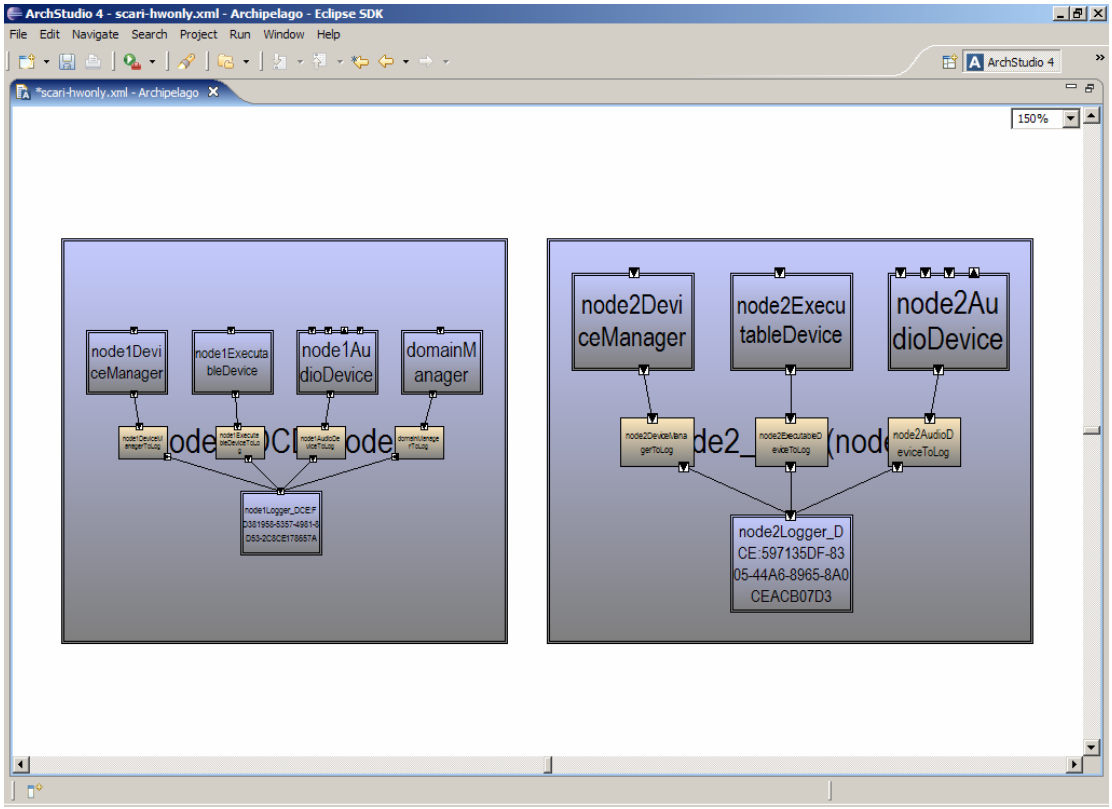


Figure 63. A screenshot of Archipelago showing the SCARI radio with no waveforms deployed.

Figure 63 shows a screenshot of Archipelago with a translated SCARI model loaded. This model shows the two-node SCARI radio with no waveforms deployed. The only components running on the two hosts are device drivers and basic services (e.g., logging).

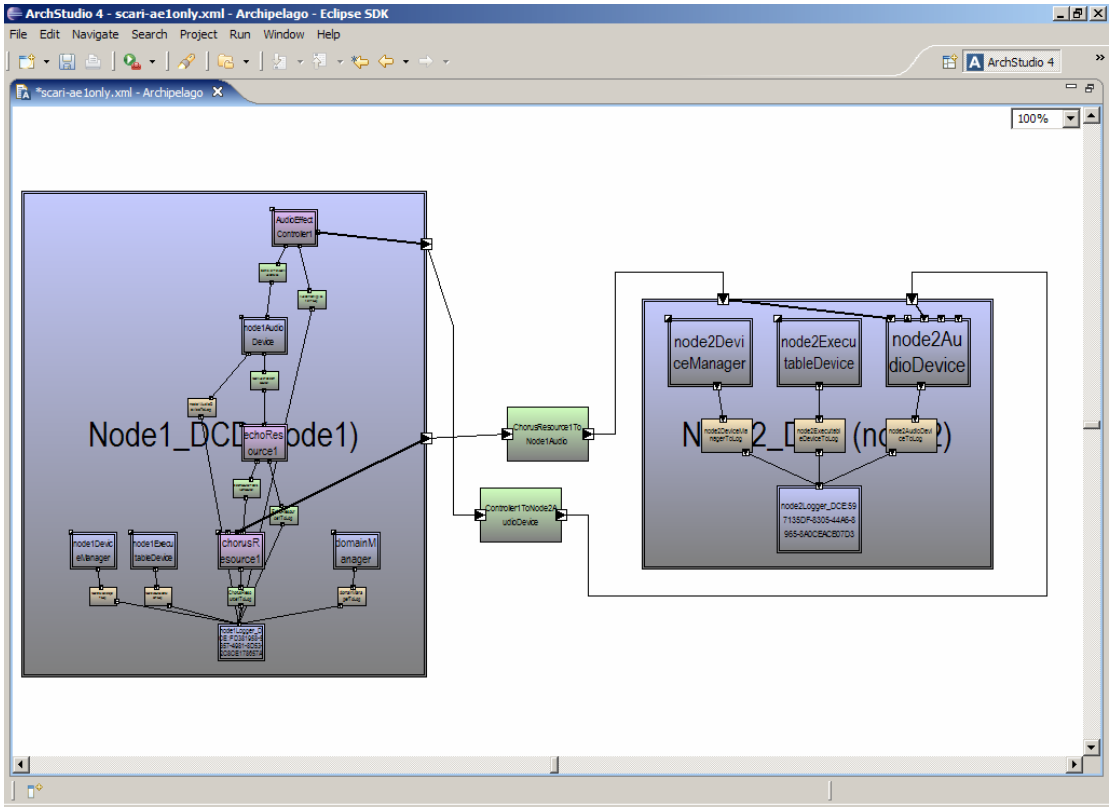


Figure 64. A screenshot of Archipelago showing the SCARI radio with one waveform deployed.

Figure 64 shows the same SCARI platform, except this time the model includes one deployed waveform. The purpose of this waveform is to receive sound on the left host, run it through effects processing (chorus and echo) and then transmit the audio to the right host for playback. Waveform elements are displayed in distinctive colors; the translator automatically adds coloring hints to waveform components and connectors to more easily visually distinguish them from operating environment elements.

Views like the above were generated for all Boeing waveforms, at least five in total. These were the first visualizations available of the deployed waveforms that were generated from actual development artifacts. The translation process and Archlight identified several hundred issues in the deployment descriptors. These were mostly errors of incompleteness: missing interfaces, unspecified identifiers, and so on. These

issues were provided to Boeing in a report. Additionally, the visualizations provided visceral insight into the structure of the waveforms. In one case, the waveform developers indicated that the visualizations were incorrect, because several components had been moved to other hosts or renamed. In this case, the designs for the waveforms had been changed, but the actual waveform deployment descriptors had not been updated to match. In another case, one waveform component wrapped a large amount of legacy code. This component had an abnormally large number of interfaces and may have represented a performance bottleneck.

6.1.3 Product-Line Views

In developing a software-defined radio, one choice that developers must make is how to assign components to processing elements. Some components are intrinsically bound to certain processing elements—because they are drivers for hardware connected directly to that processor, for example. Other components, particularly waveform components, can be deployed on almost any processing element. The underlying CORBA middleware provides these components with location transparency.

SCA deployment descriptors have no ability to specify variant deployments of this kind. It is also difficult for radio developers to visualize the impact of moving a component from one processor to another, especially in terms of connectivity to other components. Components that have many dependencies on components or devices available on a particular processor should probably be deployed on that processor as well.

The SCA-to-xADL translator was enhanced to allow users to specify optional deployments. By adding an entry to the translator's configuration file, an identified

component's deployment could be relaxed: instead of it being deployed on 'node 1,' for example, it could be deployed on 'node 1 or node 2.'

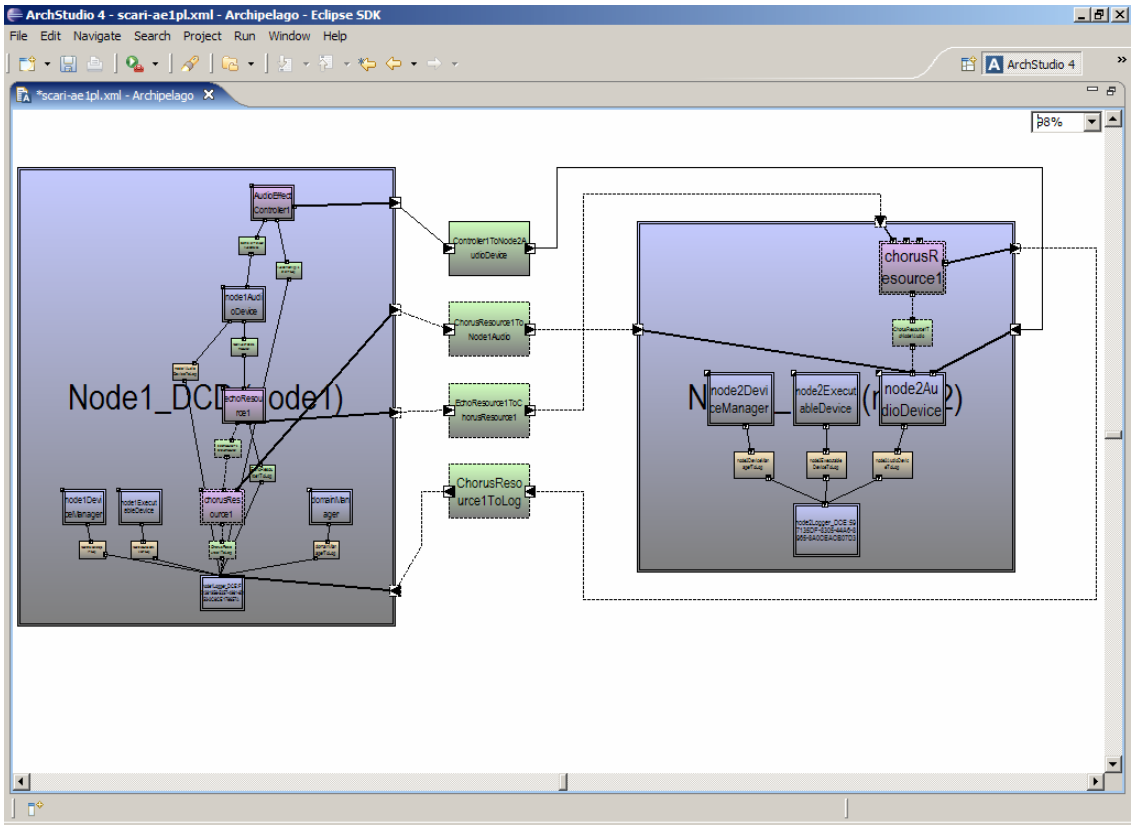


Figure 65. A screenshot of Archipelago showing the SCARI radio with alternative deployments.

Figure 65 demonstrates what the output of the translator looks like when alternative deployments are used. Here, the deployment of the 'Chorus Resource 1' component has been relaxed. It has been deployed on the left node, as usual, but it has also been deployed on the right node. In both places, the component and all links and connectors binding it to the rest of the architecture are deployed optionally. Each possible deployment is given a simple Boolean guard. The product-line selector can then be used to select down the set of alternatives to a product-line with fewer alternatives, or a single deployment. If the Chorus Resource Component in Figure 65 is

assigned to the left node using the selector, the result will be identical to the structure shown in Figure 64.

6.1.4 The Stateline Project

SDR architectures are, by their nature, not static. The architectures depicted in the end-to-end depictions are really snapshots of the SDR's architecture at certain times—for example, when all waveforms are completely loaded. However, many intermediate states exist. Obvious intermediate states occur due to software loading and waveform deployment: when the radio starts, no software is loaded. At some later time, the operating environment software is loaded and the radio is ready for waveform deployment. Then, one or more waveforms are deployed on the system, each resulting in a different architectural state. Waveforms can also be unloaded from an SDR.

Less obvious are the intermediate states that occur during these operations. For example, the operating environment software has specific a component loading order that is important to ensure proper startup—so that, for example, necessary dependencies are always satisfied. Although many architecture description languages provide some kind of viewpoint for capturing system behavior over time, few provide a connection between dynamic views of a system and traditionally static ones like structural views. In a software-defined radio, the radio structure is dynamic.

During part of the work with Boeing, a new capability was added to ArchStudio to address this situation. This capability, called *Stateline,* combined traditional statecharts with product-lines to capture how a system's structure changes over time. This need motivated the creation of the xADL STATES schema (see Section 4.1.7) as well as the development of a simple statechart editor for Archipelago. With Stateline,

the user uses the Archipelago statechart editor to define a statechart that expresses the important states and transitions for an architecture. Then, the user switches to the Archipelago structural editor, and through a Stateline plug-in to the structural editor, assigns architectural elements such as components, connectors, interfaces, and links to individual states. The assignment of elements to states occurs by making the element optional, and then assigning a product-line guard to the element. Conceptually, the guard is an ORed series of comparisons of a single variable with values representing each state, e.g., `(state == 'state1') || (state == 'state2')`. Thus, the structural view of the system becomes a product-line. Ordinarily, product-lines are used to capture different product variants or design alternatives. With Stateline, product-lines are used to capture variations in a single product at different points in time. Each product in the product-line, therefore, represents a snapshot of the product in a given state.

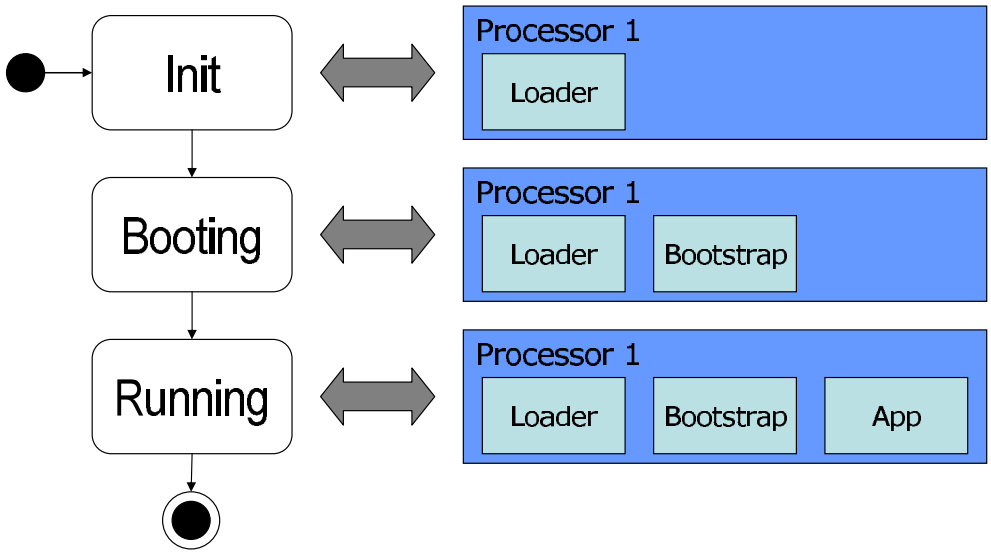


Figure 66. Stateline concept.

The Stateline concept is shown in Figure 66. Each state in the statechart is associated with a structural configuration, with each configuration being a product in a product-line that captures all states simultaneously. Boeing had produced a set of UML diagrams describing the boot process for their SDR's operating environment. One diagram was a component diagram showing all possible operating environment components deployed on the various radio processing elements, regardless of whether these were actually present simultaneously or not. Another set of diagrams were UML sequence diagrams depicting various activities occurring during the boot process of the radio.

These diagrams were used as the basis for an initial pass at using Stateline to model the radio boot sequence. The component diagram was modeled in xADL. Seemingly important states were extracted from the sequence diagrams, and a statechart was produced, also in xADL using the statechart editor. Each component in the component diagram was associated with the states in which it was believed to be loaded. This first-pass model was then presented to a Boeing engineer who had expert knowledge of the actual radio behavior. The first pass was about 60-70% correct. Two additional passes based on this engineer's comments and responses produced a sequence of structural diagrams, coupled with an associated statechart, that were agreed to depict the actual boot process of the radio. This experience suggested that the existing radio documentation did not make the boot sequence completely obvious, and that the additional documentation created using ArchStudio added value and understandability.

6.2 U.S. Air Force AWACS

During the early 2000s, the information systems on the U.S. Air Force AWACS aircraft [5] were undergoing a major upgrade known as the 'Block 40/45' upgrade. Prior to the upgrade, the software running on AWACS was primarily developed using 1970s-era technology and consisted of approximately 350,000 lines of JOVIAL code [141]. The Block 40/45 version of the software is supported by a large, distributed message-passing software architecture [113] based on a real-time CORBA ORB. A partnership between UC Irvine and Lockheed-Martin Systems Integration endeavored to create a formal architectural model of this software. During this partnership, we modeled the architecture of the AWACS aircraft's software systems in xADL 2.0 and used this model as the basis for an architecture-based simulation of AWACS. The model of AWACS was created based on available documentation and a proprietary simulator of AWACS component behavior provided to us by Lockheed.

The AWACS description consists of more than 10,000 lines (almost one megabyte) of xADL. The AWACS description describes 125 components and 206 connectors, distributed across 28 processors, along with component, connector, and interface types. The description was validated against the xADL 2.0 schemas using XSV and visualized with ArchStudio.

Along with the initial architecture description, we developed an architecture-based simulation of the communication among the components on the aircraft. The simulator consists of implementations of each component and connector type in the architecture in Java. Implementation mappings between types in the architecture description and these Java classes were added to the architecture description. A

bootstrapping program reads the architecture description into xArchADT and instantiates and links the components and connectors automatically. The AWACS architecture was primarily visualized in Visio for xADL, a predecessor to Archipelago built using Microsoft Visio. Messages sent among AWACS components were animated atop this depiction—the simulation could be 'stepped' through time by ticks of a real-time clock, and each tick depicted appropriate messages being sent among components on the graphical depiction.

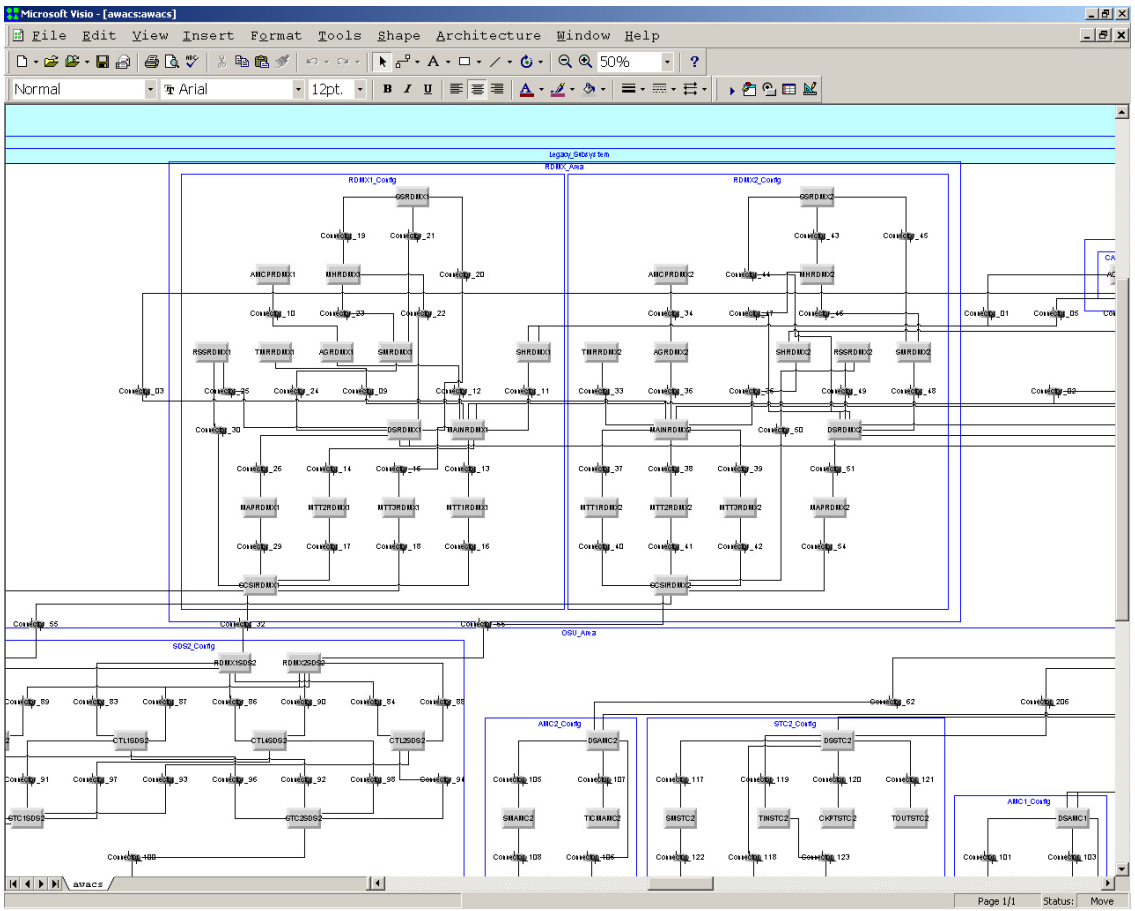


Figure 67. Screenshot of the AWACS architecture modeled in xADL.

A screenshot of the depiction of the AWACS architecture is shown in Figure 67. While no new xADL schemas were developed to model AWACS, the integration of

simulation capability was new. This capability was also used in the MTAT project, described in Section 6.5.4. The AWACS effort demonstrated the scalability of xADL and the ArchStudio toolset to deal with architectures that used large numbers of components and connectors.

6.3 JPL Mission Data Systems Project

NASA's Jet Propulsion Laboratory (JPL) develops software systems for space probes and ground systems. This domain induces unique modeling needs; specifically, these architectures are modeled as state-based systems. To accommodate these modeling constraints, JPL's Mission Data Systems group [143] and their research partners at the University of Southern California (USC) created extensions to xADL 2.0 [142]. While the specific contents of these schemas cannot be shared due to their proprietary and confidential nature, they are roughly characterized as follows:

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

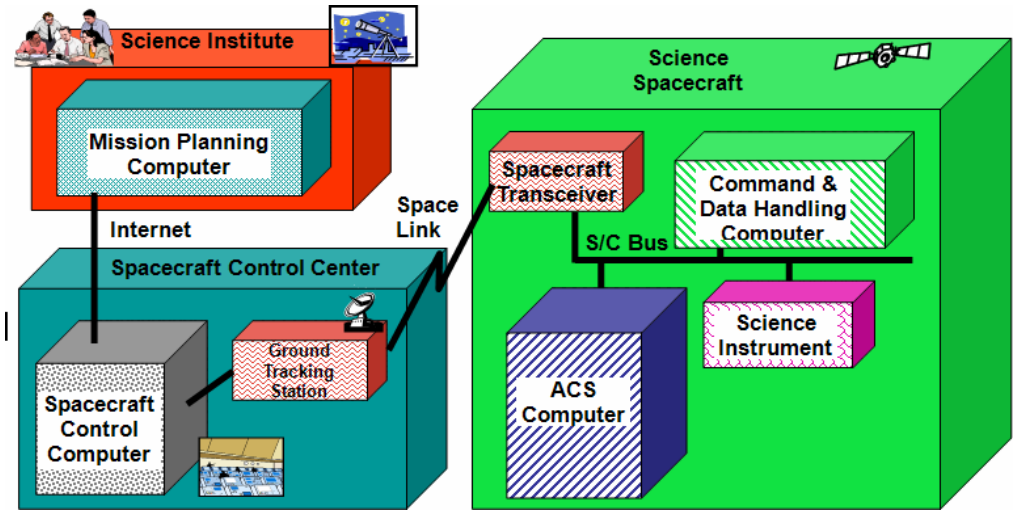
- **MDS Implementation Schema:** This schema links architectural artifacts to implementation-level counterparts as expressed in the JPL-proprietary MDS framework. It is used primarily for code generation.

JPL integrated their extended version of xADL 2.0 into their development process by creating translators to and from existing notations, such as UML and proprietary text-based notations. They also adopted our tools to visualize and manipulate architecture descriptions. Additionally, researchers at USC used the extended version of xADL 2.0 to model the architecture of the SCrover, a mobile robot based on JPL's MDS framework. They modified an existing architecture analysis tool, DRADEL [105], to use JPL-extended xADL descriptions rather than its own proprietary models, and used DRADEL to analyze the SCrover architecture. The DRADEL analysis found 21 errors in the SCrover architecture, 6 of which were not also detected by a peer review process.

This experience provides evidence that xADL and ArchStudio can indeed be adapted by users not associated with its development to handle project- and domain-specific needs. JPL was able to reuse the xADL 2.0 base schemas, creating relatively small new schemas to model domain-specific details of spacecraft software. JPL's use of Apigen and the associated data binding library to manipulate architecture descriptions further shows the value of ArchStudio's generative tools. Finally, their mapping of architecture descriptions to other representations shows the adaptability of our approach to other, non-pre-planned situations.

6.4 RASDS Space Data Systems Modeling Project

Space and ground systems architectures are characterized by complex interactions governing data exchange and data flows. The Consultative Committee for Space Data Systems (CCSDS) is an organization made up of participants from major space agencies around the world interested in sharing information and developing standards for the construction of space data systems. The CCSDS found that existing modeling notations such as UML and SysML were insufficient for modeling space data systems, and endeavored to develop a new multi-view notation optimized for modeling them. This notation is defined in a standard they have developed called the reference architecture for space data systems (RASDS) [38], based loosely on RM-ODP. The RASDS notation, however, is not accompanied by a tool-set or a rigorous syntax. As a pilot project, we experimented with using xADL and ArchStudio to capture RASDS diagrams.



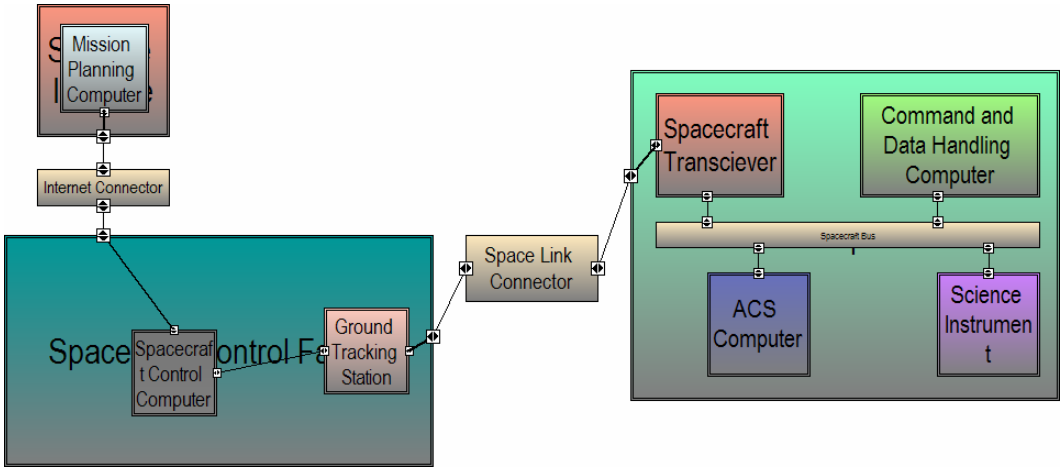
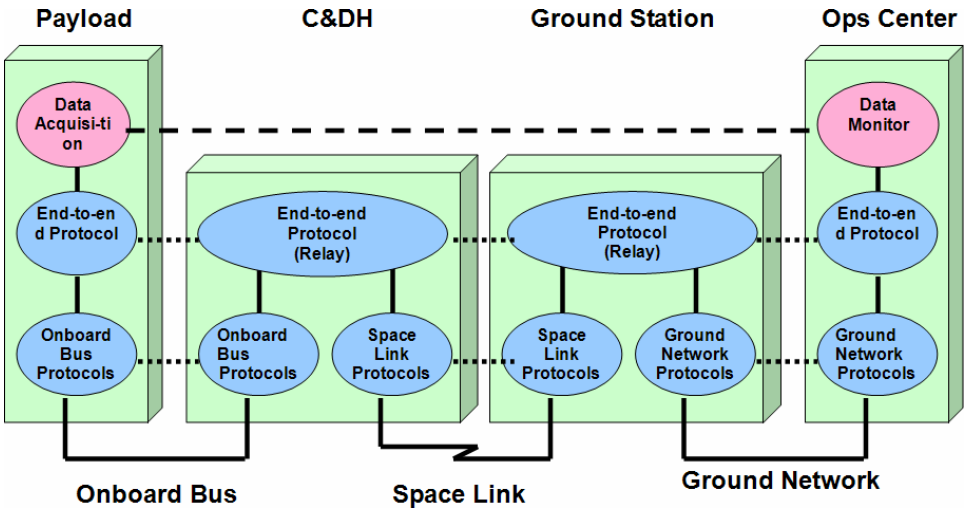


Figure 68. RASDS connectivity view from specification (top) and modeled in ArchStudio (bottom).

Figure 68 depicts one result from this pilot project. The top half of the figure shows a diagram extracted from the RASDS specification. As should be obvious, this diagram was created in a diagramming tools such as Microsoft PowerPoint rather than an architecture modeling environment. The bottom half of the diagram shows the same RASDS model, captured in xADL and visualized in ArchStudio. Unlike the top diagram, the xADL mdoel obeys specific rules and can be automatically checked for consistency using Archlight.



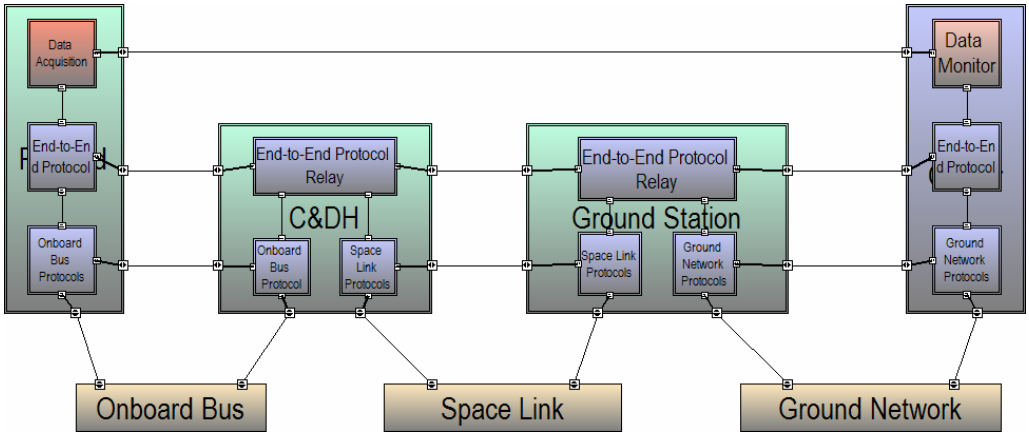


Figure 69. RASDS communications view from specification (top) and modeled in ArchStudio (bottom).

Figure 69 shows another instance of the same phenomenon, this time using the RASDS communications view. The communications view is modeled on a typical "network stack" diagram in which various layers of abstraction communicate over horizontal logical links via lower-layer services that ultimately resolve to physical-layer interconnections at the lowest level. Unlike the PowerPoint model of this view, the xADL view could be turned into a product-line, where one variant includes logical links and the other shows only the direct interconnection pathways that go all the way through the physical layer.

This experience reflects yet another instance of one of ArchStudio's core goals: to be useful as a modeling environment for domain-specific notations that would otherwise have no tool support. The CCSDS's creation of a new notation to fit their own needs validates the assumption that existing state-of-the-practice notations are not sufficient to fill all architecture modeling requirements.

6.5 Additional Validation

In addition to the projects described above, many individuals and organizations have adopted some or all of the xADL/ArchStudio technology and the approach described in this dissertation to facilitate their research or modeling needs. These projects and their use of ArchStudio/xADL are summarized briefly here, with references to the published work where appropriate.

6.5.1 Secure xADL

Jie Ren, in his dissertation work [135], adopted the extant versions of xADL 2.0 and ArchStudio as a basis for his work. He extended xADL 2.0 by adding new modules for capturing security properties of an architecture. He extended Archipelago by adding new plug-ins for visualizing and editing these security properties (including integrating a third-party property editor for the XACML access control markup language [120] he did not write himself). He extended Archlight to show the results of security violations found by his tool. ArchStudio provided Ren with a complete platform for investigating how to model, visualize, and analyze access control restrictions within a software architecture.

6.5.2 EASEL

Scott Hendrickson's EASEL project [73, 74] is an effort to rethink how architectures are modeled and manipulated. Ordinary architecture descriptions, including xADL descriptions, depict the architecture of a system as a set of interconnected views. Through product-line modeling techniques, these views can be diversified to express the architecture of multiple, related projects. In EASEL, views in

an architecture description are not singular entities, but in fact compositions of change sets. Each change set describes a set of modifications to an architecture. Starting with the null set (i.e., no elements at all) change sets may add, modify, or remove elements. In the EASEL tool, users can activate or deactivate change sets in real time and see the results immediately reflected in the visualization.

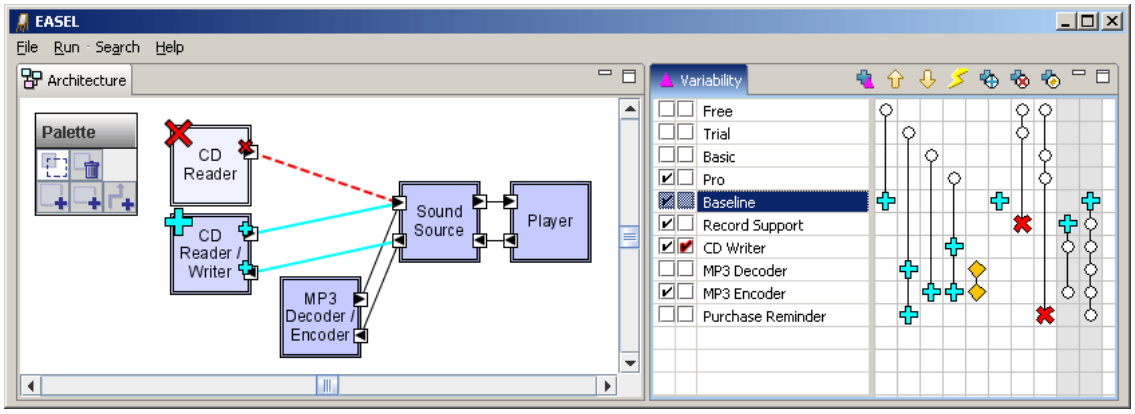


Figure 70. EASEL screenshot.

A screenshot of the EASEL graphical editor is shown in Figure 70. The initial version of EASEL was not integrated into ArchStudio; however, it used an extended version of xADL along with generated data bindings for its modeling notation and used the BNA framework as a basis for its graphical editor. A new version of this change-set based approach is being integrated into ArchStudio now, replacing EASEL's proprietary editor with an extended version of Archipelago.

6.5.3 Architectural Differencing and Merging

Architecture descriptions evolve and change over time. Product-line architectures are one way to manage variants over time, as shown in the above discussion of the Stateline project. However, using product lines in this way is only

effective when all variations are encoded into the product-line. This technique is not effective if a single architecture description is evolved by making modifications to it.

Architectural differencing and merging [31] is an effort to better manage this kind of architectural model evolution. This approach involves two tools. The first, an architectural differencing engine, takes two architecture descriptions as input and outputs a document called a ‘diff’ representing the differences between the two descriptions. The second, an architectural merging engine, takes as input a baseline architecture description and a diff, and outputs a new architecture description that consists of the baseline plus the changes captured in the diff. Differencing and merging can be expressed as functions over architecture descriptions A and B and diff D as follows:

$$Diff(A, B) = D$$

$$Merge(A, D) = B$$

Chen et. al. implemented a differencing and merging approach in ArchStudio. They extended xADL, creating a ‘diff’ schema that can be used to encode the differences between two architecture descriptions. They then implemented four new components in ArchStudio: a differencing engine ArchDiff, a user interface for the differencing engine, a merging engine ArchMerge, and a user interface for the merging engine.

A later project co-opted these differencing and merging engines to support model-driven dynamic software architectures [128]. In this project, another component was added to ArchStudio, the Architecture Evolution Manager (AEM). The AEM could bootstrap a xADL 2.0 architecture using the same approach used to bootstrap

ArchStudio itself. After the application is running, the AEM watches for merges occurring in xArchADT. When a merge is performed, AEM examines the changes made to the baseline architectural model and makes the corresponding changes on the running system. Thus, ArchDiff and ArchMerge are not only useful for understanding changes from one architecture to another and propagating changes from one architecture to another, but they are also useful as the drivers for propagating the same changes to a running system.

6.5.4 MTAT

The MTAT project [71, 72] is an attempt to increase the understandability and maintainability of event-based software systems. Event-based systems are those in which independent components, often treated as black boxes, communicate exclusively by the use of events—that is, discrete messages. In a typical event-based application, a component will receive one or more events from other components. As a result, they will perform some internal computation and emit one or more other events in response. For example, consider a data structure component. This component receives an information request event and responds by emitting an event containing the requested information. Additionally, some components may emit events in response to external stimuli (user input, network traffic, and so on), and some components may absorb events but not emit any in response.

MTAT is a tool-based approach for tracing and understanding the flows of events through event-based systems, specifically event-based systems implemented in the c2.fw framework whose architectures are specified in xADL 2.0. It consists of three main parts. The first part is an instrumenting tool that adds tracing connectors to a

xADL specification of an event-based system. This tool breaks each architectural link in the document into two separate links, bridged by a new ‘trace connector.’ This trace connector forwards events in both directions to maintain ordinary application behavior. However, it also makes a copy of each event internally and forwards this copy to a database component for storage. The second part is a xADL extension schema and mechanism for specifying causality heuristics on a component level. Specifically, using this schema, a user can write rules for each component explaining its behavior in terms of the messages it receives and the messages it emits as a result.

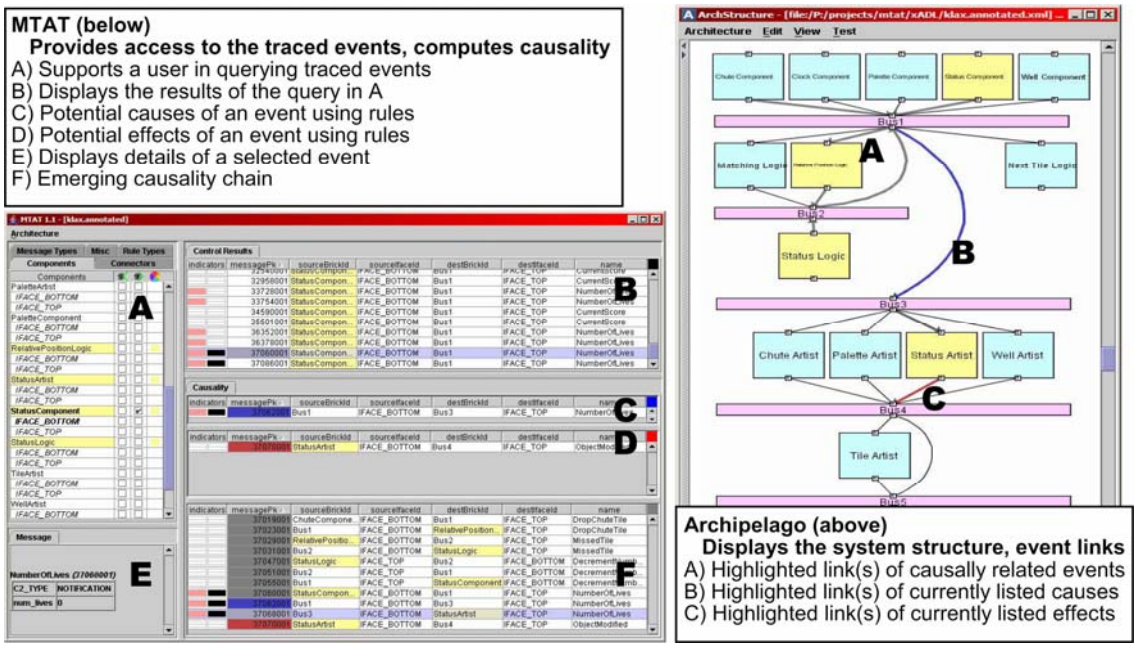


Figure 71. MTAT concept diagram and screenshots (reproduced from Hendrickson et. al.)

The third part of MTAT is the message trace analyzer; screenshots of this tool are shown in Figure 72 (reproduced from [72]). This tool takes as input the xADL description of the running system’s architecture annotated with message rules along with the database of events captured from a run of an instrumented version of the target application. It then allows the user to explore the messages captured during the

instrumented system's run. Users can simply examine the raw event stream, or they can leverage the causality rules to attempt to understand patterns of control and data flow through the architecture. Using the specified message rules, the message trace analyzer will attempt to predict the causal relationships between a message or messages received by a component and later messages emitted by that component. Through causal chains, users can trace control and data flow through the system. The message trace analyzer is also integrated with Archipelago. During the exploration, the architectural structure is decorated with highlights and animations graphically indicating where messages selected in the tool occurred and what their effects and causes are.

MTAT extends both xADL and Archipelago with new schemas and plug-ins. Although it is an application independent of ArchStudio, it integrates with ArchStudio using some of ArchStudio's tool integration mechanisms such as xArchADT and Archipelago's provided interfaces. MTAT's developers reused xADL's features for structural description and Archipelago's ability to visualize these structures. MTAT leveraged xADL and ArchStudio's extensibility mechanisms to provide a novel, dynamic view of event-based architectures without reinventing the wheel.

6.5.5 KBAAM

Georgas and Taylor [62] have developed an approach to knowledge-based architectural adaptation. This approach is different from the architectural differencing and merging approach described in Section 6.5.3. This approach is knowledge-driven, using an expert system engine to determine when and how to adapt a running system. The KBAAM approach is policy-based: an architecture is accompanied by one or more adaptation policies, each of which contains a set of *observations* and *responses*. The

framework watches a running system, and when a set of observations is matched, the responses—architectural changes—are executed. In the prototype implementation of the KBAAM framework, xADL extensions are used to capture the adaptation policies, observations, and responses. Additionally, the framework incorporates the architecture evolution manager (AEM) tool from ArchStudio to assist in performing the adaptations.

The KBAAM approach is another demonstration of xADL's adaptability to a new concern, namely knowledge-based adaptation. Additionally, KBAAM's ability to extract a part of ArchStudio—the AEM—for use in a different context is indicative of the usefulness of ArchStudio's loosely-coupled tool integration strategy.

6.5.6 XASTRO

XASTRO [101] is an effort by the European Space Agency to define a framework to improve the exchange of system engineering information between customers, contractors and suppliers within space programs. A primary contribution of XASTRO is an XML-based format for encoding system engineering information and designs. Although the XASTRO project did not use xADL directly, key elements of xADL's modular design were incorporated into the XASTRO format. The XASTRO Framework And Model Prototype specification [52] describes XASTRO's use of xADL thusly:

> "Among the described ADLs, xADL 2.0 was chosen as a basis for the XASTRO Framework for the following reasons:
>
> - Based on XML Schema
> - Support for major ADL concepts
> - Extensible design

- Support for the following XASTRO concepts:
    - Hierarchical decomposition
    - Re-usability (based on types)
    - Groups
    - External references (based on XML links)”

Section 3.2 of the XASTRO specification referenced above also describes how xADL concepts and elements are reflected in the design of XASTRO. That a major space agency selected xADL unsolicited as the basis for one of its data formats increases confidence that the design and modularity of xADL are novel and valuable contributions.

6.5.7 xADL with Statechart Behavioral Specifications

Naslavsky et. al. [116] became interested in understanding and analyzing the behavior of software architectures using statecharts as an underlying behavioral model. This research group had previously developed an analysis tool, Argus-I [173], which was able to analyze the behavior of C2-style architectures using statecharts, and wanted to generalize this work to support non-C2 architectures. They extended xADL to capture statecharts, developing a schema that was the direct predecessor to the current xADL STATES schema. They then used this extended version of xADL as a basis for refactoring Argus-I so that Argus-I could use xADL, rather than XMI, as its target format.

This experience is yet another indicator that xADL can be extended to support new modeling needs—in this case, a particular research goal. The reuse of the

externally developed schema in the Stateline project indicates that schema reusability can save time and effort.

6.5.8 Software Architecture Evolution through Dynamic AOP

Falcarin and Alonso [53] became interested in dynamic reconfiguration of distributed systems on the fly. They constructed a framework called JADDA that uses xADL 2.0 architecture descriptions as a mechanism for describing the intended architectural elements running on each host in the system. xADL documents are deployed to hosts to indicate what configuration they should adopt; runtime agents on each host make the appropriate changes to make the running software on the host conform to the xADL specifications. Several middleware technologies are supported for this purpose. This team extended xADL, particularly xADL connectors, to capture needed properties of different middleware technologies: CORBA, SOAP [24], and so on.

Here again is an example of xADL being used for its intended purpose: providing a general set of concepts for modeling different architectural concerns, and then allowing these concepts to be extended for domain- or project-specific needs. In this case, the extensions added data so that the xADL architectures could be used as runtime configuration files in a distributed system.

6.5.9 Enhancing the Role of Interfaces in Software Architecture Description Languages

Galvin et. al. [58] extended xADL to support a more detailed specification of interface types. The goal of this research was to specify component interfaces with

constraints that were checkable at runtime, and then use those specifications to generate actual runtime constraint checkers for an implemented system. This was done using a two-level approach. First, abstract, language-independent interface specifications are created and augmented with constraints (preconditions, postconditions, etc.). Second, language bindings define how the language-independent interface definitions can be transformed into language-specific definitions. This team created custom generator tools that could use the specifications to generate programming-language specific interface code, as well as proxies that could be used to verify the pre- and post-conditions at runtime.

This is another example of xADL being used as the modeling foundation for extensions that were created to support project-specific needs. Here also is an example of where xADL's incremental adoptability (i.e., the ability to use xADL productively without necessarily having to use all the support tools in ArchStudio) has been valuable.

6.5.10 Creating a Product-Line of Meshing Tools

Bastarrica et. al. [14] are concerned with the design of pieces of software called *meshing tools*. These tools create and manipulate meshes—three-dimensional models composed of simpler elements such as triangles, tetrahedra, hexahedra, or a combination of different shapes. This team identified that, from a software engineering perspective, meshing tools were being developed in an ad-hoc manner, without reusing design knowledge or components from one to the next. To remedy this, they propose the definition of a product-line architecture for meshing tools. Here, the product-line forms a sort of reference architecture for meshing tools. Unlike some reference architectures which are merely exemplars, a product-line defines the points of variation

explicitly. They specified such a product-line in xADL and visualized it using Archipelago.

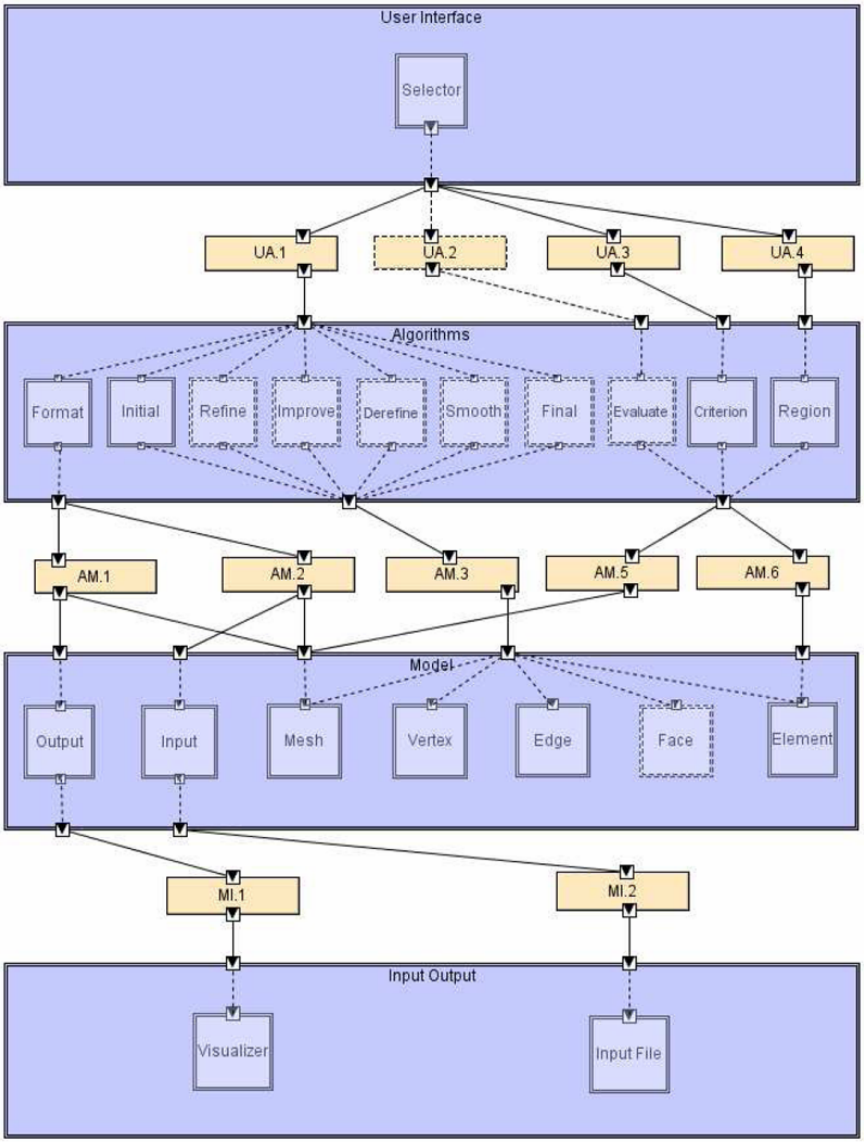


Figure 72. Meshing tools product-line rendered in Archipelago (reproduced from Bastarrica et. al.)

Figure 72 shows the actual reference architecture developed by this team, rendered in Archipelago. The diagram is extracted from [14].

In this project, the research team did not create extensions to xADL or ArchStudio—none were necessary in this case. This experience indicates that xADL

and ArchStudio's existing capabilities, especially those for specifying product-lines, are applicable in yet another domain.

6.5.11 Composing Architectural Cross-Cutting Features

Boucké et. al. [23] add modeling constructs to xADL to support cross-cutting structural features. They do so by adding a new top-level element to xADL, 'archComposition,' which contains three new elements. *Unifications* are relationships between elements in different structural views that indicate that two elements refer to the same element and therefore must be consistent with one another. *Mappings* map elements or groups of elements from one architectural structure onto a single element in a target structure. Correspondences are established via interfaces; the target element's interfaces must be the composition of the interfaces on the mapped elements. *Substructures* are very similar to xADL sub-architectures, except they can be composed through a different process; here, a group of elements is used to represent the internals of another element, with outer interfaces mapped to inner interfaces.

This research shows again how xADL can be extended for project-specific needs, as well as to serve as a basis for architectural research into different modeling mechanisms. The fact that this research and EASEL approach similar problems in completely different ways speak to xADL's ability to deal with divergent extension.

6.6 ArchStudio Itself

The final evaluation of ArchStudio's technology is found in the implementation of ArchStudio itself. As described in detail in Chapter 4, ArchStudio uses its own technology for its implementation. Its architecture is specified in xADL 2.0, visualized

in Archipelago, and analyzed in Archlight. The xADL 2.0 description is deployed as part of ArchStudio's implementation, and implementation mappings within the description are used to bootstrap ArchStudio whenever the environment is started.

ArchStudio and the various extensions that have been created to support modeling, visualizing, and analyzing different concerns provide data that can be used to evaluate the fourth and final hypothesis of this research. Recall that this hypothesis is:

H4. The size of the support modules will be commensurate with a level of development effort suitable to a single developer, generally less than 5,000 lines of code per concern.

Examining a selection of the features described throughout this dissertation, one can make a size estimate of each concern by looking at the number of lines of code developed for that concern's support in ArchStudio (xADL schemas, Archipelago plug-ins, and Archlight tests).

Concern	Artifact	Code Size
Structural Modeling	xADL Schema	~160 LOC
	Archipelago Plug-ins	~3500 LOC
	Archlight Tests	~520 LOC
Architectural Types	xADL Schema	~120 LOC
	Archipelago Plug-ins	~3300 LOC
	Archlight Tests	~1200 LOC
	Type Wrangler	~1200 LOC
Options	xADL Schema	~180 LOC
	Archipelago Plug-ins	~320 LOC

Variants	xADL Schema	~120 LOC
	Archipelago Plug-ins	~600 LOC
Boolean Guards	xADL Schema	~300 LOC
	Archipelago Plug-ins	~250 LOC
	Infix Parser	~4000 LOC (auto-generated)
	Evaluator	~1800 LOC
Product-Line Selection	Selector Engine	~1100 LOC
	Selector UI	~700 LOC
States / Stateline	xADL Schema	~170 LOC
	Archipelago Plug-ins (statechart editor)	~2000 LOC
	Archipelago Plug-ins (Stateline features)	~500 LOC
Architectural Differencing and Merging	xADL Schema	~120 LOC
	Differencing Engine	~750 LOC
	Differencing UI	~350 LOC
	Merging Engine	~650 LOC
	Merging UI	~350 LOC
Secure xADL	xADL Schema	~500 LOC
	Archipelago Plug-ins	~1300 LOC

MTAT	(Combined)	~16,000 LOC
EASEL	(Combined)	~20,000 LOC
KBAAM	(Combined)	~6000 LOC

Table 2. Sizes of various ArchStudio concerns.

The code sizes of various ArchStudio concerns are shown in Table 2. This data indicates that the initial premise of the hypothesis, that support modules for a given concern can be developed by a single developer, is plausible. The size of support modules for a given concern can be measured in hundreds or thousands of lines of code. The second premise of the hypothesis, that support modules are generally less than 5000 lines of code, is less well-supported by this data. Basic support—xADL schemas, Archipelago plug-ins, and Archlight tests, generally fit within this limit, even for more complex concerns like structure and types. However, the 5000-line budget is exceeded when additional, specialized tools are incorporated into the equation—this is seen in, for example, the Type Wrangler and Selector components, as well as tools that incorporate significant new functionality such as MTAT, EASEL, and KBAAM.

One trend that does emerge from this data is that the code size of concerns generally increases along with the amount of additional functionality added to the environment. Complex concerns that are the foundation for many others, such as structure and types, tend to be larger. Comprehensive architectural approaches such as MTAT and EASEL also tend to be larger. Concerns that only decorate existing concepts or add small new capabilities to the system, such as options and variants, are proportionally smaller. This is consistent with the ArchStudio goal that the effort to integrate a new concern into the environment should increase proportionally with the complexity of that concern.

7 Conclusions and Future Work

This dissertation has set out to address the problem of stakeholder-driven, multi-view software architecture modeling. Put another way, how can architecture modeling be supported in a world where stakeholders, not just approach developers, play a principal role in defining the views and concerns to be modeled? This problem motivated the postulation of four hypotheses.

The first hypothesis, **H1**, asserted that an environment for stakeholder-driven multi-view software architecture modeling could be constructed that addressed modeling, visualization, and consistency checking. This existence hypothesis is validated through the development and evolution of xADL and the ArchStudio architecture-based software development environment, along with its major constituent subsystems Archipelago and Archlight.

The second hypothesis, **H2**, posits that within such an environment, architectural concerns can be supported by interdependent, reusable modules corresponding to concerns that can be composed into modeling support for a particular project or need. This hypothesis is validated by the development of many different support modules for both common and uncommon architectural concerns. Common concerns, such as structural modeling, type modeling, product-line creation and management, and implementation mappings are part of the shipping version of ArchStudio. However, many additional domain- and project-specific concerns are supported by modules developed by others. Chapter 6 describes more than 10 of these projects and the extensions that have been developed.

The third hypothesis, **H3**, asserts that the capabilities exemplified by the constructed environment can be enabled by following a particular set of design guidelines. Chapter 5 catalogs these design principles in detail. For each design principle, rationale is provided explaining what qualities or capabilities that principle provides to ArchStudio. Furthermore, each design principle is accompanied by information detailing exactly how ArchStudio embodies that design principle. These principles are further validated through the lens of experience: ArchStudio has evolved through four major versions (two of which were constructed specifically as stakeholder-driven environments) and many of these design principles emerged only after significant refinement in the environment.

The fourth hypothesis, **H4**, asserts that the size of the support modules will be commensurate with a level of development effort suitable to a single developer, generally less than 5,000 lines of code per concern. The modules developed as part of ArchStudio and externally partially validate and partially refute this hypothesis. Concerns can be feasibly implemented by a single developer. In general, if the scope of 'support' is considered to be limited to xADL schemas, Archipelago plug-ins, and Archlight tests, then the 5000-line-of-code limit holds. However, if 'support' includes custom tools such as task-specific editors or tools that bridge architecture with other lifecycle activities such as MTAT or KBAAM, the custom tools generally exceed the 5000-line limit.

The contributions of this work include:

- A modularly-extensible architecture description language (xADL) and support environment (ArchStudio) for stakeholder-driven, multi-view

software architecture modeling. The environment addresses and prescribes well-defined extensibility-mechanisms for capturing, visualizing, analyzing, and applying architectural design decisions;

- A demonstration that support for modeling architectural concerns can indeed be modularized and recomposed into solutions. A spectrum of concerns has been modularized in this way by myself and others, including structural modeling, types, product-line modeling, architectural states, security, behavior of event-based systems, change-set-based composition of structures, and so on;
- A set of design principles extracted from many years of development experience with the environment that can be applied in the construction of similar stakeholder-driven architecture modeling environments; and
- Evidence and experience indicating that, within this environment, the amount of effort required to develop extension modules scales roughly with the conceptual complexity of those modules and how different they are from existing supported concerns.

7.1 Future Work

This research provides a substantial infrastructure and platform for further investigation of architecture modeling, in many different respects. Primary future directions include the investigation of new modeling concerns. In fact, some of the early motivation for the development of ArchStudio was to provide an infrastructure for software architecture researchers interested in investigating new concerns or new ways of modeling existing concerns. In this regard, the scope and applicability of the

environment can be expanded both in terms of *depth* and *breadth*. In terms of depth, additional concerns that extend and augment the existing capabilities of the environment can be developed. Projects like Secure xADL have done this for concerns such as security and access control; many other concerns can be addressed in the same way. In terms of breadth, support for that goes beyond technical or structural concerns should be developed. Existing xADL schemas focus primarily on concerns that are of interest to stakeholders such as system architects and developers, but many more stakeholders are involved in the development of large, software-intensive systems: managers, customers, domain experts, end users, and so on. These stakeholders are often concerned about non-technical matters: schedules, costs, how the software system will fit within a larger organizational context or process, and so on. This broader array of concerns could inspire a cornucopia of future research projects.

Another area of potential interest is the use of the infrastructure to provide better interconnections between architecture and other lifecycle activities. Currently, some of this has been demonstrated: xADL's implementation mapping schemas and projects such as MTAT tie architectures to system implementations, for example. However, many additional artifacts are produced during the course of system development: requirements, detailed designs, architectural design decisions captured in natural language or alternative notations, test plans and results, deployment specifications, and so on. Given the diversity and heterogeneity of these artifacts, an intriguing project might be to incorporate an open-hypermedia system like Chimera [10] into the environment to create and manage interconnections among them. In addition to simply documenting and maintaining interconnections, some artifacts might be partially or

wholly generated based on architectural specifications. The OMG's Model-Driven Architecture (MDA) initiative [115] is a recent exemplar of this trend. In contrast to the MDA-style approach of generating complete implementations from models, more value might be attained by generating partial implementations or implementation skeletons and providing round-trip capabilities such that changes to either the model or the generated artifacts are reflected in the other to minimize architectural erosion and drift.

An interesting direction to pursue would be to reconstruct or enhance support for earlier, independently developed architecture modeling approaches. For example, the ideas and algorithms behind Rapide and Wright remain valid, but tool support for these ADLs has stagnated and many of their tools will no longer run on modern platforms. Additionally, notations like AADL and Koala provide unique modeling and analysis capabilities, but lack tooling infrastructure to allow users to maximize their potential. It would be interesting to compare the size, development effort, and capabilities of an ArchStudio-based tool-set for these languages with the native tools provided by their creators.

Further refinements of the infrastructure itself are also possible and may yield valuable results. Currently, extending the environment requires a degree of technical knowledge—code must be written, particularly to extend Archipelago. As such, non-technical stakeholders such as managers desiring new or custom extensions must work with a technically proficient individual as a proxy. The benefit to ArchStudio's approach thus far is that extensions tend to have great flexibility, and the resulting integrations of extensions are seamless and do not provide overly homogenous user experiences. However, it may be possible to reduce the complexity involved in

extending the environment, possibly to the point where non-technical stakeholders can participate directly in their creation. The SoftArch environment [67] and related projects have invested effort in this area and reported good initial results. Another area of interest might be further refactoring of the environment to extract all related extensions (to xADL, Archipelago, Archlight, and so on) into physically separate packages that are composed entirely at late-binding time. This would make it easier for users to independently develop and contribute their own environment extensions with less explicit coordination.

Adding further analysis capabilities by incorporating additional off-the-shelf model checkers such as Alloy and Spin has the potential to add a tremendous amount of value to architectural models. Since these model checkers are often domain-independent and not specialized to check the properties of software architecture, initial investigations must be done to determine how the types of analyses provided by the model checkers can be applied in the context of real-world architecture modeling problems.

Other research directions lie even further afield. Archipelago's construction and separation of behavior from depiction makes it relatively easy to swap out one set of user-interface elements for another. This might facilitate usability studies concerning the user interfaces of architecture modeling environments, or design environments in general. The design principles and portions of the environment's implementation may be useful in constructing design environments in alternative domains unrelated to software or architectural modeling. The environment might also be extended to support physically distributed developers in a collaborative setting, or to support esoteric user input and output devices such as smartboards, footswitches, PDAs, and so on.

8 References

[1] *SCA Reference Implementation Project*. <http://www.crc.ca/scari/>.

[2] *Argo/UML*. <http://argouml.tigris.org/>.

[3] Abdurazik, A. *Suitability of the UML as an Architecture Description Language with Applications to Testing.* George Mason University, Technical Report ISE-TR-00-01, p. 24, February 2002, 2000. <http://www.isse.gmu.edu/techrep/2000/00_01_abdurazik.pdf>.

[4] Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[5] Air Combat Command Public Affairs Office. *Fact Sheet: E-3 Sentry (AWACS).* U.S. Air Force, Report, July, 2000. <http://www.af.mil/news/factsheets/E_3_Sentry__AWACS_.html>.

[6] Allen, R. *A Formal Approach to Software Architecture.* Ph.D. Thesis. Carnegie Mellon University, p. 248, 1997. <http://reports-archive.adm.cs.cmu.edu/anon/1997/CMU-CS-97-144.pdf>.

[7] Allen, R. and Garlan, D. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology (TOSEM).* 6(3), p. 213-249, July, 1997.

[8] Altheim, M., Boumphrey, F., Dooley, S., McCarron, S., Schnitzenbaumer, S., and Wugofski, T. *Modularization of XHTML.* World Wide Web Consortium, W3C Recommendation Report, April 10, 2001. <http://www.w3.org/TR/xhtml-modularization/>.

[9] Altova GmbH. *XML Spy Website*. <http://www.xmlspy.com/>, Altova GmbH, Website, 2003.

[10] Anderson, K.M., Taylor, R.N., and Whitehead, E.J. Chimera: Hypermedia for Heterogeneous Software Development Environments. *ACM Transactions on Office Information Systems (TOIS).* 18(3), p. 211-245, July, 2000.

[11] Apache XML Project. *Xerces Java Parser*. <http://xml.apache.org/xerces-j/>, Website, 2003.

[12] AT&T Corporation. *UNIX System V C++ Language System Product Reference Manual*. Release 2.0 ed. AT&T Corporation, 1989.

[13] Bannon, L. From Human Factors to Human Actors: The Role of Psychology and Human-Computer Interaction Studies in System Design. In *Design at Work* Greenbaum, J. and Kyng, M. eds. p. 25-44, Lawrence Erlbaum Associates: Hilldale, New Jersey, 1991.

[14] Bastarrica, M., Hitschfeld-Kahler, N., and Rossel, P.O. Product Line Architecture for a Family of Meshing Tools. In *Proceedings of the 9th International Conference on Software Reuse (ICSR 2006).* LNCS 4039, p. 403-406, Springer. Turin, Italy, June, 2006. <http://www.dcc.uchile.cl/~cecilia/papers/icsr.pdf>.

[15] Beck, K. Embracing Change with Extreme Programming. *IEEE Computer.* 32(10), p. 70-77, 1999.

[16] Bederson, B.B., Grosjean, J., and Meyer, J. Toolkit Design for Interactive Structured Graphics. *IEEE Transactions on Software Engineering* 30(8), p. 535-546, August, 2004. <http://portal.acm.org/citation.cfm?id=1018036.1018385>.

[17] Binns, P., Englehart, M., Jackson, M., and Vestal, S. Domain-Specific Software Architectures for Guidance, Navigation and Control. *International Journal of Software Engineering and Knowledge Engineering*. 6(2), p. 201-227, June, 1996.

[18] Biron, P. and Malhotra, A. *XML Schema Part 2: Datatypes.* World Wide Web Consortium, W3C Recommendation Report, May 2, 2001. < http://www.w3.org/TR/xmlschema-2/>.

[19] Booch, G. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings, 1993.

[20] Booch, G., Garlan, D., Iyengar, S., Kobryn, C., and Stavridou, V. *Is UML an Architectural Description Language?* <http://www.acm.org/sigplan/oopsla/oopsla99/2_ap/tech/2d1a_uml.html>, OOPSLA '99, Website, 1999.

[21] Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modeling Language User Guide*. 2nd ed. Addison-Wesley Object Technology Series. Reading, Massachusetts: Addison-Wesley Professional, 2005.

[22] Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. ACM Press, Addison-Wesley Professional: Reading, Massachusetts, 2000.

[23] Boucke, N., Garcia, A., and Holvoet, T. Composing Architectural Crosscutting Structures in xADL. In *Proceedings of the 10th Workshop on Early Aspects - Aspect Oriented Requirements Engineering and Architecture Design (AOS'07).* Vancouver, Canada, 2007.

[24] Box, D., Enhnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., and Winer, D. *Simple Object Access Protocol (SOAP) 1.1*. <http://www.w3.org/TR/SOAP/>, World Wide Web Consortium (W3C), HTML, 2000.

[25] Bray, T., Paoli, J., and Sperberg-McQueen, C.M. *Extensible Markup Language (XML): Part I. Syntax.* World Wide Web Consortium, Recommendation Report, February, 1998. <http://www.w3.org/TR/1998/REC-xml>.

[26] Brooks, F.P. *The Mythical Man-Month: Essays on Software Engineering*. 2 ed. 336 pgs., Addison-Wesley, 1995.

[27] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., and Grose, T.J. *Eclipse Modeling Framework*. 1st ed. The Eclipse Series. Gamma, E., Nackman, L., and Wiegand, J. ed. 720 pgs., Addison-Wesley Professional, 2003.

[28] Carnegie Mellon University. *How Do You Define Software Architecture?* <http://www.sei.cmu.edu/architecture/definitions.html>, Software Engineering Institute, Webpage, 2005.

[29] Carrel-Billiard, M. and Akerley, J. *Programming With VisualAge for Java*. Book and CD-ROM ed. VisualAge Series. 486 pgs., Prentice Hall, 1998.

[30] Chen, P.S. *The Entity-Relationship Approach to Software Engineering*. Elsevier Science, 1983.

[31] Chen, P.S., Critchlow, M., Garg, A., Van der Westhuizen, C., and van der Hoek, A. Differencing and Merging within an Evolving Product Line Architecture. In *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5).* p. 269-281, Siena, Italy, November 4-6, 2003.

[32] Clark, J. *XSL Transformations (XSLT) Version 1.0.* <http://www.w3.org/TR/xslt>, World Wide Web Consortium (W3C), Specification, 1999.

[33] Clark, J. and DeRose, S. *XML Path Language (XPath) Version 1.0.* World Wide Web Consortium, W3C Recommendation Report REC-xpath-19991116, November 16, 1999. <http://www.w3.org/TR/xpath>.

[34] Clements, P. and Northrop, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley: New York, New York, 2002.

[35] CodeGear. *JBuilder*. <http://www.codegear.com/products/jbuilder>, 2007.

[36] CollabNet. *Subclipse*. <http://subclipse.tigris.org/>, HTML, 2004.

[37] Collins-Sussman, B., Fitzpatrick, B.W., and Pilato, C.M. *Version Control with Subversion*. <http://svnbook.red-bean.com/>, HTML, 2004.

[38] Consultative Committee for Space Data Systems. *Reference Architecture for Space Data Systems.* Report, March, 2003.

[39] Dashofy, E., van der Hoek, A., and Taylor, R.N. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. *ACM Transactions on Software Engineering and Methodology (TOSEM).* 14(2), p. 199-245, April, 2005.

[40] Dashofy, E. *The Myx Architectural Style.* University of California, Irvine, Whitepaper Report, 2006. <http://www.isr.uci.edu/projects/archstudio/resources/myx-whitepaper.pdf>.

[41] Dashofy, E.M. Issues in Generating Data Bindings for an XML Schema-Based Language. In *Proceedings of the Workshop on XML Technologies in Software Engineering (XSE 2001).* Toronto, Canada, May 15, 2001.

[42] Dashofy, E.M. and van der Hoek, A. Representing Product Family Architectures in an Extensible Architecture Description Language. In *Proceedings of the International Workshop on Product Family Engineering (PFE-4).* p. 330-341, October, 2001.

[43] Dashofy, E.M., van der Hoek, A., and Taylor, R.N. A Highly-Extensible, XML-Based Architecture Description Language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001).* Amsterdam, The Netherlands, August 28-31, 2001.

[44] Dashofy, E.M., van der Hoek, A., and Taylor, R.N. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002).* p. 266-276, ACM. Orlando, Florida, May, 2002.

[45] DeRemer, F. and Kron, H.H. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering.* 2(2), p. 80-86, June, 1976.

[46] DeRose, S., Maler, E., and Orchard, D. *XML Linking Language (XLink) Version 1.0.* World Wide Web Consortium, W3C Recommendation Report, June 27, 2001. <http://www.w3.org/TR/xlink/>.

[47] Dictionary.com. *Dictionary.com*. <http://dictionary.reference.com/>, Website, 2007.

[48] Dijkstra, E. The Humble Programmer. *Communications of the ACM.* 15(10), p. 859-866, 1972. <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF>.

[49] Dillinger, M., Madani, K., and Alonistioti, N. *Software Defined Radio: Architectures, Systems and Functions*. Wiley, 2003.

[50] DoD Architecture Framework Working Group. *DoD Architecture Framework, Version 1.0*. United States Department of Defense, Report, February 9, 2004. <http://www.defenselink.mil/nii/doc/DoDAF_v1_Volume_I.pdf>.

[51] Eclipse Foundation. *Eclipse*. <http://www.eclipse.org/>, HTML, 2007.

[52] Ellsiepen, P., De Deus Silva, M., and Walsh, A. *XASTRO Framework And Model Prototype.* European Space Agency, Report DTOS-SST-TN-0691-TOS-GIC, p. 117, October 1, 2002. <http://www.ccsds.org/docu/dscgi/ds.py/Get/File-434/Framework_and_Model.doc>.

[53] Falcarin, P. and Alonso, G. Software Architecture Evolution through Dynamic AOP. In *Proceedings of the First European Workshop on Software Architecture.* Springer. St. Andrews, Scotland, May 21-22, 2004. <http://www.springerlink.com/index/5FG08HWEY96885GH.pdf>.

[54] Fallside, D.C. *XML Schema Part 0: Primer.* World Wide Web Consortium, W3C Recommendation Report, May 2, 2001. <http://www.w3.org/TR/xmlschema-0/>.

[55] Feiler, P.H., Lewis, B., and Vestal, S. The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. In *Proceedings of the RTAS 2003 Workshop on Model-Driven Embedded Systems.* Washington, D.C., May, 2003.

[56] Fielding, R., Gettys, J., Mogul, J.C., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. *Hypertext Transfer Protocol -- HTTP/1.1*. Internet Engineering Task Force, Request for Comments Report 2616, June, 1999.

[57] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 Users Manual*. 82 pgs., 2005.

[58] Galvin, S., Exton, C., and McGurren, F. Enhancing the Role of Interfaces in Software Architecture Description Languages. In *Proceedings of the Workshop on Architecture Description Languages.* Tolouse, France, August 27, 2004.

[59] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Professional: Reading, MA, 1995.

[60] Garlan, D. and Notkin, D. Formalizing Design Spaces: Implicit Invocation Mechanisms. In *Proceedings of the VDM '91: 4th International Symposium of VDM Europe on Formal Software Development Methods.* p. 31-44, Springer-Verlag. Noordwijkerhout, The Netherlands, October 21-25, 1991.

[61] Garlan, D., Monroe, R.T., and Wile, D. ACME: An Architecture Description Interchange Language. In *Proceedings of the CASCON '97.* p. 169-183, IBM Center for Advanced Studies. Toronto, Ontario, Canada, November, 1997.

<http://www-2.cs.cmu.edu/afs/cs/project/able/ftp/acme-cascon97/acme-cascon97.pdf>.

[62] Georgas, J.C. and Taylor, R.N. Towards a Knowledge-based Approach to Architectural Adaptation Management. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems (WOSS '04).* p. 59-63, ACM Press. Newport Beach, CA, October, 2004.

[63] Goldman, N. and Balzer, R. The ISI Visual Design Editor Generator. In *Proceedings of the IEEE Symposium on Visual Languages.* p. 20-27, Tokyo, 1999.

[64] Gomaa, H. and Wijesekera, D. The Role of UML, OCL and ADLs in Software Architecture. In *Proceedings of the 1st ICSE Workshop on Describing Software Architecture with UML.* Toronto, Ontario, Canada, May 15, 2001, 2001.

[65] Gorlick, M.M. and Razouk, R.R. Using Weaves for Software Construction and Analysis. In *Proceedings of the 13th International Conference on Software Engineering.* p. 23-34, May, 1991.

[66] Gruber, O., Hargrave, B.J., McAffer, J., Rapicault, P., and Watson, T. The Eclipse 3.0 platform: adopting OSGi technology. *IBM Systems Journal.* 44(2), p. 289-299, July, 2005.

[67] Grundy, J. Software Architecture Modelling, Analysis and Implementation with SoftArch. In *Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS).* IEEE. Maui, Hawaii, Jan 3-6, 2001. <http://www.cs.auckland.ac.nz/~john-g/papers/hicss34.ps.gz>.

[68] Harel, D. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming.* 8, p. 231-274, 1987.

[69] Harold, E.R. *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX*. 1st ed. 1120 pgs., Addison-Wesley Professional, 2002.

[70] Hendrickson, S.A., Dashofy, E.M., and Taylor, R.N. An Approach for Tracing and Understanding Asynchronous Architectures. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003).* p. 318-322, Montreal, Quebec, Canada, October 6-10, 2003.

[71] Hendrickson, S.A., Dashofy, E.M., and Taylor, R.N. An Approach for Tracing and Understanding Asynchronous Architectures. Short paper. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003).* p. 318-322, Montreal, Quebec, Canada, October 6-10, 2003.

[72] Hendrickson, S.A., Dashofy, E.M., and Taylor, R.N. An (Architecture-Centric) Approach for Tracing, Organizing, and Understanding Events in Event-Based Software Architectures. In *Proceedings of the 13th International Workshop on Program Comprehension, in conjunction with ICSE 2005.* p. 227-236, St. Louis, MO, May 15-16, 2005.

[73] Hendrickson, S.A., Jett, B., and van der Hoek, A. Layered Class Diagrams: Supporting the Design Process. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006).* p. 722-736, Genova, Italy, October 1-6, 2006.

[74] Hendrickson, S.A. and van der Hoek, A. Modeling Product Line Architectures through Change Sets and Relationships. In *Proceedings of the 29th International*

Conference on Software Engineering (ICSE 2007). Minneapolis, MN, May 20-26, 2007. To Appear.

[75] Hilliard, R. Using the UML for Architectural Description. In *Proceedings of the «UML»'99: The Unified Modeling Language - Beyond the Standard, Second International Conference.* Fort Collins, CO, USA, 1999.

[76] Hoare, C.A.R. Communicating Sequential Processes. *Communications of the ACM.* 21(8), p. 666-677, August, 1978.

[77] Hofmeister, C., Nord, R.L., and Soni, D. Describing Software Architecture with UML. In *Proceedings of the First IFIP Working Conference on Software Architecture.* San Antonio, TX, February, 1999. <http://citeseer.nj.nec.com/cache/papers/cs/15435/http:zSzzSzwww.scr.siemens.comzSzpdfzSzUsingUML-unix.pdf/hofmeister99describing.pdf>.

[78] Holzmann, G.J. The Model Checker SPIN. *IEEE Transactions on Software Engineering.* 23(5), p. 279-295, May, 1997.

[79] Honeywell Inc. *DOME Guide*. 227 pgs., 1999.

[80] IBM Corporation. *Rational Rose*. <http://www-306.ibm.com/software/awdtools/developer/rosexde/>.

[81] IBM International Technical Support Organization. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework.* 1st ed. IBM Redbooks. 256 pgs., 2004.

[82] IEEE. *IEEE Std 1003.1, 2004 Edition.* Report, 2004. <http://www.unix.org/version3/ieee_std.html>.

[83] Institute for Software Research. *xADL 2.0*. <http://www.isr.uci.edu/projects/xarchuci/>, University of California, Irvine.

[84] International Organization for Standardization. *Information technology – Open Systems Interconnection – Remote Procedure Call (RPC).* ISO/IEC 11578:1996 [Defines UUIDs], Report, 1996.

[85] ISO. LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior. (ISO 8807), 1989.

[86] ISO/IEC. *Information technology - Open Distributed Processing - Reference model.* ISO/IEC, Report 10746, December 15, 1998.

[87] Jackson, D. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM).* 11(2), p. 256-290, April, 2002.

[88] Jacobson, I. *Object-Oriented Software Engineering: A Use Case Driven Approach*. 1st ed. 552 pgs., Addison-Wesley Professional, 1992.

[89] Jansen, A. and Bosch, J. Software Architecture as a Set of Architectural Design Decisions. In *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture.* p. 109-119, Pittsburgh, November, 2005. <http://gradius.home.fmf.nl/research/papers/conference/wicsa2005_final.pdf>.

[90] Jelliffe, R. *Schematron: A Language for Making Assertions about Patterns found in XML Documents.* <http://www.schematron.com/>, topologi.com, Website, 2006.

[91] Jensen, C. and Scacchi, W. Collaboration, Leadership, Control, and Conflict Negotiation in the NetBeans.org Software Development Community. In

Proceedings of the Thirty Eighth Hawaii International, Conference Systems Science (HICSS38-OSSD). Waikola Village, Kona, HI, January, 2005.

[92] JGraph Ltd. *Java Graph Visualization and Layout*. <http://www.jgraph.com/>, 2007.

[93] Johnson, S.C. *Yacc-Yet Another Compiler Compiler.* AT&T Bell Laboratories, Manual Report, 1975.

[94] Kadia, R. Issues Encountered in Building a Flexible Software Development Environment: Lessons from the Arcadia Project. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments.* p. 169-180, ACM Press. Tyson's Corner, Virginia, United States, December 9-11, 1992.

[95] Kaiser, G.E., Feiler, P.H., and Popovich, S.S. Intelligent Assistance for Software Development and Maintenance. *IEEE Software.* 5(3), p. 40-49, May, 1988.

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

[97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97).* p. 220-242, Jyväskylä, Finland, June 9-13, 1997.

[98] Kruchten, P. The 4+1 View Model of Architecture. *IEEE Software.* 12(6), p. 42-50, November, 1995. <http://www.rational.com/media/whitepapers/Pbk4p1.pdf>.

[99] Le Hors, A., Le Hégaret, P., Wood, L., Nicol, G., Robie, J., Champion, M., and Byrne, S. *Document Object Model (DOM) Level 3 Core Specification.* World Wide Web Consortium, W3C Working Draft Report, June 9, 2003. <http://www.w3.org/TR/2003/WD-DOM-Level-3-Core-20030609/>.

[100] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. *The Generic Modeling Environment.* Vanderbilt University, Technical Report Report, p. 6, 2000. <http://www.isis.vanderbilt.edu/Projects/gme/GME2000Overview.pdf>.

[101] Lindman, N., Walsh, A., Ellsiepen, P., and de Deus Silva, M. XASTRO - An XML Based Space Data Exchange Framework. In *Proceedings of the Conference on SpaceOps 2002.* Houston, Texas, October, 2002. <http://www.aiaa.org/Spaceops2002Archive/papers/SpaceOps02-P-T5-13.pdf>.

[102] Luckham, D.C. and Vera, J. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering.* 21(9), p. 717-734, September, 1995.

[103] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC 95).* 989, p. 137-153, Springer-Verlag, Berlin. 1995. <http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=651497>.

[104] McIlroy, M.D. 'Mass Produced' Software Components. In *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee* Naur, P. and Randell, B. eds. p. 138-155: Garmisch, 1968.

[105] Medvidovic, N., Rosenblum, D.S., and Taylor, R.N. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. p. 44-53, IEEE Computer Society. Los Angeles, CA, May 16-22, 1999. <http://www.ics.uci.edu/~dsr/old-home-page/icse99-dradel.pdf>.

[106] Medvidovic, N. and Taylor, R.N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*. 26(1), p. 70-93, January, 2000. Reprinted in Rational Developer Network: Seminal Papers on Software Architecture. Rational Software Corporation, <http://www.rational.net/>, 2001.

[107] Medvidovic, N., Rosenblum, D.S., Redmiles, D., and Robbins, J.E. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 11(1), p. 2-57, January, 2002. <http://doi.acm.org/10.1145/504087.504088>.

[108] Medvidovic, N., Dashofy, E., and Taylor, R.N. Moving Architectural Description from Under the Technology Lamppost. *Information and Software Technology*. 49(1), p. 12-31, January, 2007.

[109] MetaCase. ABC to MetaCase Technology. MetaCase: Irving, TX, 2007. <http://www.metacase.com/papers/ABC_to_metaCASE.pdf>.

[110] Microsoft Corporation. *Visual Studio Developer Center*. <http://msdn.microsoft.com/vstudio/>, 2007.

[111] Microsoft Corporation. *Visio Home Page - Microsoft Office Online*. <http://office.microsoft.com/visio>, 2007.

[112] Microsoft Corporation. *PowerPoint Home Page - Microsoft Office Online*. <http://office.microsoft.com/powerpoint>, 2007.

[113] Milligan, M.K.J. Implementing COTS Open Systems Technology on AWACS. *CrossTalk: The Journal of Defense Software Engineering*. September, 2000. <http://www.stsc.hill.af.mil/crosstalk/2000/09/milligan.html>.

[114] Modular Software-programmable Radio Consortium. *Software Communications Architecture Specification v2.2*. Specification Report MSRC-5000SCA, p. 165, November 17, 2001.

[115] Mukerji, J. and Miller, J. eds. *MDA Guide Version 1.0.1*. Object Management Group, 2003.

[116] Naslavsky, L., Xu, L., Dias, M., Ziv, H., and Richardson, D.J. Extending xADL with Statechart Behavioral Specification. In *Proceedings of the Twin Workshops on Architecting Dependable Systems (WADS '04), in conjunction with ICSE 2004*. Edinburgh, Scotland, UK, May 25, 2004.

[117] Nentwich, C., Capra, L., Emmerich, W., and Finkelstein, A. xlinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology (TOIT)*. 2(2), p. 151-185, May, 2002. <http://www.systemwire.com/whitepapers/xlinkit.pdf>.

[118] Nentwich, C., Emmerich, W., Finkelstein, A., and Ellmer, E. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology*. 12(1), p. 28-63, 2003.

[119] Northover, S. and Wilson, M. *SWT: The Standard Widget Toolkit, Volume 1*. The Eclipse Series. 592 pgs., Addison-Wesley Professional, 2004.

[120] OASIS Group. *eXtensible Access Control Markup 2 Language (XACML) Version 1.0*. Report, p. 132, February 18, 2003. <http://www.oasis-open.org/committees/download.php/2406/oasis-xacml-1.0.pdf>.
[121] Object Management Group. ed. *The Common Object Request Broker: Architecture and Specification*. 946 pgs., Object Management Group, 2001.
[122] Object Management Group. *Meta Object Facility (MOF) Specification*. <http://www.omg.org/cgi-bin/doc?formal/02-04-03.pdf>, PDF, 2002.
[123] Object Management Group. *OMG XML Metadata Interchange (XMI) Specification*. <http://www.omg.org/cgi-bin/doc?formal/02-01-01.pdf>, PDF, 2002.
[124] Object Management Group. *UML Profile for DODAF/MODAF (UPDM) Request for Proposal*. OMG, RFP Report syseng/05-08-01, August 22, 2005.
[125] Object Management Group. *OMG SysML Specification Coversheet*. Report, p. 262, March 28, 2007. <http://www.omg.org/cgi-bin/apps/doc?ptc/07-02-04.pdf>.
[126] Ommering, R.v., Linden, F.v.d., Kramer, J., and Magee, J. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*. 33(3), p. 78-85, March, 2000.
[127] Oreizy, P., Medvidovic, N., and Taylor, R.N. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*. p. 177-186, IEEE Computer Society. Kyoto, Japan, April, 1998. <http://www.ics.uci.edu/~peymano/papers/ICSE98.pdf>.
[128] Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., and Wolf, A.L. An Architecture-based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*. 14(3), p. 54-62, May-June, 1999.
[129] Perry, D.E. and Wolf, A.L. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*. 17(4), p. 40-52, October, 1992.
[130] Popkin Software. *System Architect User Guide*. Popkin Software, 2003.
[131] Rational Software Corporation. *Rational Rose: Using Rose*. IBM Corporation, Report 800-024462-000, p. 258, 2003. <ftp://ftp.software.ibm.com/software/rational/docs/v2003/win_solutions/rational_rose/rose_user.pdf>.
[132] Reenskaug, T. The Model-View-Controller (MVC): Its Past and Present. In *Proceedings of the Java and Object-Oriented Software Engineering (JAOO)*. Aarhus, Denmark, 22-25 September, 2003. <http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf>.
[133] Reiss, S.P. *Interacting with the FIELD Environment*. Brown University Department of Computer Science, Report CS-89-51, p. CS-89-51, May 1989, 1989.
[134] Ren, J. and Taylor, R.N. Visualizing Software Architecture with Off-The-Shelf Components. In *Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2003)*. p. 132-141, San Francisco, CA, July 1-3, 2003.

[135] Ren, J. *A Connector-Centric Approach to Architectural Access Control.* Thesis. Information and Computer Science, University of California, Irvine, p. 222, 2006. <http://www.ics.uci.edu/~jie/Thesis.pdf>.

[136] Robbins, J., Hilbert, D., and Redmiles, D. Using Critics to Analyze Evolving Architectures. In *Proceedings of the Second International Software Archichitecture Workshop (ISAW-2).* 1996. <http://www.ics.uci.edu/~jrobbins/papers/ISAW96.pdf>.

[137] Robbins, J., Hilbert, D., and Redmiles, D. Argo: A Design Environment for Evolving Software Architectures. In *Proceedings of the Nineteenth International Conference on Software Engineering.* ACM Press. Boston, MA, 1997.

[138] Robbins, J., Medvidovic, N., Redmiles, D., and Rosenblum, D. Integrating Architecture Description Languages with a Standard Design Method. In *Proceedings of the 20th International Conference on Software Engineering (ICSE).* p. 209-218, ACM Press. Kyoto, Japan, April, 1998. <http://www.isr.uci.edu/architecture/papers/TR-ICS-UCI-97-35.pdf>.

[139] Robbins, J. and Redmiles, D. Software Architecture Critics in the Argo Design Environment. In *Proceedings of the International Conference on Intelligent User Interfaces (UIST '98).* p. 47-60, San Francisco, CA, January 6-9, 1998.

[140] Robbins, J. and Redmiles, D. Software Architecture Critics in the Argo Design Environment. *Knowledge Based Systems.* p. 47-60, 1998.

[141] Robinson, A. AWACS Awaits Essential Upgrades. In *Air Combat Command,* 2006. <http://www.acc.af.mil/news/story.asp?id=123026125>.

[142] Roshandel, R., Schmerl, B., Medvidovic, N., Garlan, D., and Zhang, D. Understanding Tradeoffs among Different Architectural Modeling Approaches. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04).* Oslo, Norway, June, 2004.

[143] Rouquette, N. and Reinholtz, K. *The Mission Data System's Software Architecture Framework.* <http://www-scf.usc.edu/~csci577/teams/team12a/MDS/mds_sw_arch_framework.ppt>, JPL, PowerPoint, presented at the International Conference on Software Engineering (ICSE 2002), Orlando, Florida, 2002.

[144] Rumbaugh, J., Blaha, M., Lorensen, W., Eddy, F., and Premerlani, W. *Object-Oriented Modeling and Design*. 1st ed. Prentice Hall, 1990.

[145] SAX Project. *SAX: Simple API for XML*. <http://www.saxproject.org/>, Website, 2003.

[146] Schmerl, B. and Garlan, D. Exploiting Architectural Design Knowledge to Support Self-repairing Systems. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering.* p. 241-248, ACM Press. Ischia, Italy, July 2002, 2002. <http://portal.acm.org/citation.cfm?id=568804>.

[147] Schön, D. *The Reflective Practitioner: How Professionals Think in Action*. 374 pgs., Basic Books, Inc. Publishers: New York, 1983.

[148] Shaw, M. and Garlan, D. *Software Architecture: Perpectives on an Emerging Discipline*. 242 pgs., Prentice Hall, 1996.

[149] Software Engineering Institute. *ACMEStudio*. <http://www.cs.cmu.edu/~acme/AcmeStudio>, Carnegie Mellon University.

[150] Sommerville, I. *Software Engineering (7th Edition)*. Addison Wesley: Reading, MA, 2004.
[151] Spencer, J. *Architecture Description Markup Language (ADML): Creating an Open Market for IT Architecture Tools.* The Open Group, White Paper Report, September 26, 2000. <http://www.opengroup.org/tech/architecture/adml/background.htm>.
[152] Spivey, J.M. *The Z Notation: A Reference Manual*. Prentice-Hall International Series In Computer Science. 155 pgs., Prentice-Hall International: Englewood Cliffs, N.J., 1989.
[153] Sreedhar, V.C. Mixin'Up components. In *Proceedings of the 24th International Conference on Software Engineering.* ACM Press. Orlando, Florida, 2002.
[154] Stanford Research Institute. *Protégé 2000 User's Guide*. 259 pgs., SRI, 2005.
[155] Steele, G. *Common Lisp: the Language*. 2nd ed. 1029 pgs., Digital Press: Woburn, MA, 1990.
[156] Sun Microsystems. *The Java HotSpot Virtual Machine*. Whitepaper Report, September, 2002. <http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf>.
[157] Sun Microsystems. *Crimson*. <http://xml.apache.org/crimson/>, Apache Group, Website, 2003.
[158] Sun Microsystems. *Java Architecture for XML Binding (JAXB)*. <http://java.sun.com/xml/jaxb/>, Website, 2003.
[159] SysML Partners. *Systems Modeling Language (SysML) Specification version 0.9*. Report, p. 270, January 10, 2005. <http://www.sysml.org/artifacts/spec/SysML-v0.9-PDF-050110R1.pdf>.
[160] SysML.org. *SysML Partners*. <http://www.sysml.org/partners.htm>, Webpage, 2007.
[161] Taylor, R.N., Medvidovic, N., Anderson, K.M., E. James Whitehead, J., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D.L. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*. 22(6), p. 390-406, June, 1996.
[162] Taylor, R.N. and Redmiles, D. *Evolutionary Design of Complex Systems Open Technology for Software Evolution: Hyperware, Architecture, and Process.* Department of Information and Computer Science, University of California Irvine, Technical Report, September, 1998. <http://ftp.ics.uci.edu/pub/edcs/reports/quarterly_reports/EDCS-QR-Sep98-pub.pdf>.
[163] Teitelman, W. and Masinter, L. The Interlisp Programming Environment. *Computer*. 14(4), p. 25-34, April, 1981.
[164] Telecommunication Standardization Sector of ITU. *Specification and Description Language (SDL)*. Report ITU Standard Z.100, 2002. <http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf>.
[165] The Apache Software Foundation. *Xalan-Java*. <http://xml.apache.org/xalan-j/>, 2007.
[166] The Omni Group. *The Omni Group - Applications - OmniGraffle*. <http://www.omnigroup.com/applications/omnigraffle/>, 2007.

[167] The Standish Group International, I. *Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50%*. The Standish Group, Press Release Report, March 25, 2003. <http://www.standishgroup.com/press/article.php?id=2>.

[168] The Standish Group International, I. *2004 Third Quarter Research Report.* Report, 2004. <http://www.standishgroup.com/sample_research/PDFpages/q3-spotlight.pdf>.

[169] Thomas, I. and Nejmeh, B.A. Definitions of Tool Integration for Environments. *IEEE Software*. 9(2), p. 29-35, March, 1992.

[170] Thompson, H.S., Beech, D., Maloney, M., and Mendelsohn, N. *XML Schema Part 1: Structures.* World Wide Web Consortium, W3C Recommendation Report, May 2, 2001. <http://www.w3.org/TR/xmlschema-1/>.

[171] Thompson, H.S. and Tobin, R. *Current Status of XSV.* University of Edinburgh, Report, July 9, 2003. <http://www.ltg.ed.ac.uk/~ht/xsv-status.html>.

[172] University of California, B. *UNIX Programmer's Manual, 4.3 Berkeley Software Distribution*. Computer Science Division, University of California, Berkeley, 1986.

[173] Vieira, M.E.R., Dias, M.S., and Richardson, D.J. Analyzing Software Architectures with Argus-I. In *Proceedings of the International Conference on Software Engineering (ICSE 2000).* p. 758-761, Limerick, Ireland, June 4-11, 2000.

[174] Walrath, K., Campione, M., Huml, A., and Zakhour, S. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. 2nd ed. 784 pgs., Prentice Hall, 2004.

[175] Warmer, J.B. and Kleppe, A.G. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Object Technology Series. Addison-Wesley Professional, 1998.

[176] Williams, S. and Kindel, C. *The Component Object Model: A Technical Overview*. <http://msdn.microsoft.com/library/techart/msdn_comppr.htm>, Microsoft Corporation, HTML, 1994.