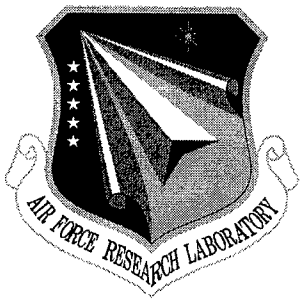
AFRL-IF-RS-TR-2001-254
Final Technical Report
December 2001



RESEARCH IN ADVANCED ENVIRONMENTS

University of California

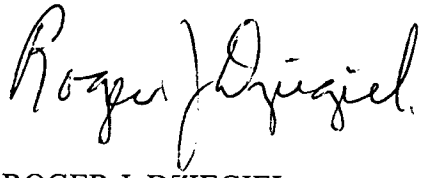
Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. B130

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-254 has been reviewed and is approved for publication.

APPROVED: 

ROGER J. DZIEGIEL
Project Engineer

FOR THE DIRECTOR: 

MICHAEL TALBERT, Maj., USAF, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	DECEMBER 2001	Final Jun 94 - Jan 98

4. TITLE AND SUBTITLE
RESEARCH IN ADVANCED ENVIRONMENTS

5. FUNDING NUMBERS
C - F30602-94-C-0218
PE - 62301E
PR - B130
TA - 01
WU - 01

6. AUTHOR(S)
Richard N. Taylor, Debra J. Richardson, Richard W. Selby, and Michal Young

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California
Department of Information & Computer Science
Irvine California 92697-3425

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

Air Force Research Laboratory/IFTD
525 Brooks Road
Rome New York 13441-4514

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLR-IF-RS-TR 2001 254

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Roger J. Dziegiel/IFTD/(315) 330-2185

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Active process support is being created to help human developers, including non-technical managers, coordinate their activities while integrating commercial project planning tools. Analysis and testing tools are being created to provide high assurance in software, through provision of prerun-time analysis, test case development, test result checking, and test process management tools. Powerful program analysis tools are being created which enable early detection of subtle coordination errors in concurrent systems composed of heterogeneous parts. Evolvable software architectures, first in the domain of user interface software, are being created to enable a more component-based software economy.

World-Wide-Web (WWW) and hypermedia technology is being developed to foster easy information access, connections between software artifacts, and human understanding of complex systems. The UCI/Purdue team developed these technologies and validated them through interaction with the aerospace community, military organizations, the commercial software world, and the open Internet community.

14. SUBJECT TERMS
Distributed Software Development, Groupware, Software Architecture, High Assurance Software, Software Understanding, Hyperware

15. NUMBER OF PAGES
52

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

Executive Summary

Cost-effective evolution of complex software systems requires evolvable software product architectures, support for development team members (coordination, understanding), and powerful tools (including analysis, test, measurement, visualization, artifact connection). The University of California, Irvine/University of Oregon team has worked at developing these technologies and validating them through interaction with the aerospace community (Hughes, Northrop Grumman, Loral, TASC, Lockheed Martin, NASA Ames Research Center), military organizations (MICOM), the commercial software world (Sun, FileNet, CoGenTex), and the open Internet community. These developments have been done in coordination with other research organizations in ITO's Environments Program, to yield a comprehensive, open, deep, integrated software engineering environment.

1 Objective

The fundamental objective of Arcadia was to develop technically rich solutions across the breadth of the problem (developer support, product architectures, tool technologies) that are technically compatible, functionally comprehensive, and mutually reinforcing.

Active process support was developed to help human developers, including non-technical managers, coordinate their activities while integrating commercial project planning tools. Analysis and testing tools were developed to provide high assurance in software, through provision of pre-run-time analyses, test case development, test result checking, and test process management tools. Powerful program analysis tools were developed which enable early detection of subtle coordination errors in concurrent systems composed of heterogeneous parts. Evolvable software architectures, first in the domain of user interface software, were developed to enable a more component-based software economy. World-Wide-Web and hypermedia technology were developed to foster easy information access, connections between software artifacts, and human understanding of complex systems.

2 Approach

Procedurally, the Arcadia project's approach was one of iteration, prototyping, and integration. Emphasis was placed on identification of principles that transcend idiosyncratic implementations. The integrating environment conceptual architecture was based on a set of coherent principles iteratively refined through experience with the prototype component implementations. Specific technology transition efforts with various consumers were used to validate the technology in actual practice situations. Separate technologies were first prototyped, then pair-wise integrations performed, then multi-way integrations.

The work in distributed, team-oriented software development was focused on determining effective ways of explicitly describing team-based activities, determining which parts of those activities could best be supported by automated tools, and technologies were developed to perform that automation. A critical aspect of this work was developing an effective basis for management of projects over the net; consequently, a distributed architecture implemented in Java was adopted. In a nutshell, this was research targeted at the intersection of software process research and computer-supported cooperative work (CSCW)/Groupware research. Specific technical innovation came from support for non-technical users, techniques supporting two-way integration with commercial project planning and communication tools, and support for reuse and adaptation of process fragments.

Providing high assurance in the dependability of software is a critical need in software-intensive systems. Under the UCI contract, one emphasis was on providing analysis support for incomplete and evolving systems, combining information from a heterogeneous set of formalisms and notations for specification, design, and code. The analysis techniques employed include pre-run-time analyses, including state-space analysis and dependence analysis, and run-time testing. Synergistic combinations of techniques were employed to obtain improved levels of assurance at acceptable cost.

Since a great deal of application software is dedicated to managing the user interface, and since requests for changes to user interfaces are common, our research in developing effectively evolvable product architectures focused initially on this domain. The approach was to emphasize component based architectures in which components communicate through a disciplined style of event notifications and requests. While building on past successes, the approach is notable for its support for distributed, heterogeneous (language, machine), multi-user systems and provision of a graphical architecture design environment.

Software products are complex entities, comprised of various kinds of parts, each of which is the product of some development tool and/or process. Capturing, managing, and navigating through this web of relations is essential for product evolution. The Arcadia project at UCI supported this through development of hypermedia services which enable convenient linkages between heterogeneous tools and data. This linkage service also supports connections to the World-Wide-Web (WWW) and some of its browsers (e.g. Mosaic). Additionally, some of our effort was a part of the WWW project itself, focused in particular on WWW protocols and name schemes.

The work in software understanding and reengineering had as its basis a focus on information derived from analysis and metric data. This work focused on the use of coarse-grain measurement of large-scale systems to aid in system structure evaluation and architecture analysis.

3 Accomplishments

3.1 Distributed, Team-Oriented Software Development

Maintaining control of software engineering projects has become increasingly difficult as the projects have grown in size, scope and complexity. Traditional management methods have fallen short of providing the control needed to manage these large scale software development efforts. They don't model workflow well, have limitations in their ability to specify expected

behavior, and support for detecting and correcting deviations is, if provided, severely limited and requires considerable manual involvement.

Project coordination, like control, is also important. Coordination includes making sure that everyone knows the current status of each part of the project, and making sure that each team member knows which tools to use. As with control, coordination becomes increasingly difficult with size. Tools must be able to work in a geographically distributed environment where team members are separated not only by walls, but also by large distances, diverse operating systems, and software unique to each site's environment.

Our intent was to build upon current environment prototypes, integrating an evolving and growing set of Arcadia capabilities. This approach served as a validation that the tools and technologies that we developed are compatible with each other and are mutually supportive of larger efforts to construct production environments.

[YT95] discussed why process programming languages are indeed programming languages, some of the approaches taken in software process research, and identified some of the areas in which existing programming languages have proved inadequate for defining and executing software development processes. The Teamware process programming language was described, and some of the language issues related to Teamware were addressed.

[YT94] described the consequences of considering human beings as the primary executors of a process program. The key differences between traditional applications written for hardware processors vs. software process specifications written for execution by humans were identified, and as a consequence of these differences, five key requirements on process programming languages for team coordination and control were identified. An examination of how Teamware meets these key requirements was followed by a comparison between Teamware's approach and that found in other process programming systems.

Initially, the bulk of our effort in this category of distributed, team-oriented software development was focused on the Teamware language and system. The language was defined before the project start. Early project activities were directed at its implementation and preparation for experimentation.

The design of the Teamware system employed the Chiron 1.4 user interface system to support graphical interactions. Artists supported editing of networks, categories, specifications, and instances. The foundation level, which provided implementation of the category object model, was built in a mixture of C++ and Python. The foundation level supported persistency, dynamic objects (a direct consequence of the Teamware object model), event-handlers as Python scripts, and metaprocesses. The basis for a distributed foundation level was in place, and decisions as to what object distribution policy should be followed were faced. The system level relied on foundation level objects for the bulk of its implementation. The system level was to be

populated with pre-defined language objects (as specified in the Teamware language reference manual) built out of the lower-level objects. End-to-end demonstrations of the new system relied on an incomplete set of system level objects. To aid development of system level objects we leveraged off of the Rational Rose tool.

Working versions of the system and user levels were developed. The implementation included a functioning Teamware interpreter and demonstrated Teamware's ability to integrate with external systems. Initial integration efforts showed Teamware interacting with both Teamware's native agenda tool, an external third party calendar system (ical), and a simple external third party messaging/mail system.

Development on Teamware was superseded by work on Endeavors. Based in part on experience with, and lessons learned from Teamware, process research, as exemplified in Endeavors, focused on providing a flexible process modeling and execution environment to help improve coordination and managerial control, and techniques for collecting application usage data over the Internet to help inform the development process. The Endeavors system is an open, distributed process modeling and execution infrastructure that addresses communication, coordination, and control issues. Complex processes may require: distribution of people and processes; event-based and intermediate-format integration of external tools; a low entry barrier through ease-of-use and incremental adoption; ability to customize and reuse objects, tools, and policies; and dynamic change in runtime processes, objects, and behaviors. Endeavors' solution architecture achieves these goals through application of five key design strategies: maintaining multiple object model layers; implementing the architecture as a set of highly componentized, lightweight, transportable, concurrent elements; providing customization capabilities for each layer of the object model; using a reflexive object model to support dynamic change; and allowing dynamic loading and changing of objects including loading of executable handlers, new object types, and extensions. Endeavors is described more fully in [BT96].

Experiments with several real processes by Pacific Bell and ISI provided valuable feedback on the design and implementation of Endeavors. An initial prototype of the Endeavors workgroup server, which allows the infrastructure for multiple people within a workgroup to collaborate on the execution of a workflow process, was developed using Sun Microsystems' Jeeves server. Experiments and designs to integrate other HTTP servers such as Apache and Netscape's Enterprise server were performed. An initial design for integration of a rule-based inference engine (Amber), from the University of Columbia, was produced.

3.2 Open Environment Architectures

Open environment architectures are critical to the establishment of a component based software development economy. Open environments can allow projects to incrementally incorporate new technologies and commercial off-the-shelf (COTS) tools. Our focus on open environment architectures had key emphases in hyperware, user interfaces, measurement and empirical evaluation, and interoperability.

3.2.1 Hyperware

Chimera is an open, serverized, hypermedia system that supports n-ary links between heterogeneous tools and applications in a network. Objects manipulated by separate applications can be linked together through Chimera. Chimera has bindings to C and Ada, and bindings to several popular tools have been constructed. Chimera makes no assumptions or demands regarding user interface systems employed, or how objects are stored. Chimera embodies the following technical attributes:

1. Heterogeneous object editor and viewer support: Many different applications can be integrated with Chimera allowing them to participate in a hyperweb.
2. Anchors specialized to particular views: Chimera handles just the links between objects and not their display, thus the applications which display the objects can make the display of anchors customized to the views of the objects.
3. Multiple-view, concurrent, and active displays: Chimera has a client-server architecture. The Chimera server handles connections from client applications and handles the routing of messages (e.g. link traversals) between them.
4. Links across heterogeneous object managers: Since an application manages the access and display of an object, and since anchors are created with respect to an object's view and not directly on an object, links between objects stored in different object managers (or even a file system) are easily established.
5. n-ary links: A link is not restricted to a source and destination. Instead, links are modeled as a set. One anchor can take a user to multiple destinations in Chimera and all of them can be viewed at once (limited only by screen real estate).

A more in-depth description of Chimera can be found in [ATW94].

Chimera was integrated with Ivan, a hyperweb visualization tool. Work was done to make the integration more robust and add increased functionality to the browser such as the ability to save and load hyperweb layouts and incremental updates of layouts as the hyperweb is modified. Chimera was also integrated with the web-browser Mosaic.

The Chimera abstract program interface (API) was specified in CORBA's interface definition language, and an experiment demonstrating the ability of a CORBA client to make calls on a Chimera server after a connection was established between the two using a CORBA Object Request Broker took place. Another CORBA-related project demonstrated the ability to give access to Chimera hyperwebs such that they can be traversed and queried by CORBA clients.

Further Chimera client development work focused on increasing client reliability, improving their user interface (especially with respect to error reporting) development of a new "Hotlist" client that provides the ability to maintain lists of hypertext anchors of interest to the user, and development of a "ChimeraScope" client which provides detailed visibility into the Chimera link database for Chimera developers.

Chimera was re-written in Java to utilize WWW protocols such as HTTP in order to take advantage of the global distribution features enabled by these protocols.

[And96a] and [And96c] described a technique to automatically provide extra-application hypertext functionality (HTF) for client applications by placing the responsibilities of being a client on the user-interface toolkit of an application. This technique addresses the consistency problems common to clients of open hypermedia systems since the toolkit can be extended to handle common error conditions and hypermedia commands automatically. While this technique cannot be used to incorporate HTF into legacy systems, it is still an important contribution to the general problem of incorporating HTF into information systems. The problem of upgrading future systems built with this technique to make use of new advances in HTF is reduced, because the majority of changes will most likely occur within the toolkit integration substrate.

Open hypermedia systems (OHSs) employ a variety of techniques to provide hypermedia services to a diverse range of applications. The World Wide Web is the largest distributed hypermedia system in use and was developed largely independent of the research in OHSs. The popularity of the Web along with problems inherent in its design has motivated OHS researchers to integrate their systems with it. Research in this area has primarily focused on enhancing the functionality of the Web via the services of an OHS. [And96b] presented three experiments exploring the integration of the Chimera OHS with the Web. While one of the experiments indeed described work which enhances the Web, the other two investigated ways in which the Web can beneficially enhance an OHS. How integration with the Web enables interoperability between OHSs was examined.

[ZO94] described issues of mutual interest and benefit to researchers at the intersection of hypertext and software development environments. On one hand, the domain of software development appears to be a natural candidate for applying hypertext. Hypertext and hypermedia

capabilities may be used in the creation, manipulation, examination and evolution of software products. On the other hand, software engineering principles and practices may be used to improve the quality and productivity of hypertext system development. In particular, software development environments may facilitate the provision of hypertext capabilities for those environments, which would otherwise have required substantial development efforts.

[ZR96] contended that large spaces of software elements and relationships may be viewed and navigated using hypertext. Hypertext provides both a clear conceptual metaphor as well as a useful set of browsing capabilities for software systems. IVAN (Integrated Visualization and Navigation) was created as a hypertext browser for webs of software artifacts and relationships, and its use in a test case failure scenario was examined. It was proposed that these methods could be extended to address the unique concerns of configuration management.

[ZR97a] further explored the intersection of hypertext and software development environments, and looked at a technique for adding uncertainty modeling, called Bayesian belief networks, for software uncertainties. These uncertainty models may provide support for a coherent conceptual correlation among the architecture of hypertext, software systems and Bayesian networks. Bayesian networks allow for uncertainties to be initially captured and later revised dynamically, allowing their update as software development progresses. Bayesian networks support multiple sources of evidence, which may be valuable since multiple sources of uncertainty, both internal and external, typically exist for most software systems. The feasibility of applying the Bayesian approach to a real system under development was examined.

[WFA96] explored how a fusion between distributed link data systems, as exemplified by the World-Wide Web (WWW), and link server systems, as exemplified by the Open Hypermedia Systems (OHS) community, might be achieved. In such a hybrid system, a user's interface to their information space is presented by a network of cooperating processes (similar to both WWW client "helper" applications and OHS client applications) which interact with a local link-server process. This local link server interacts with data managers which maintain a local, personal hypertext, along with a cache of the user's local context within the global hypertext, which may be expanded by retrieving artifacts and links from HyperText Transfer Protocol (HTTP) servers.

[Whi96a] described an architectural framework for modeling third-party application integrations with open hypermedia systems, which collects and extends the integration experience of the open hypermedia community. The framework was used to characterize applications prior to integration, and describe the qualities of a complete integration. Elements of the architectural model are artists, which are used to manipulate anchors, links, and native

application objects; communicators, which manage information flow to and from the open hypermedia system; and containers which group the other elements. Prior integration experience is collected in a standard way using the model. Guidance in selecting the final integration architecture is provided by this prior integration experience, in conjunction with the degree of difficulty of an integration, which is related to the integration architecture.

A World-Wide Web client protocol library for applications, libwww-ada95, written in Ada95, was built. The architecture of the library allows both direct binding of the library at compile time and independent operation as a retrieval service.

Work was done on the Hypertext Transfer Protocol (HTTP), which is the primary protocol for Web information transfer, and forms the basis for libwww-ada95's internal communication and cache behavior. The HTTP/1.0 protocol specification [BFF96] was published by the Internet Engineering Steering Group (IESG) as RFC 1945 in May of 1996. Version 1.1 [FG+97] was published in January of 1997.

Authoring work on the computer has meant having to save the work to the storage medium of the user's server. This typically has involved using operating services to write to a file system. This implies having some form of local access, often a login session on a machine connected via a local area network to a network file system containing the content exported by the Web server. A variation of this approach is to use FTP to remotely write content to a server's file system. A third option is to use HTTP capabilities to write directly to a name space of the Web server. [KW97] took a look at existing problems regarding authoring, especially collaborative work, and looks to the future of distributed authoring, including the formation of WebDAV, and extensions to HTTP to facilitate this type of work.

The Working Group on Versioning and Configuration Management of World Wide Web Content was created in the Spring quarter of 1996 to address the interoperability issues of versioning and configuration management that were anticipated to be encountered by distributed web authoring tools. In the same quarter, a working group was created whose mission was to make authoring on the World Wide Web as ubiquitous as browsing. These two groups were combined the following quarter to become the Working Group on World Wide Web Distributed Authoring and Versioning (WebDAV), with the goal of developing interoperability specifications for functionality which supports remote authoring and versioning of web content, functionality which is also supportive of widely distributed software engineering via the World Wide Web [Whi96b], [Whi96c]. In March, 1997, the Internet Engineering Task Force approved the charter of the WebDAV working group, making it an official working group of this Internet standards-setting body.

The WebDAV group submitted a WebDAV protocol specification [SVWD97] to the Internet Engineering Task Force as an Internet-Draft in February, 1997.

The WebDAV group recognized that while the HyperText Transfer Protocol, version 1.1 (HTTP/1.1), provides simple support for applications that allow remote editing of typed data, the existing features have proven inadequate to support efficient, scalable remote editing free of overwriting conflicts. [Whi96d] presented a list of features in the form of requirements that, if implemented, would improve the efficiency of common remote editing operations, provide a locking mechanism to prevent overwrite conflicts, improve relationship management support between non-HTML data types, provide a simple attribute-value metadata facility, and provide for the creation and reading of container data types. These requirements are also supportive of versioning capability.

[Whi97] described a rationale for the existence of a standards body such as WebDAV, charged with developing an interoperability specification for distributed authoring and versioning on the World Wide Web, and provided an overview of the capabilities that the working group proposed adding to the WWW.

An Internet proposed standard for Uniform Resource Locators (URL), which defines the syntax and semantics for addressing resources on the Internet [BFM96], was published in December of 1996. (A draft of this document appeared in [Fiel95]).

Most documents made available on the World Wide Web can be considered part of an infostructure - an information database with a specifically designed structure. Infostructures often contain a wide variety of information sources, in the form of interlinked documents at distributed sites, which are maintained by a number of different owners. Since it is rarely static, the content of the infostructure is likely to change over time and may vary from the intended structure. Maintenance of these structures has relied on error logs of each server, the complaints of users, and periodic manual traversals by each owner. [Fiel94] described the Multi-Owner Maintenance spider (MOMspider), which was developed to at least partially solve this problem. MOMspider can periodically traverse a list of webs, check each web for any changes requiring attention, and build a special index hypertext document listing the attributes and connections of the web. The design of MOMspider and how it was influenced by the nature of distributed hypertext and requirements for the good behavior of any web-traversing robot were described. Efficiency requirements for maintaining world-wide webs and proposed changes to HTML and HTTP to support distributed maintenance were examined.

Maintenance will be critical to digital libraries, especially those that promote access to diverse, informal materials. If ignored, maintenance issues within the digital library will threaten its usefulness and even its long term viability. [AF95] examined digital library collection maintenance from several vantage points, including software architecture and the type of collection, arguing that digital libraries containing informal and dynamic material will have greater maintenance problems. While several technical solutions were offered, it was emphasized that the problem is both technical and institutional, and both perspectives must be taken into account.

[Whi96e] presented the position that a peer-to-peer relationship between a hypertext system and a software configuration management (SCM) system would be expected to have several advantages, notably: use of a commercial SCM system allows hypertext to be applied to much larger software development efforts; preservation of existing investments in SCM technology; easier adoption of hypertext technology due to its cooperation with the existing environment; and a clean separation of versioning concerns between the hypertext system and the SCM system.

The World Wide Web has been successful in connecting islands of information, along with those who seek that information, both within local networks and across the global Internet. The ability to share information easily and effectively, without regard to physical location, has given rise to new forms of business and social endeavors. A *virtual enterprise* is an organization without the constraints of geographic location, and with a membership that often intersects multiple traditional organizations. The Web provides a minimum level of support for the needs of a virtual enterprise by helping in the discovery of shared goals and the people who share them, by providing a standard mechanism for reading the shared information space, and by assisting coordination. However, the existing support requires specialized software installations, non-standard interfaces, and remains paltry in regard to remote authoring and task coordination. [FWA+97] sought to identify those aspects of the WWW infrastructure that need improvement, and made specific recommendations on how they can be improved with respect to these issues.

3.2.2 Architectures and User Interfaces

[TNB+95] described the Chiron-1 user interface system. This system demonstrated key techniques that enable a strict separation of an application from its user interface. These techniques include separating the control-flow aspects of an application and user interface: they

are concurrent and may contain many threads. Chiron also separates windowing and look-and-feel issues from dialogue and abstract presentation decisions via mechanisms employing a client-server architecture. To separate application code from user-interface code, user-interface agents called *artists* are attached to instances of application abstract types (ADTs). Operations on ADTs within the application implicitly trigger user interface activities within the artists. Multiple artists can be attached to ADTs, providing multiple views and alternative forms of access and manipulation by either a single user or by multiple users. Each artist and the application run in separate threads of control. Artists maintain the user interface by making remote calls to an abstract depiction hierarchy in the Chiron server, insulating the user interface code from the specifics of particular windowing systems and toolkits. The Chiron server and clients execute in separate processes. The client-server architecture also supports multilingual systems: mechanisms are demonstrated that support clients written in programming languages other than the server, while nevertheless supporting object-oriented server concepts.

The User Interface Design Assistant (UIDA) was described in [Bol94] and [Bol95]. UIDA addresses the specific design problems of style and integration consistency throughout the user interface development process, and aids in the automated feedback and evaluation of a system's graphical user interface according to knowledge-based rules and project-specific design examples. UIDA is able to identify inconsistent style guide interpretations and UI design decisions resulting from distributed development of multiple UI sub-systems.

Chiron-2 represents a novel architectural style directed at supporting larger grain reuse and flexible system composition. Chiron-2, or C2 as it has come to be known, supports design of distributed, concurrent applications. A key aspect of the style is that components are not built with any dependencies on what typically would be considered lower-level components, such as user interface toolkits. All components in a C2 architecture are oblivious to the existence of any components to which notification messages are sent. Asynchronous notification messages and asynchronous request messages are the sole basis for inter-component communication. The C2 architecture is further described in [TMA+95] and [TMA+96], and a formal definition appeared in [Med95].

As a validation test-bed of the C2 style, the game KLAX was developed. This example, coded in C++, used a hierarchy of base classes to represent the basic objects in C2, components and connectors, with individual components and connectors derived from these base classes. This example also demonstrated multi-threaded, multi-process and multi-lingual implementations of a C2 architecture. Evidence from this exercise indicated that the C2 style does promote strong separation of concerns between components, provides a useful component-based abstraction for discussing the structure of a system, and is a viable technique for the

construction for non-trivial systems with a strong user interface aspect. While the focus has been on applications involving graphical user interfaces, the style has the potential for broader applicability.

SkyBlue, a constraint-solver from University of Washington, was incorporated into the *KLAX* example, followed by incorporation of a constraint manager from Carnegie Mellon University's Amulet user interface development environment. The intent of the exercise was to explore any potential global (architecture-wide) effect of substituting one constraint manager for another, as well as the possibility of multiple constraint managers being active in the same architecture. The changes necessary to substitute SkyBlue with Amulet proved to be localized to a given component, and the simultaneous usage of two constraint managers within an architecture, both at different levels of the architecture, and 'with'ing a single component. One conclusion of these efforts is that the C2 style offers significant potential for the development of application families and that wider trials were warranted. [MOT96], [MT96a], [MT96c], and [MT96d] presented further results from these exercises.

The issues and ramifications of applying object-oriented (OO) type theory to the C2 architectural style were examined in [MORT96]. Relating the two clarifies the composition properties of components and enables more aggressive architecture analysis. Specifically, by making subtyping relationships between components explicit, it can be determined when one component may be substituted for another. By extending type checking mechanisms beyond those available in object-oriented programming languages (OOPLs), a richer set of relationships can be expressed. More broadly, applying other concepts of OOPL typing to architecture enables a wider class of architectural analysis technique. The limits of applicability of OO typing to C2 and how they were addressed in the C2 ADL were also looked at. This work stemmed from a series of experiments conducted to investigate component reuse and substitutability in C2.

Software architectures shift the focus of developers from lines of code to coarser grained elements and their interconnection structure. Architecture description languages (ADLs) have been proposed as domain specific languages for the domain of software architecture, although there is still little agreement on what concerns are most importantly addressed in a study of software architecture, what aspects of an architecture should be modeled in an ADL, or even what an ADL is. [MR97] provided a framework of architectural domains for software, evaluated existing ADLs with respect to the framework, and studied the relationship between architectural and application domains. One lesson learned was that, while the architectural domains perspective enables one to approach architectures and ADLs in a new, more structured manner, further understanding of architectural domains, their tie to application domains, and their specific influence on ADLs is needed.

[Med96] presented a possible solution to the problem that existing ADLs typically support only static architecture specification and don't provide facilities for the support of dynamically changing architectures. The solution presented suggested that in order to adequately support dynamic architectural changes, ADLs can leverage techniques used in dynamic programming languages. In particular, changes to ADL specifications should be interpreted. To enable interpretation, an ADL should have an architecture construction component that supports explicit and incremental specification of architectural changes, in addition to the traditional architecture description facilities. This approach would allow software architects to specify the changes to an architecture after it was built. This expands upon the results from work in building a development environment for C2-style architectures.

[MT96b] and [Med97] attempted to provide an answer to the questions of what aspects of an architecture should be modeled by an ADL, and which of several possible ADLs is best suited for a particular problem. The distinction is rarely made between ADLs on one hand, and formal specification, module interconnection, simulation, and programming languages on the other. A motivation and a definition for a classification framework for ADLs was given, and the utility of the definition was demonstrated by using it to differentiate ADLs from other modeling notations. The framework was also used to classify and compare several existing ADLs.

Architecture-based software development is an approach to designing software in which developers focus on one or more high-level models of the software system rather than program source code. Choosing which aspects to model and how to evaluate them are two decisions that frame software architecture research. Some software architecture researchers have proposed special-purpose notations that have a great deal of expressive power but are not well integrated with common development methods. Others have used general-purpose notations that are accessible to developers, but lack details needed for extensive analysis. [RRR97] described an approach to combining the advantages offered by these two different kinds of notation. UML, an emerging standard notation for object-oriented design, was extended with semantics specific to C2, an architectural style for user interface intensive systems. Doing so suggested a practical strategy for bringing architectural modeling into the mainstream of software development and achieving partial integration of architectural models as needed.

Some researchers have proposed special-purpose architecture notations that have a great deal of expressive power but are not well integrated with common development tools, while others have used mainstream development methods that are accessible to developers but lack semantics needed for extensive analysis. [RMRR97] and [RMRR98] described an approach to combine the advantages of these two ways of modeling architectures. Two examples of extending UML, an emerging standard design notation, were presented for use with two

architecture description languages, C2 and Wright. The approach suggested a practical strategy for bringing architectural modeling into wider use by incorporating substantial elements of architectural models into a standard design method.

[RHR96c] presented approaches to architectural analysis that closely support evolution of a developing architecture by providing feedback as design decisions are made. This is in contrast to analysis techniques that provide feedback only after "complete" sequences of design decisions have been made and which do not directly support the evolutionary nature of the architectural design process. This approach more closely supports evolution and the needs of architects by providing feedback as individual design decisions are considered. The theoretical motivations for the critic-based approach, the implementation and management critics, support for diverse and extensible groups of critics, and the combined use of critics and existing analysis techniques were presented.

The Object Block Programming Environment (OBPE) was described in [RMR+96]. OBPE is a visual design, programming, and simulation tool which emphasizes support for both human-human and human-computer communication. BPE provides several features to support effective communication: multiple, coordinated views and aspects; customizable graphics; the "machines with push-buttons" metaphor; and the host/transient pattern. OBPE uses a diagram-based, visual object-oriented language that is intended for quickly designing and programming visual simulations of factories.

Software architectures are multi-dimensional entities that can be full understood only when viewed and analyzed at four different levels of abstraction: internal functionality of a component; the interface(s) exported by the component to the rest of the system; interconnection of architectural elements in an architecture; and rules of the architectural style. [MTW96] presented the characteristics of each of the four abstraction levels, outlined the kinds of analysis that need to be performed at each level, and discussed the kinds of formal notations that are suitable at each level. The pipe-and-filter and C2 architectural styles were used as illustrations. The formal models of C2 at the last three levels of abstraction were presented as a first step in enabling a C2 design environment to perform the necessary analyses of architectures.

A characterization of software architectures which become the foundation for the establishment of marketplaces for software components is described in [WRM+95]. Key properties that a software architecture should exhibit to be a basis of a component marketplace are described: multiple component granularities, substitutability of components, parameterizable components, customizable components, component development in multiple programming languages, component-specific help, composing component-specific user interface dialog and

presentation properties, easy distribution of components from seller to buyer, and support for multiple sales models. The C2 architectural style was then analyzed with respect to these properties to determine its potential as a future software component marketplace in the user interface domain.

[RWMT95] described plans for, and an initial prototype of the C2 software architecture design environment, a prototype of Argo.

[RHR96a] and [RHR96-b] presented the facilities and architecture of Argo, a domain-oriented design environment for software architecture. Argo's architecture is motivated by the desire to achieve reuse and extensibility of the design environment. It separates domain-neutral code from domain-oriented code, which is distributed among intelligent design materials as opposed to being centralized in the design environment. Argo's facilities are motivated by the observed cognitive needs of designers. These facilities extend previous work in design environments to support reflection-in-action, opportunistic design, and comprehension and problem-solving.

Argo differs from other software architecture design environments in that it pays attention to the human, cognitive needs of software architects as much as to the representation and manipulation of the architecture itself. [RR96] emphasized the primary considerations by contrasting the human cognitive design process with the systems-oriented software design process. Human-centered features in Argo focus on the application of critics for feedback, design processes for supporting critics, and multiple architectural perspectives for aiding human designers.

[RHR98] focused on software architecture *critics* in Argo, and how Argo supports decision making by automatically supplying knowledge that is timely and relevant to decisions at hand. Critics can deliver knowledge to architects about the implications of, or alternatives to, a given decision. Usually, critics simply advise of potential errors or areas needing improvement; only the most severe errors are prevented outright, thus allowing the architect to work through invalid intermediate states of the architecture.

[ZKR96] examined improving a software re-architecting technique to be more effective in the presence of partial documentation. Most re-engineering efforts assume there is no documentation, but rather, reverse engineer documentation (such as design or software architecture) from source code, and then forward engineer a new system. In contrast, software tracing technology assumes full documentation of software artifacts and links between related artifacts. In reality, the situation is somewhere in between, and available documentation should be used to produce more accurate reverse engineered documentation and ultimately better re-engineered software systems.

3.2.2.1 Graphical User Interface Development and Runtime Support

All Chiron client and server code was ported to the GNAT Ada95 compiler and all SunAda dependencies were removed. This included all client support libraries and all generator tools and comprises ~65K lines of documented code. All Chiron development support tools were loaded and tested. Sample Chiron applications were generated, compiled, and loaded. A release was built, including a user manual addendum for Ada95 support.

The Chiron server was not able to run under the GNAT compiler available at the time. Client code did run (communicating with a SunAda- compiled server) on an older patched version of the GNAT compiler.

All Chiron client development tools were modified to support full Ada95 source and the client implementation was modified to take advantage of some of the new Ada95 features. Although the Chiron artist design was not significantly modified, support was added for procedure pointers to call-back routines. We looked into improvements in the Chiron "wrapper" technology but found no significant benefits afforded by Ada95 object-oriented features. However, there were promising findings in using classes to restructure Chiron client event definitions, although this would have required a major redesign of the client.

3.2.3 Measurement and Empirical Evaluation

A study of the 15 year evolution of a software system was done in order to characterize system errors and facilitate the development of early lifecycle metrics. Work was also done with the Loral Corporation to develop architecture based software measurement techniques.

3.3 Analysis for High Assurance Software

Developing high-assurance software is a critical emphasis in software development. Accordingly, the Arcadia project placed particular emphasis on developing and integrating analysis technologies as part of its central mission. The emphasis was on supporting analyses of heterogeneous systems, where analysis is performed both on design as well as code, and where

multiple specification formalisms are used. Testing and dependence analyses were also key emphases.

Arcadia was unique in addressing both the problem of scaling up individual analysis and testing techniques to large-scale, team-developed software. Our approaches to analysis and testing for high assurance software were predicated on integration through object management and definition of active processes using Arcadia infrastructure capabilities. The Arcadia platform supports large-scale integration of tools and tool fragments, thereby enabling us to synergistically integrate complementary analysis and testing techniques.

3.3.1 Tool Support

Much of the work in this area was associated with the tools ProDAG, TAOS, and CATS/Pal. ProDAG analyzes program dependences, which represent information flow between program components and as such are the essential semantic relationships determining when one component may affect another's behavior. These relationships are useful for software understanding, testing, debugging, maintenance, reverse and re-engineering. ProDAG provides software developers with these capabilities: data, control and syntactic dependence analysis; detection of dependence anomalies; language independent analysis; graphical representation and manipulation; separating programmatic interfaces to distinct dependences; and dependence-based test adequacy criteria.

TAOS is a toolkit and environment supporting analysis and testing processes. TAOS reduces costs associated with analysis and testing while improving productivity and enables higher assurance of software dependability. It accomplishes these goals by providing developers and testers with formal, automated support for: test artifact development and maintenance; parallel, monitored test execution; formal behavior verification via automated test oracles; test adequacy criteria definition and measurement; and interactive and programmatic test process management. TAOS is further described in [Ric93].

CATS (Concurrency Analysis Tool Suite) is a suite of tools for statically analyzing synchronization properties in concurrent programs. The first version of CATS, CATS/Ada, was described in [YTL+95]. CATS/Ada is designed to satisfy several criteria: it must analyze implementation-level Ada source code and check user-specified conditions associated with program source code; it must be modularized in a fashion that supports flexible composition with other tool components, including integration with a variety of testing and analysis techniques; and its performance and capacity must be sufficient for analysis of real application programs.

CATS/Pal [YY94] includes an Ada design language (PAL) and an analysis tool (Pal) that: translates specifications written in PAL into process graphs, and constructs a hierarchical analysis plan based on scope structure and task communication structures; performs reachability analysis and verifies correct implementations based on various notions of 'implements' relations (bi-simulations and preorders); applies abstraction and reductions automatically or under user direction during hierarchical composition of analysis results to control growth of the state-space; uses user-supplied context assumptions to further prune the state-space, as well as verifying the validity of those assumptions; and can be configured as a single client/server pair (the default configuration) or as multiple servers analyzing sub-problems in parallel on a network of workstations. Process graphs can also be used as test oracles to validate consistency between an Ada program and the verified PAL description.

3.3.2 Specification-based Testing

[CSR95] examined the use of formal specification languages to allow automation of functional test selection based on the specification. In particular, the formal specification language ADL (Assertion Definition Language), a formal specification language based on predicate logic, was examined as the basis for test selection. Such an automated approach should provide better and more consistent functional test coverage, and thus can improve the dependability of the software under test.

Structural testing generally refers to techniques where test cases are designed based on the structure of the implementation and typically cover that structure and exercise the program. [CRS96] discussed Structural Specification-based Testing (SST), which refers to techniques where test cases are designed based on the structure of the specification. SST test cases may be used to exercise the specification as well as to test the implementation. Structural testing based solely on the implementation is usually not adequate because certain types of errors (in particular, errors of omission) cannot, in general, be detected by structural testing. SST is a functional approach to testing that allows the detection of errors of omission in the implementation as well as errors of commission. How SST can be applied using specifications written in ADL, how structural specification-based test selection criteria can be determined from ADL specifications, and ways to generate test cases from ADL specifications were examined.

As software grows more complex, the difficulty of ensuring its behavioral correctness becomes much more difficult. Furthermore, software systems are being placed in safety-critical applications. Effective testing of critical systems has been hampered by the lack of a cost-effective method for deciding the correctness of a program's behavior under test. Typically, test

results are checked manually, which is tedious and error-prone. Using formal specifications to describe the critical system properties and then checking test results against these specifications overcomes these problems. If these *test oracles*, which are mechanisms for determining whether a test passes or fails, are efficient, they can be combined with automatic test generation to cost-effectively automate the testing of large numbers of test cases that more adequately cover the system requirements and structure. [ORD96] presented an algorithm for automatically deriving efficient test oracles from Graphical Interval Logic (GIL), which is a graphical temporal logic that is easier for non-experts to understand than many formal languages. To develop efficient test oracles from GIL, specifications are converted into automata that can be checked in time linear to the length of the program trace. Additionally, the automata can be checked incrementally to identify failures when they occur, which provides the opportunity to obtain useful information about the erroneous state of the system.

[OMal96] described the Execution-time Analysis for Specification-based Oracle Failures (EASOF) model for specification-based testing. This model addresses verifying behavioral correctness of test executions in a generic fashion as well as specification-based test selection. Thus, EASOF uses formal specifications to support testing early in the development lifecycle, before the design or implementation have been started. The EASOF model was validated by analyzing the trade-offs within the model, instantiating the model with two diverse specification languages, and applying the instantiations to two examples widely used to validate testing and analysis techniques. Trade-offs in the model include accuracy versus execution time, memory versus speed, and pre-computation versus demand. A language for representation mappings that is flexible enough to support many different types of specifications, without requiring a second level of specification, was also presented.

[RW96] made the argument that with the advent of explicitly specified software architectures, testing can be done effectively at the architecture level. A software architecture specification provides a solid foundation for developing a plan for testing at this level. Several architecture-based test criteria based on the Chemical Abstract Machine model of software architecture were proposed. An architectural test plan, developed by applying selected of these criteria, can be used to assess the architecture itself or to test the implementation's conformance to the architecture. This facilitates detecting defects earlier in the software lifecycle, enables leveraging software testing costs across multiple systems developed from the same architecture, and also leverages the effort put into developing a software architecture.

[Rey96] proposed that a solution to the problem of domain theory maintenance is to apply Knowledge-Based Software Engineering (KBSE) techniques at the meta- level: Domain theory experts construct formal specifications of domain theories using domain modeling languages suitable to the domain of domain theory construction. Automated deductive synthesis techniques would then be used to generate domain theories supporting efficient deductive synthesis. A theoretical foundation for Meta-Amphion's specification acquisition subsystem was presented. It is believed that this foundation will enable domain experts, without training in

KBSE, to maintain domain theories via intuitive domain modeling languages with which they are already familiar. This solution would be made possible in large part by constructing knowledge bases and domain theories from algebraic specifications, specification morphisms, and colimits.

3.3.3 Concurrency Analysis

Concurrent systems are inherently more difficult to analyze and visualize that sequential programs. The difficulty in producing correct concurrent programs is mirrored in maintenance as difficulty in extracting a correct high-level model of task interactions and predicting the effect of modification to portions of a system. [ARY94] advocated a methodology that combines static analysis of an abstract model with dynamic analysis of source code. While the abstract model is amenable to exhaustive analysis, dynamic analysis is capable of checking richer classes of specifications, and moreover provides a check on the correctness of simplifications and assumptions inherent in abstract models. The approach was illustrated by combining two tools, the CATS/Pal system for compositional reachability analysis, and the FORESEE analysis tool for temporal analysis of run-time traces.

The abstraction capabilities of process algebra provide an effective means to control state explosion in automated state-space analysis, but only if a design is carefully modularized to encapsulate details of behavior. [YY94] reported on experience in modifying an existing design (a remote temperature sensor system) to make it more amenable to compositional, or hierarchical, analysis. Redesign for analysis was effective in improving the design in other ways as well: Flaws uncovered in the analysis (and present in the original design) were easy to understand and correct because of the increased understandability of the revised design. This also suggests that these design flaws might have been avoided, and the design generally improved, had "design for analysis" been applied from the start.

The key features and relative advantages of different graph models for analyzing concurrent programs were described in [PTY95]. Software applied in critical applications requires high degrees of assurance of various functional properties. Many critical applications involve task-level concurrency. For such applications, guarantees of freedom from deadlock and race conditions are essential. Reachability is no panacea, but it can be a useful, practical technique for assessing such properties. Unlike most common static analyses, reachability analysis can be spectacularly expensive if applied without discipline. Thus, issues of efficiency and scalability were the focus of this work. The representation of internal choice, static versus dynamic matching, and encapsulation (modular or compositional analysis) were identified as important basic differences among models. A model used for reachability analysis should support modularity, and preferably in a manner that permits incremental hierarchical

composition of analysis results, with reductions of intermediate results to battle combinatorial explosion.

A concurrent program fault can propagate both within a single task (thread of control) and between tasks, making fault localization difficult. [XY95] proposed a two-dimensional approach and supporting techniques for integrating analysis of task interactions, inter- and intra-task dataflow, and conventional sequential debugging of individual tasks. The term *augmented concurrent dynamic slice*, a variant of dynamic slicing, which balances the cost and accuracy for concurrent program debugging and permits adjustment of that balance and focus on small parts of large complex systems, was defined. The design and implementation of prototype tools which add concurrent slicing capability to an existing debugger were also described.

As software evolves from early architectural sketches to final code, a variety of representations are appropriate. At most stages, different portions of a system evolve at different rates and consequently in different representations. State-space analysis techniques (reachability analysis, model checking, simulation, etc.) have been developed for several representations of concurrent systems, but each tool or technique has typically been targeted to a single design or program notation. [PY96] described an approach to constructing space analysis tools using a core set of basic representations and components. Such an approach differs from translation to a common formalism. Not every supported design formalism need be mapped to a single internal form that completely captures the original semantics. Rather, a shared "inframodel" represents only the essential information for interpretation by tool components that can be customized to reflect the semantics of each formalism, resulting in more natural and compact internal representations, and more efficient analysis, than a purely translational approach.

State-space analysis techniques have been developed for several representations of concurrent systems, but each tool or technique has typically been targeted to a single design or program notation. [PY97] described an approach to constructing multi-formalism state-space analysis tools for heterogeneous system descriptions, using a shared "inframodel" that represents only the essential information for interpretation by tool components that can be customized to reflect the semantics of each formalism. The (operational) semantics of each formalism, as well as interactions between components described in different formalisms, were described separately through rules governing enabling, matching, and firing of transitions. This results in more natural and compact internal representations, and more efficient analysis than a purely translational approach. A prototype tool for generating custom, interoperable tools for analysis of concurrent systems was upgraded to use this more flexible and general approach for description of the semantics of design notations. The new approach interprets a flexible language for describing the operational semantics of each notation and the ways in which parts of heterogeneous systems

synchronize and communicate. The rule-based approach described accommodates a wider range of state-transition formalisms than the execution semantics of the inframodel controlled through a limited set of parameters, described in [PY96].

3.3.4 Test Process and Evaluation

Software Test Environments (STEs) provide a means of automating the test process and integrating testing tools to support required testing capabilities across the test process. STEs may support test planning, test management, test failure analysis, test development, and test execution. The software architecture of an STE describes the allocation of the environment's functions to specific implementation structures. An STE's architecture can facilitate or impede modifications such as changes to processing algorithms, data representation, or functionality. Performance and reusability are also subject to architecturally imposed constraints. Evaluation of an STE's architecture can provide insight into modifiability, extensibility, portability and reusability of the STE. [ER96b] proposed a reference architecture for STEs. The reference architecture's analytical value is demonstrated by using SAAM (Software Architectural Analysis Method) to compare three software test environments: PROTest II (Prolog Test Environment, version II), TAOS (Testing and Analysis with Oracle Support), and CITE (CONVEX Integrated Test Environment).

Software testing expends as much as 50% of software development costs and comprises up to 50% of development time, yet most of the research in software architectures marginalizes the importance of testing in relation to architectural design decisions. [ER96c] stated the belief that testing has a dual role in software architectures: defect prevention and defect detection. How effectively testing fulfills its roles is dependent on three criteria: matching of software architecture to test strategy, the ability to detect certain fault types given a specific test strategy, and the interaction of software architecture, test strategies, and fault detection. Salient issues in achieving defect prevention and detection under these three criteria, and examples of interdependencies between software testing and software architectures, were presented.

Software process improvement is necessary to continually improve the quality of software and organizations. Process improvement, however, requires process measurement. This is an inherently difficult task as much of the software process is not observable until the final stages of development. Software process models are used to define, characterize, measure, and control the software process. [ER97a] introduced a software process model to capture and store process information with respect to the software test process. This approach leverages the cost of software testing by also applying test results in a process measurement and control framework.

Organizations require that diverse groups of (non-)technical personnel engage in process improvement programs under such rubrics as continuous measurable improvement, statistical process improvement, and total quality management. Yet, understandable descriptions of valid software measures are not available to these groups. Graphical measurement models based on a measurement theoretic approach provide intuitively understandable models of software measures. [ER96a] demonstrated the analytical value of these models by comparing three measures of testability as published by McCabe, Bache, Mullerburg, and Binder. The constructive value of graphical measurement models was demonstrated by incorporating findings in a proposed measure of testability. A measurement theoretic model of testability, developed and discussed further in [ER97b], is based on integration testing strategies with respect to architectural styles, thus making it applicable to a level of testing most impacted by architectural choices.

Software processes are executed for a purpose: to satisfy a set of process requirements and to meet process constraints. [TRK96] demonstrated that there is a critical set of process constraints, often considered only implicitly or even ignored, that are derived externally from social expectations. An approach to determining this set of process constraints and a basic method for their consideration during safety-critical testing process design was suggested.

3.3.5 Uncertainty in Software Testing

[ZRK96] made the observation that uncertainty permeates software development, but is rarely captured explicitly in software models. The Uncertainty Principle in Software Engineering (UPSE), which states that uncertainty is inherent and inevitable in software development processes and products, was presented. Uncertainty in software testing was explored in detail, including its presence in test planning, test enactment, error tracing, and quality estimation. A technique for modeling this uncertainty, called Bayesian belief networks, was presented and its application to software systems was justified.

[ZR97b] made the claim that software development would do well by explicit modeling of its uncertainties using existing uncertainty modeling techniques. The value of this was demonstrated with the Maxim of Uncertainty in Software Engineering (MUSE), followed by a detailed presentation of uncertainty in software testing. Bayesian Belief Networks were proposed to be used to model software testing uncertainties. The use of Bayesian networks to confirm beliefs in the validity of software artifacts and relations in an elevator control system was demonstrated. A prototype implementation was described that allows for such "software belief networks" to be defined and updated. Relevant issues, concerns, and future prospects for modeling software uncertainties were presented.

The lifetime of many software systems is surprisingly long, often far exceeding initial plans and expectations. During development and maintenance of long-lived software, requirements are analyzed and specified, designs and code modules are developed, testing is planned and code is tested many times. Consequently, developers and managers frequently lose or gain confidence in software artifacts, especially when existing uncertainties are relieved or when new uncertainties are encountered. Fluctuations in developers' confidences may in turn affect process actions or decisions, for instance determining the impact of change, the need for regression testing, or when to stop testing. [ZR97c] presented an approach that allows developers' confidences or "beliefs" regarding software components to be modeled and updated directly. This approach is part of an overall strategy that calls for explicit modeling of software engineering uncertainties using Bayesian belief networks. Several types of software uncertainty and how they may be modeled were presented, as well as discussion of experiences in using Bayesian network models for an existing system. Finally, a design and implementation of a Java program that allows software systems and associated beliefs to be modeled explicitly was presented.

Several aspects of computing technology are growing at astounding rates, driving the entire field forward at an accelerated pace. These include evolution of hardware, and the growth of the Internet and the World Wide Web. As a result of hardware availability, including increased speeds and reduced prices, and increasingly ambitious software systems, users by-and-large are no longer burdened with too little information, but rather with handling and sifting through too much information. User disorientation in large and complex spaces is but one of many human anxieties and uncertainties in the Information Age. [ZR97d] described an approach that begins to address some of these up and coming concerns. By promoting explicit modeling of uncertainties, this approach relieves anxieties and, ultimately, leads to reduced overload and improved orientation in large information spaces.

3.4 Software Understanding

Reusing and reengineering software will help catalyze software developers to achieve large improvements in software quality, cost and duration of usefulness. Effective reuse and reengineering enable gains in quality by incorporating components whose reliability has already been established, gains in productivity by avoiding redevelopment, and gains in flexibility by shortening the time required to make changes. Deep understandings of software-intensive systems accelerate effective reuse and reengineering as well as provide a foundation for new and improved systems and underlying technologies.

Research in software understanding was conducted in three related areas: (a) process for guiding software understanding, reengineering, and reuse: (b) analysis techniques for enabling

software understanding, reengineering, and reuse: and (c) infrastructure for facilitating understanding, reengineering, and reuse.

Revealing insights into new methods, techniques, and tools for software processes, product architectures, and systematic improvements can be achieved through studies of best practice software development at leading companies and laboratories. A collaborative effort with Microsoft was conducted to investigate their software development approaches and to reveal these insights. The focus of the work was on characterizing Microsoft's specification, design, implementation, and testing approaches, and understanding the trade-offs among issues such as feature volatility, up-front design, reuse, automated test support, and down-stream maintainability in the development of large-scale systems. These investigations yielded best practice insights as well as prototypes for facilitating the rapid synchronization and evolution of large systems.

[SR95] described a study of the understanding of a large software system structure and its relationship to error-proneness by analyzing the system's interconnections. A key feature of the study was that it was based on design-level information, which was used to show that interconnectivity analysis can be used to provide valuable feedback early in the design process. A general model of connectivity was defined, and a specific instance of the model was applied to a large system. Message-based interconnection criteria were used for computing the degree of interconnectivity among components and for deriving visualizations of the system's structure. The results suggest that interconnectivity analysis can be used successfully to locate error-prone components; this type of analysis is scalable; it is applicable during multiple development phases; it can be based on multiple interconnection criteria; it provides a means for visualizing software system structure from many different views; it can be automated; and it is aided by the use of a structured software architecture formalism.

4 Conclusions

Arcadia achieved insights and developed models and tools while striving to attain its goal of providing solutions to the need for cost-effective evolution of complex software systems. While achieving this goal is a never ending task in a fast-moving technological economy, we made significant progress toward improving the state of the art through explorations, experimentations, and development of tools for communication, support of development teams, analysis, testing, measurement, visualization, and software architectures. Frequently, new relationships between these areas were discovered, inspiring new, unforeseen synergies. Some of these results impact limited domains, while others impact very large ones, such as the World Wide Web, and will affect large populations.

5 Technical Transition

This work was performed by two universities, and was fundamentally research in nature. Nevertheless substantial effort went into producing prototype products that are usable by advanced development organizations, and support is provided for key systems that we distribute.

The best possible impact of the work is to have the ideas which we have created and demonstrated in the prototype systems used widely in systems built by others. Such use would demonstrate that the ideas are in fact general, not idiosyncratic, and described in such a way that they are not bound to a single prototypical artifact on a particular platform.

All of our projects strove to integrate existing COTS and ROTS tools with newly developed prototypes. This approach facilitated transition and reduced transition costs, by integrating new capabilities with familiar ones and into familiar environments, and added value to existing tools.

Ideas are effectively communicated by good prototypes, among other ways. The World-Wide-Web protocols, for instance, are concepts, effectively demonstrated in good prototypes, and taken and implemented differently by other developers. Thus, part of our work was part of the WWW effort, with world-wide impact. Our major prototyping efforts, in user interface systems and architectures and heterogeneous hypertext are also available freely on the Web, and together have been acquired by over 100 sites. Earlier software products from this project, focusing on language processing, have been installed and used in several hundred sites. While such usage is perhaps hardest to effectively assess, it is also probably the most effective in the long-run.

Effective communication of ideas can also depend critically on one-on-one relations. Accordingly we engaged in such relationships with Northrop Grumman's B-2 division (UI architectures and hyperware), Hughes (testing and analysis technology), Loral Federal Systems Division (hypermedia technology), Boeing (testing and analysis technology), TASC (testing and analysis technology), USC's ISI Dasher project (distributed, team-oriented software development), Columbia University OzWeb project (hyperware technology), CoGenTex, Inc. (hyperware), Lockheed Martin C2 Integration Systems (hyperware), and MCC (testing and analysis), among others.

Primary customers were advanced development groups in both commercial and military organizations, as well as other researchers. Results have been exploited on the basis of ideas and on direct use of the technology prototypes. In the analysis and testing of a reusable component library for radar signal processing systems, Hughes Radar and Communication Systems attests that the TAOS software testing environment saved them 25% in testing-related costs, improved productivity, and increased software quality; they are using the system on a follow-on project in

which they are reengineering F/A-18 flight software. The Analytical Sciences Corporation (TASC) used the ProDAG program dependence analysis system in the development of a Avionics V&V System for Wright-Patterson AFB with great success; they are integrating the TAOS environment (including the next version of the ProDAG system) into a follow-on AV&V environment for Wright Labs.

The technologies that were the subject of this contract fundamentally provide benefits that are qualitatively different from other contemporary approaches, and are not readily the subject of quantitative analyses. Thus the impact has to be assessed through evidence of community interest, use, respect, and testimonial.

5.1 Key Technologies Exported

Distributed, Team-Oriented Software Development: Endeavors WWW-based process automation tool.

Hyperware: Chimera heterogeneous hypermedia system, hypertext generation from program source code, src2fm and src2www (tools to take source code as input and format for FrameMaker and html), world-wide-web tools and protocols, including libwww-ada95.

Architectures and User Interfaces: Chiron-1 User interface development system, Argo/UML cognitive support for object-oriented design, UCI Graph Editing Framework (GEF).

Analysis and Testing: TAOS (Testing and Analysis with Oracle Support) software test environment and ProDAG (Program Dependence Analysis Graph)

5.2 System name, purpose, environment requirements, and point of contact

Distributed, Team-Oriented Software Development

Endeavors: WWW-based process automation tool

> Endeavors is an open, distributed, extensible process execution environment. It is designed to improve coordination and managerial control of development teams by allowing flexible definition, modeling, and execution of typical workflow applications.

Endeavors has customizable distribution and decentralization policies which provide support for transparently distributed people, artifacts, process objects, and execution behavior (handlers). In addition, Endeavors processes, as well as the means to visualize, edit, and execute them, are easily downloaded using current and evolving world wide web (WWW) protocols.

Endeavors allows bi-directional communication between its internal objects and external tools, objects, and services through its open interfaces across all levels of the architecture. Implementation of object behaviors in multiple languages is supported, allowing them to be described in whatever programming language is most suitable for integration.

Endeavors requires low cost and effort to install across all software platforms. All process objects are file (ascii) based allowing greater portability across different machine architectures. Components of the system, including user interfaces, interpreter, and editing tools, may be down loaded as needed, and no explicit system installation is required to view and execute a workflow-style process.

Endeavors is implemented as a layered virtual machines architecture, and allows object-oriented extension of the architecture, interfaces, and data formats at each layer. Because processes, objects, tool integrations, and policies can be used across platforms, processes may be adapted or evolved through embedding and composition of process fragments using cutting, copying, and pasting of activity representations.

Endeavors allows dynamic changing of object fields and methods, the ability to dynamically change the object behaviors at runtime, and late-binding of resources needed to execute a workflow process. Process interpreters are dynamically created as needed.

Environment Requirements: Endeavors is 100% Java compatible and was even developed across several platforms. Initial experiments show that Endeavors objects, tools, and services are platform cooperative across all major platforms but has only been tested under Windows NT/95, Solaris 2.x, and Irix 6.3.

POC: Richard N. Taylor, taylor@ics.uci.edu, endeavors@ics.uci.edu

Hyperware

Chimera: Heterogeneous hypermedia system

Chimera is an open, serverized, hypermedia system that supports n-ary links between heterogeneous tools and applications in a network. Objects manipulated by separate applications can be linked together through Chimera. From the user's standpoint, for example, while working with one object in one application, an anchor on the displayed object may be selected, causing another application to start up, displaying a related (linked) object. Chimera comes with bindings to C, Ada,

and Java; bindings to several popular tools have been constructed. Chimera makes no assumptions or demands regarding user interface system employed or how or where objects are stored.

Environment Requirements: SunOS version 4.1 or higher, SunAda version 1.1, C compiler, header files in X window system, version 11, revision level 5. Also, Q v3.2 Arpc v402.3. Binary distributions also available.

POC: Yuzo Kanomata, yuzok@ics.uci.edu, or Richard N. Taylor, taylor@ics.uci.edu

WebSoft: Software for Webmasters

The available software includes MOMspider: A web-roaming robot that specializes in the maintenance of distributed hypertext infostructures (i.e. wide-area webs); caching.html: The Conditional GET proposal for HTTP (also known as If-modified-since). (This is not software); libwww-perl: A library of Perl4 packages which provides a simple and consistent programming interface to the World-Wide Web. This library is being developed as a collaborative effort to assist the further development of useful WWW clients and tools; rewrite_history.txt: A simple perl program for cleaning the garbage out of a mosaic-global-history file; wwwstat: A program for processing a sequence of NCSA httpd access_log files and printing a log summary in HTML format suitable for publishing on your web.

Environment Requirements: SunOS 4.1 or higher

POC: Roy Fielding, fielding@ics.uci.edu

Architectures and User Interfaces

Argo/UML: Providing Cognitive Support for Object-Oriented Design

Argo/UML is a domain-oriented design environment that provides cognitive support of object-oriented design. Argo/UML provides some of the same automation features of a commercial CASE tool, but it focuses on features that support the cognitive needs of designers. These cognitive needs are described by three cognitive theories: reflection-in-action, opportunistic design, and comprehension and problem solving.

Argo/UML is based directly on the UML 1.1 specification. In fact, a large part of Argo/UML was generated automatically from the UML specification. Argo/UML is (to the best of our knowledge) the only tool that implements the UML meta-model exactly as specified.

Environment Requirements: Java 1.1 virtual machine, Sun's Java Foundation Classes (JFC) 1.1, including Swing version 1.02

POC: Jason Robbins, jrobbins@ics.uci.edu, David Redmiles, redmiles@ics.uci.edu, David Hilbert, dhilbert@ics.uci.edu, Adam Bonner, abonner@ics.uci.edu

GEF: Graph Editing Framework

The Graph Editing Framework is a library of Java(tm) classes that make it easier to develop new applications that involve diagram editing (ala MacDraw(tm)) and connected graph editing (something like Visio(tm)). GEF can be used to for both applications and applets, although, some features (e.g., saving files) are only available in applications.

Environment Requirements: JDK 1.1.5, Swing version 1.0.1 (both from www.javasoft.com)

POC: Jason Robbins, jrobbins@ics.uci.edu, David Redmiles, redmiles@ics.uci.edu

C2: Architectural style for GUI Software

C2 is an architectural style and supporting toolset designed to support the particular needs of applications that have a graphical user interface aspect, but has the potential for supporting other types of applications. C2 allows software components to be written in different languages and C2-style components can be readily reused and/or substituted with other components in an architecture. Components can be of various granularities, promoting scalability. They may execute in a distributed, heterogeneous environment. There is no assumption of shared address space among components and each component may have its own thread(s) of control. Multiple users may be simultaneously interacting with applications. Multiple dialogs may be active and described in different formalisms. Multiple tool kits and media types may be employed. Architectures may be changed dynamically. Multiple implementation architectures may realize a single conceptual architecture.

Environment Requirements: Java.

POC: Richard N. Taylor, taylor@ics.uci.edu

Chiron-1: User Interface Development System

The Chiron-1 system provides tools for assisting in the development of graphical user interfaces and provides a run-time system for managing and supporting their

dynamic behavior. The objective of the Chiron-1 system is to reduce long-term costs associated with developing and maintaining graphical user interface (GUI) software. It achieves this objective by providing key interface layers which are resilient to change. In particular, Chiron-1 strongly separates an application from its user interface code, as well as separating the user interface code from the underlying toolkit substrates. Chiron-1 supports the construction of GUIs which provide multiple coordinated views of application objects and allows flexible restructuring of the configuration of those views. Chiron-1 supports a concurrent model of control. While the Chiron-1 architecture supports heterogeneous, multi-lingual systems, the development tools which are part of this release only support clients (applications) written in Ada.

Chiron-1 provides separation of concerns at two levels: between the application and the user interface software, and between the user interface software and the underlying windowing system and toolkit.

Chiron-1's client-server architecture provides flexibility in terms of windowing systems and toolkits, application languages, and process interconnection topology.

Chiron-1 supports concurrent applications and a concurrent model of control. Unlike most user interface architectures, Chiron-1 does not impose a callback structure on applications. The application, the user interface, and the Chiron-1 server all run in parallel.

Environment Requirements: SunOS 4.1/SunAda version 1.1 or Solaris 5.4/SunAda version 2.1 or RS6000/Verdix VADS v6.2; gcc compiler v2.5.8; X Window System, version 11 revision level 5; XView version 3.0 and/or Motif version 1.2; Q v3.2, an Ada-C interprocess communication support utility and Arpc v402.3, an extension to Sun RPC/XDR 4.0, both developed by the Department of Computer Science, University of Colorado, Boulder. (included with the Chiron-1 distribution)

POC: Richard N. Taylor, taylor@ics.uci.edu

GLAD: Generic Layout for Directed Graphs

GLAD is an Ada generic package providing static or dynamic layout facilities for directed graphs. Given a description of a graph, it provides coordinate sets for the positioning of elements of a graph on an arbitrary cartesian coordinate plane. Layout coordinates can be provided for either a complete graph description or incremental, dynamic updates to a graph structure. This allows dynamic graph editing, including addition and deletion of any graph element. Provisions are available to provide minimum disturbance of a current layout description when edit updates to the graph structure are made. The system does not impose any particular user interface on the instantiator, and does not assume anything about the actual display of a graph, except the location of graph elements, which it provides. A

facility is provided for balancing the layout structure. Facilities are also available to reduce edge crossings if order of graph nodes is not important.

Environment Requirements: Ada compiler.

POC: Craig Snider, snider@ics.uci.edu, or Debra J. Richardson, djr@ics.uci.edu

Analysis for High Assurance Software

ProDAG: Program Dependence Analysis Graph System

ProDAG analyzes program dependencies, which represent information flow between program components and as such are the essential semantic relationships determining when one component may affect another's behavior. These relationships are useful for software understanding, testing, debugging, maintenance, reverse and re-engineering. ProDAG provides software developers with these capabilities: data, control, and syntactic dependence analysis; detection of dependence anomalies; language independent analysis; graphical representation and manipulation; separate programmatic interfaces to distinct dependencies; dependence-based test adequacy criteria.

Environment Requirements: SunOS version 4.1 and SunAda 1.0, or Solaris 5.4 and SunAda 2.1, PLEIADES 2.4, LPT version 2, Chiron-1

POC: Debra J. Richardson, djr@ics.uci.edu, or Craig Snider, snider@ics.uci.edu

TAOS: Testing with Analysis and Oracle Support

TAOS is a toolkit and environment supporting analysis and testing processes. TAOS reduces costs associated with analysis and testing while improving productivity and enables higher assurance of software dependability. It accomplishes these goals by providing developers and testers with formal, automated support for: test artifact development and maintenance; parallel, monitored test execution; formal behavior verification via automated test oracles; test adequacy criteria definition and measurement; interactive and programmatic test process management.

Environment Requirements: SunOS version 4.1 and SunAda 1.0, or Solaris 5.4 and SunAda 2.1, PLEIADES 2.4, Chiron-1

POC: Debra J. Richardson, djr@ics.uci.edu, or Craig Snider, snider@ics.uci.edu

CATS/Pal: Synchronization Analysis

CATS/Pal is a tool for statically analyzing synchronization properties in concurrent programs. It is a prototype of task parceling and composition techniques based on

process algebra that will later be incorporated into the CATS tool suite. It includes an Ada design language (PAL) and an analysis tool (Pal) that: translates specifications written in PAL into process graphs, and constructs a hierarchical analysis plan based on scope structure and task communication structures; performs reachability analysis and verifies correct implementations based on various notions of "implements" relations (bi-simulations and preorders); applies abstraction and reductions automatically or under user direction during hierarchical composition of analysis results to control growth of the state-space; uses user-supplied context assumptions to further prune the state-space, as well as verifying the validity of those assumptions; and can be configured as a single client/server pair (the default configuration) or as multiple servers analyzing sub-problems in parallel on a network of workstations. Process graphs can also be used as test oracles to validate consistency between an Ada program and the verified PAL description.

Environment Requirements: Unix workstation or personal computer with at least Current version requires Linux X86; versions for BSD and SysV variants are also available. 32M main memory, 64M preferred. Requires gcc (C compiler) and Gnu lisp previously known as akcl), which are free software available elsewhere. Additional software, all available at no charge, is required for graphical interfaces. Depending on which graphical features are desired, requirements will include some combination of: X11 window system, Tk/TCL scripting language, AT&T GraphViz graph visualization tools, Ghostscript.

POC: Michal Young, michal@cs.uoregon.edu

Utilities

Plumber

Developing large systems based on many levels of abstraction and ADT's has lead to programs that utilize large amounts of virtual memory and swap space. Without tool support, the only method for finding the source of memory leaks is code inspections. Plumber was written to help identify the memory leaks in Arcadia software. It has been applied to several of the Arcadia tools, such as Chimera, Chiron and ProDAG (and indirectly to Pleiades and Q). Plumber was designed to be used with Ada with or without C, but could be used with only C.

Plumber keeps additional bookkeeping information with each block of memory that is allocated using NEW in Ada or malloc/realloc/calloc/memalign in C. When the block is freed via UNCHECKED_DEALLOCATION in Ada or free in C, its information is removed from the list. At program termination any unfreed blocks are reported in a log file along with the call stack from when they were created via NEW or malloc.

Plumber only requires relinking the application with a new library. The code will run slower than normal, but when the program terminates it will produce a log file

with a list of all the unfreed blocks of memory and where they were created with NEW.

Environment Requirements: SunOS 4.1, SunAda1.1, gcc

POC: Debra J. Richardson, djr@ics.uci.edu

src2fm/src2www

src2fm and src2www take source code as input, and produce nicely formatted listings in FrameMaker and html, respectively. src2fm can be used to prepare small portions of source code for inclusion in presentations, manuals, or technical papers, or it can be used to prepare listings of whole programs, with hypertext indices. src2www produces an html version of each source file, optionally including embedded diagrams, with alphabetical indexes of declarations, within each file and for a set of files. In future src2fm will be a link generator for the Chimera heterogeneous hypertext system, and will provide views and navigation specialized to software analysis and maintenance tasks. Currently Ada83, Ada95, ANSI C, Csh scripts, and some simple C++ are supported.

Environment Requirements: Requires workstation or personal computer running Unix. Full source code is available, optionally compiled for a Sun SparcStation. src2fm requires the Unix version of FrameMaker version 4 or greater. src2www runs on the Unix operating system; generated documents can also be served from VMS systems. Building the binaries requires Gnu make, flex and gcc from FSF. The scripts require an awk interpreter (which may be called nawk, v7awk, or gawk depending on Unix system vendor.)

POC: Michal Young, michal@cs.uoregon.edu

6 WWW Home Page

http://www.ics.uci.edu/Arcadia/atUCI.html

7 Publications

[AF95] Mark S. Ackerman and Roy T. Fielding. Collection maintenance in the digital library. Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries (Digital Libraries Ô95), Austin, Texas, June 11-13, 1995.

[ATW94] Kenneth M. Anderson, Richard N. Taylor, and E. James Whitehead Jr. Chimera: Hypertext for heterogeneous software environments. Proceedings of the 1994 ACM Conference on Hypertext, Edinburgh, Scotland, September, 1994, pp. 94-107. <http://www.ics.uci.edu/pub/chimera/overview/papers/ECHT94/>.

[And96a] Kenneth M. Anderson. Extending user-interface toolkits with hypermedia functionality. Proceedings of the 30th Hawaii International Conference on System Sciences, Wailea, Hawaii, USA, January, 1997. <http://www.ics.uci.edu/pub/chimera/overview/papers/HICSS30/>.

[And96b] Kenneth M. Anderson. Integrating open hypermedia systems with the World Wide Web. Proceedings of the Eighth ACM Conference on Hypertext (Hypertext '97), Southampton, UK, April 6-11, 1997, pp. 157-166. <http://www.ics.uci.edu/pub/chimera/overview/papers/HT97a/>.

[And96c] Kenneth M. Anderson. Providing automatic support for extra-application hypertext functionality. Proceedings of the Second International Workshop on Incorporating Hypertext Functionality Into Software Systems. Part of the 1996 ACM Conference on Hypertext. Washington D.C., USA, March, 1996. <http://www.ics.uci.edu/pub/kanderso/htf2/autoehtf.html>.

[ARY94] Frank D. Anger, Rita R. Rodriguez, and Michal Young. Combining static and dynamic analysis of concurrent programs. Proceedings of the International Conference on Software Maintenance (ICSM Ô94), Victoria, British Columbia, Canada, September 19-23, 1994, pp. 89-98. <http://www.cs.uoregon.edu/homes/young/papers/papers.html>.

[BFF96] T. Berners-Lee, R. Fielding, H. Frystyk. Hypertext Transfer Protocol Ñ HTTP/1.0. Internet RFC 1945, MIT/LCS, UC Irvine, February 1996. <http://www.ics.uci.edu/pub/ietf/http/rfc1945.html>.

[Bol94] Gregory Alan Bolcer. User interface design assistance for large-scale software development. Proceedings of the 9th Knowledge Based Software Engineering Conference (KBSE '94), Monterey, CA. September 1994. <http://www.ics.uci.edu/~gbolcer/uida/ps.Z>.

[Bol95] Gregory Alan Bolcer. User interface design assistance for large-scale software development. Journal of Automated Software Engineering. Vol. 2, No. 3, September, 1995. <http://www.ics.uci.edu/~gbolcer/asefmt.ps.Z>.

[BT96] Gregory Alan Bolcer and Richard N. Taylor. Endeavors: a process system integration infrastructure. Proceedings of the 4th International Conference on Software Process (ICSP4), Brighton, U.K. December 2-6, 1996. <http://www.ics.uci.edu/Arcadia/Endeavors/EndeavorsPaper.ps.Z>.

[CRS96] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. Proceedings of the International Symposium on Software Testing (ISSTA '96), San Diego, CA, January 8-10, 1996.

[CSR95] Juei Chang, Sriram Sankar, and Debra J. Richardson. Automated test selection from adl specifications. Proceedings of the California Software Symposium (CSS '95), Irvine, CA, March 30, 1995, pp. 27-40. <url: http://www/ics/uci/edu/~djr/test-adl-css.html>.

[CS95] Michael Cusumano and Richard Selby. Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People. Simon and Schuster, October, 1995.

[ER96a] Nancy Eickelmann and Debra J. Richardson. An evaluation of measurement theoretic models on software testability. Unpublished report, September 1996.

[ER96b] Nancy Eickelmann and Debra J. Richardson. An evaluation of software test environment architectures. Proceedings of the 18th International Conference of Software Engineering (ICSE 18), Berlin, Germany, March 25-29, 1996, pp. 353-364.

[ER97a] Nancy Eickelmann and Debra J. Richardson. Leveraging the cost of software testing with measurable process improvement. Proceedings of the Computing in Engineering Conference, ETCE-ASME Ô97, Houston, TX, January 1997.

[ER97b] Nancy Eickelmann and Debra J. Richardson. A software testability measure for architecture-based test processes. Unpublished report, January 1997.

[ER96c] Nancy Eickelmann and Debra J. Richardson. What makes one software architecture more testable than another? Proceedings of SIGSOFT Ô96 ISAW2-FSE Joint Conference, San Francisco, October 1996.

[Fiel94] Roy T. Fielding. Maintaining Distributed hypertext infostructures: Welcome to MOMspider's web. Computer Networks and ISDN Systems, 27(2), November 1994, also appears in First International World-Wide Web Conference (WWW94). Geneva, Switzerland, May 25-27, 1994. <url: http//www.ics.uci.edu/WebSoft/MOMspider/WWW94/paper_US.ps>.

[Fiel95] Roy T. Fielding. Relative Uniform Resource Locators. Internet Proposed Standard RFC 1808, UC Irvine, June 1995. <url: http://ds.internic.net/rfc/rfc1808.txt>.

[FG+97] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. Internet Proposed Standard RFC 2068. UC Irvine, DEC, MIT, January, 1997.

[FWA+97] R. Fielding, E. James Whitehead Jr., Kenneth M. Anderson, Gregory A. Bolcer, Peyman Oreizy, Richard N. Taylor. Support for the virtual enterprise: web-based development of complex information products. Communications of the ACM, Vol. 41, No. 8, August 1998. <http://www.ics.uci.edu/~fielding/cacm/>.

[KW97] Gail Kaiser, E. James Whitehead. Collaborative work: Distributed authoring and versioning. IEEE Internet Computing, Vol. 1, No. 2, March/April, 1997, pp. 76-77.

[Med96] Nenad Medvidovic. ADLs and dynamic architecture changes. Proceedings of the Second International Software Architecture Workshop (ISAW-2), held in conjunction with SIGSOFT '96, San Francisco, CA, October 14-15, 1996, pp. 24-27.

[Med97] Nenad Medvidovic. A classification and comparison framework for software architecture description languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, February 1997.

[Med95] Nenad Medvidovic. Formal definition of the Chiron-2 software architectural style. Technical Report UCI-ICS-95-24, Department of Information and Computer Science, University of California, Irvine, August 1995.

[MORT96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard Taylor. Using object-oriented typing to support architectural design in the C2 style. *Proceedings of* SIGSOFT '96: The Fourth Symposium on the Foundations of Software Engineering (FSE4), San Francisco, CA, October 16-18, 1996, pp 24-32.

[MOT96] Nenad Medvidovic, Peyman Oreizy, and Richard N. Taylor. Reuse of off-the-shelf components in C2-style architectures. Proceedings of the 1997 Symposium on Software Reusability (SSR '97), Boston, MA, May 17-19, 1997, pp 190-198. Also in Proceedings of the 1997 International Conference on Software Engineering (ICSE '97), Boston, MA, May 17-23, 1997, pp. 692-700.

[MR97] Nenad Medvidovic and David S. Rosenblum. Domains of concern in software architectures and architecture description languages. Proceedings of the USENIX Conference on Domain-Specific Languages, Santa Barbara, CA, October 15-17, 1997, pp. 199-212.

[MT96a] Nenad Medvidovic and Richard N. Taylor. Exploiting architectural style to develop a family of applications. IEE Proceedings Software Engineering, vol. 144, no. 5-6 (October-December 1997), pp. 237-248.

[MT96b] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, September 22-25, 1997, pp. 60-76.

[MT96c] Nenad Medvidovic and Richard N. Taylor. Reuse of off-the-shelf constraint solvers in C2-style architectures. Technical Report UCI-ICS-96-28, Department of Information and Computer Science, University of California, Irvine, July 1996.

[MT96d] Nenad Medvidovic, Richard N. Taylor. Reusing off-the-shelf components to develop a family of applications in the C2 architectural style. Proceedings of the International Workshop on Development and Evolution of Software Architectures for Product Families, Las Navas del Marqués, Ávila, Spain, November 18-19, 1996.

[MTW96] Nenad Medvidovic, Richard N. Taylor and E. James Whitehead, Jr. Formal modeling of software architectures at multiple levels of abstraction. Proceedings of the California Software Symposium (CSS '96), Los Angeles, CA, April 17, 1996, pp. 28-40.

[ORD96] T. Owen O'Malley, Debra J. Richardson, and Laura K. Dillon. Efficient specification-based test oracles for critical systems. Proceedings of the California Software Symposium (CSS '96), Los Angeles, CA, April 17, 1996, pp. 50-58.

[OMal96] Thomas Owen O'Malley. A model of specification-based test oracles. Ph.D. dissertation, Information and Computer Science, University of California, Irvine, December, 1996.

[PTY95] Mauro Pezzè, Richard N. Taylor, and Michal Young. Graph models for reachability analysis of concurrent programs. ACM Transactions on Software Engineering and Methodologies, vol. 4, no. 4, April 1995.

[PY96] Mauro Pezzè and Michal Young. Generation of multi-formalism state-space analysis tools. Proceeding of the International Symposium on Software Testing and Analysis (ISSTA '96), San Diego, CA, January 1996.

[PY97] Mauro Pezzè and Michal Young. Constructing multi-formalism state-space analysis tools. Proceedings of the 1997 International Conference on Software Engineering (ICSE '97), Boston, MA, May 17-23, 1997, pp. 239-249.

[Rey96] Arthur Alexander Reyes. An approach to automatic generation of domain theories from intuitive, semiformal domain models. Proceedings of the California Software Symposium (CSS '96), Los Angeles, CA, April 17, 1996, pp. 41-49.

[Ric93] Debra J. Richardson. TAOS: Testing with analysis and oracle support. Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '94), Seattle, WA, August 17-19, 1994.

[RW96] Debra J. Richardson and Alex L. Wolf. Software testing at the architectural level. Proceedings of the Second International Software Architecture Workshop (ISAW-2), held in conjunction with SIGSOFT '96, San Francisco, CA, October 14-15, 1996.

[RHR96a] Jason E. Robbins, David M. Hilbert, and David F. Redmiles. Argo: A design environment for evolving software architectures. Demonstration paper. Proceedings of the 1997 International Conference on Software Engineering (ICSE '97), Boston, MA, May 17-23, 1997, pp. 600-601.

[RHR96b] Jason E. Robbins, David M. Hilbert, and David Redmiles. Extending design environments to software architecture design. Proceedings of the Eleventh Conference on Knowledge-Based Software Engineering (KBSE '96), Syracuse, NY, September 25-28, 1996. Also in *International Journal of Automated Software Engineering, Special Issue with Selected Papers from KBSE '96.*

[RHR98] Jason E. Robbins, David M. Hilbert, and David F. Redmiles. Software architecture critics in Argo. Proceedings of the 1998 Conference on Intelligent User Interfaces, San Francisco, CA, January 1998.

[RHR96c] Jason E. Robbins, David M. Hilbert, and David Redmiles. Using critics to analyze evolving architectures. Proceedings of the Second International Software Architecture Workshop (ISAW-2), held in conjunction with SIGSOFT '96, San Francisco, CA, October 14-15, 1996.

[RMRR97] Jason E. Robbins, Nenad Medvidovic, David Redmiles, and David S. Rosenblum. Integrating architecture description languages with a standard design method. Arcadia Technical Report UCI-97-35, University of California, Irvine, August 1997.

[RMRR98] Jason E. Robbins, Nenad Medvidovic, David Redmiles, and David S. Rosenblum. Integrating architecture description languages with a standard design method. In *Proceedings of the International Conference on Software Engineering (ICSE '98),* Kyoto, Japan, April 19-25.

[RMR+96] Jason E. Robbins, David Morley, David Redmiles, Vadim Filatov, Dima Kononov. Visual language features supporting human-human and human-computer communication. Proceedings of the 1996 IEEE Symposium on Visual Languages (VL '96), Boulder, CO, September 3-6, 1996.

[RR96] Jason E. Robbins and David F. Redmiles. Software architecture design from the perspective of human cognitive needs. Proceedings of the California Software Symposium (CSS '96), Los Angeles, CA, April 17, 1996, pp. 16-27.

[RRR97] Jason E. Robbins, David F. Redmiles and David S. Rosenblum. Integrating C2 with the Unified Modeling Language. Proceedings of the California Software Symposium (CSS '97), Irvine, CA, November 7, 1996, pp. 11-18.

[RWMT95] Jason E. Robbins, E. James Whitehead Jr., Nenad Medvidovic, and Richard N. Taylor. A software architecture design environment for Chiron-2 style architectures. Arcadia Technical Report UCI-95-01, University of California, Irvine, January 1995.

[SR95] Richard Selby and Ronald Reimer. Interconnectivity analysis for large software systems. Proceedings of the California Software Symposium (CSS '95), Irvine, CA, March 30, 1995, pp. 3-17.

[SVWD97] J.A. Slein, F. Vitali, E.J. Whitehead, Jr., D.G. Durand. Requirements for distributed authoring and versioning on the World Wide Web. Internet-Draft, work-in-progress, September 24, 1997. <ftp://ds.internic.net/internet-drafts/draft-ietf-webdav-requirements-03.txt>.

[TMA+95] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., and Jason E. Robbins. A component and message-based architecture style for GUI software. Proceedings of the Seventeenth International Conference on Software Engineering (ICSE 17). Seattle WA, April 24-28, 1995, pp. 295-304. <ftp: //liege.ics.uci.edu/pub/arcadia/c2/C2-ICSE17.fm.ps.Z>

[TMA+96] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., Jason E. Robbins, Kari Nies, Peyman Oreizy, and Deborah Dubrow. A component- and message-based architectural style for GUI software. IEEE Transactions on Software Engineering, Vol. 22, No. 6, June 1996, pp. 390-406.

[TNB+95] Richard N. Taylor, Kari A. Nies, Gregory Alan Bolcer, Craig A. MacFarlane, and Kenneth M. Anderson. Chiron-1: A software architecture for user interface development, maintenance, and run-time support. ACM Transactions on Computer-Human Interaction, Vol. 2, No. 2, June 1995, pp. 105-144.

[TRK96] Clark Turner, Debra J. Richardson, and John King. Legal sufficiency of testing processes. Proceedings of the 15th International Conference on Computer Safety, Reliability and Security (SAFECOMP '96), Vienna, Austria, October, 1996.

[Whi96a] E. James Whitehead. An architectural model for application integration in open hypermedia environments. Proceedings of the Eighth ACM Conference on Hypertext (Hypertext '97), Southampton, UK, April 6-11, 1997, pp. 1-12.

[Whi96b] E. J. Whitehead, Jr. Distributed authoring on the World Wide Web: Informal working group meeting. World Wide Web Journal, Issue 4, O'Reilly, Fall 1996, pp. 75-77.

[Whi96c] E. James Whitehead, Jr. Final report of the San Mateo meeting of the Working Group on Distributed Authoring on the World Wide Web. Technical Report UCI-ICS-96-32, Department of Information and Computer Science, University of California, Irvine, August 1996.

[Whi96d] E. J. Whitehead, Jr. Requirements on HTTP for distributed content editing. Submitted to the IETF as an Internet Draft, <draft-whitehead-http-distreq-00.txt>, September 1996. Also published in World Wide Web Journal, Issue 4, O'Reilly, Fall 1996, pp. 239-244.

[Whi96e] E. James Whitehead. SCM and hypertext versioning: A compelling duo. Presented at the Sixth International Workshop on Software Configuration Management (SCM6), held in conjunction with the 18th International Conference of Software Engineering (ICSE 18), Berlin, Germany, March 25-26, 1996.

[Whi97] E. James Whitehead. World Wide Web distributed authoring and versioning (WebDAV): An Introduction. ACM StandardView, Vol. 5, No. 1, March, 1997.

[WFA96] James Whitehead, Jr., Roy T. Fielding, and Kenneth M. Anderson. Fusing WWW and link server technology: One approach. Proceedings of the 2nd Workshop on Open Hypermedia Systems (Hypertext '96), Washington, DC, March, 1996, pp. 81-86.

[WRM+95] E. James Whitehead Jr., Jason E. Robbins, Nenad Medvidovic, and Richard N. Taylor. Software architecture: foundation of a software component marketplace. Proceedings of the First International Workshop on Architectures for Software Systems, held in cooperation with ICSE-17, Seattle, WA, April, 1995, pp. 276-282.

[XY95] Lu Xu and Michal Young. Two dimensional concurrent program debugging. Proceedings of the Asian Pacific Software Engineering Conference, Australia, December, 1995.

[YY94] Wei Jen Yeh and Michal Young. Redesigning tasking structures of Ada programs for analysis: A case study. SERC Technical Report TR-148-P. Published in revised form in the Journal of Software Testing, Verification, and Reliability, Vol. 4, December 1994, pp. 223-253.

[YTL+95] Michal Young, Richard N. Taylor, David L. Levine, Kari A. Nies, and Debra A. Brodbeck. A concurrency analysis tool suite for Ada programs: Rational, design, and preliminary experience. ACM Transactions on Software Engineering and Methodology, Vol. 4, No. 1, January 1995, pp. 65-106.

[YT94] Patrick S. Young and Richard N. Taylor. Human-executed operations in the Teamware process programming system. Proceedings of the Ninth International Software Process Workshop, Arlie, VA, October 1994.

[YT95] Patrick Young and Richard N. Taylor. Process programming languages: Issues and approaches. Presented at the ICSE17 Workshop on the Interaction Between Software Engineering and Programming Languages, Seattle, WA, April 1995.

[ZKR96] Hadar Ziv, René Klösch, and Debra J. Richardson. Software re-architecting in the presence of partial documentation. Unpublished report, October 1996.

[ZO94] Hadar Ziv and Leon J. Osterweil. Research issues in the intersection of hypertext and software development environments. Proceedings of the ICSE '94 Workshop on SE-HCI: Joint Research Issues, Sorrento, Italy, May 1994, pp. 268-279.

[ZR97a] Hadar Ziv and Debra J. Richardson. Adding uncertainty to hypertext models of software systems. International Workshop on Incorporating Hypertext Functionality into Software Systems, in conjunction with the Eighth ACM International Hypertext Conference, Southampton, UK, April 1997.

[ZR97b] Hadar Ziv and Debra J. Richardson. Bayesian-network confirmation of software testing uncertainties. Unpublished report, January 1997.

[ZR97c] Hadar Ziv and Debra J. Richardson. Constructing Bayesian-network models of software testing and maintenance Uncertainties. Proceedings of the International Conference on Software Maintenance (ICSM '97), Bari, Italy, September 29-October 3, 1997.

[ZR97d] Hadar Ziv and Debra J. Richardson. Lost and found in software space: A Bayesian approach. MTAC '97: Multimedia Technology and Applications Conference, Irvine, CA, March 1997.

[ZR96] Hadar Ziv and Debra J. Richardson. Tracing configuration-management information in large software spaces. Presented at the Sixth International Workshop on Software Configuration Management (SCM6), held in conjunction with the 18th International Conference of Software Engineering (ICSE 18), Berlin, Germany, March 25-26, 1996.

[ZRK96] Hadar Ziv, Debra J. Richardson, and René Klösch. The Uncertainty Principle in Software Engineering. Unpublished report, May 1996.

*U.S. GOVERNMENT PRINTING OFFICE: 2002-710-038-10209

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.