# Software Architecture

## Many faces, many places, yet a central discipline

Richard N. Taylor

University of California, Irvine

SIGSOFT

SPECIAL INTEREST GROUP ON SOFTWARE ENGINEERING

Ken Anderson, 1997
University of Colorado

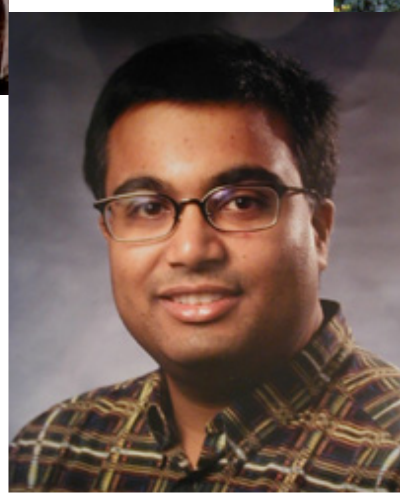
Greg Bolcer, 1998
Keroseneandamatch.com

Neno Medvidovic, 1998
USC

Peyman Oreizy, 2000
Launch21

Jim Whitehead, 2000
UC Santa Cruz

Roy Fielding, 2000
Day Software



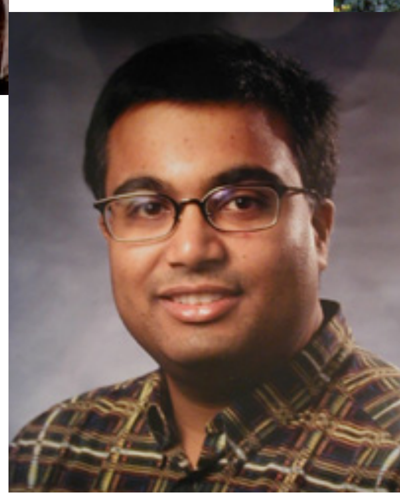
Rohit Khare, 2003
Angstro.com

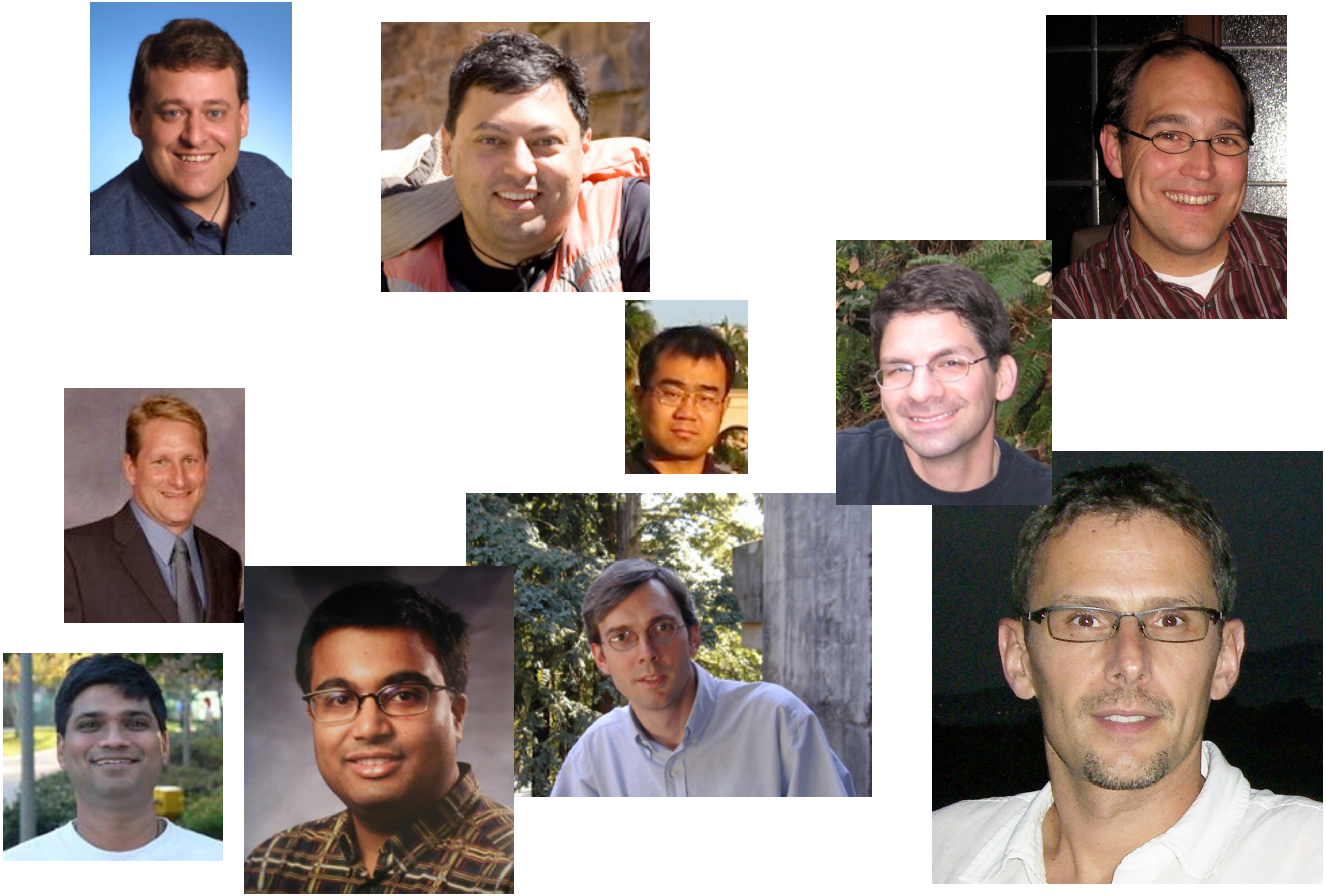
Peter Kammer, 20
Google, Inc.



Jie Ren, 2006
Google, Inc.



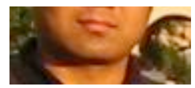
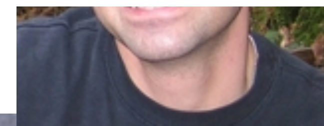


Girish Suryanarayana, 2007
Siemens

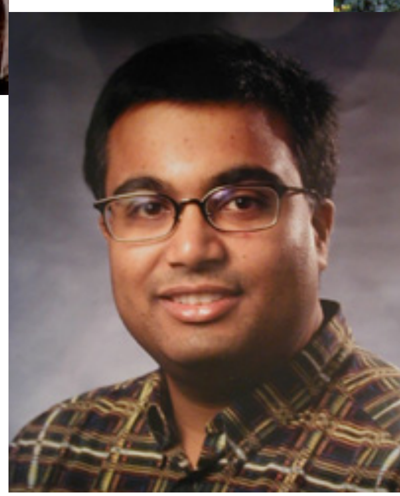


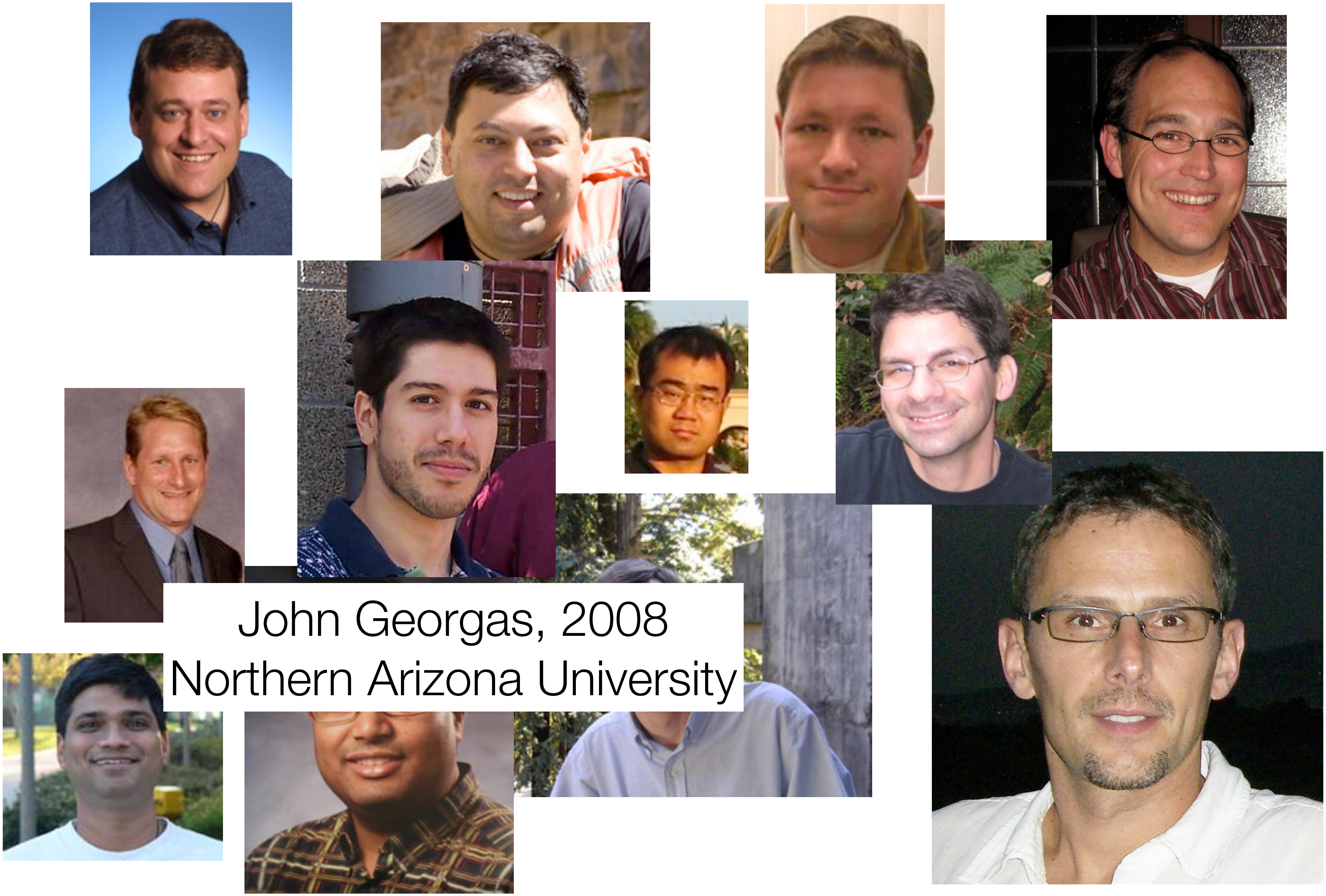
Eric Dashofy, 2007

The Aerospace Corporation





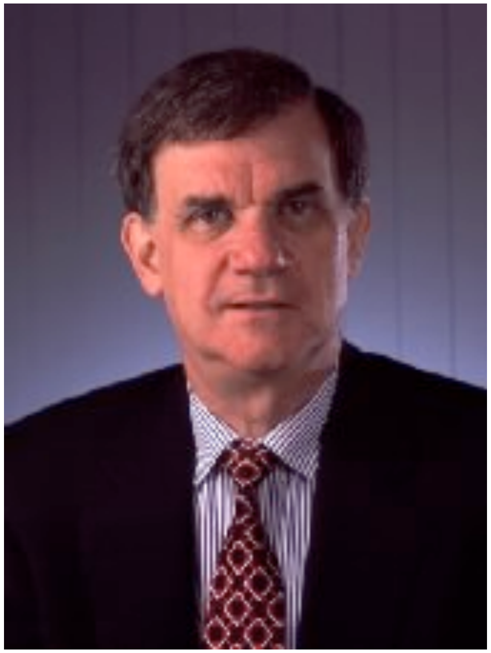


John Georgas, 2008
Northern Arizona University

Hazel Asuncion, 2009
UC Irvine (post-doc)

Justin Erenkrantz, 2009
The Apache Foundation


Scott Hendrickson, 2009
The Aerospace Corporation



Lee Osterweil
UMass

Alex Wolf
Imperial

André van der Hoek
UCIrvine

# Outline

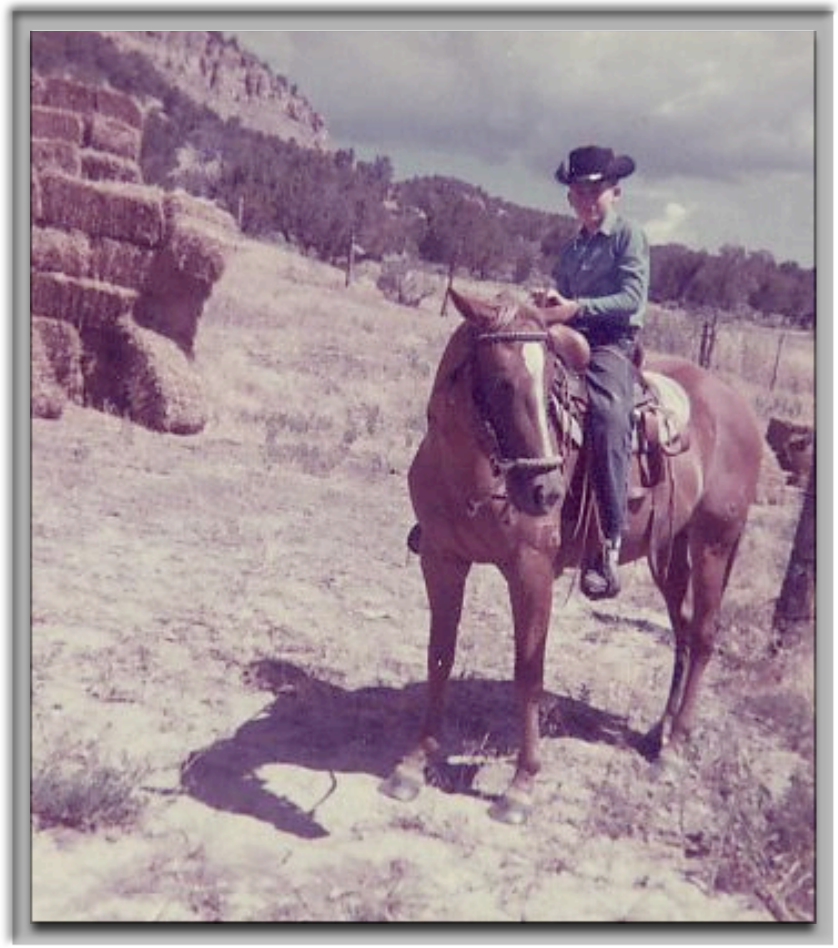
How did I get here?

Just what *IS* architecture?

Some REST-ful thoughts

A note about research and publications

“Just one more thing”



# How Did I Get Here?
# I Love Design

Painting

Photography

Music

*Software*

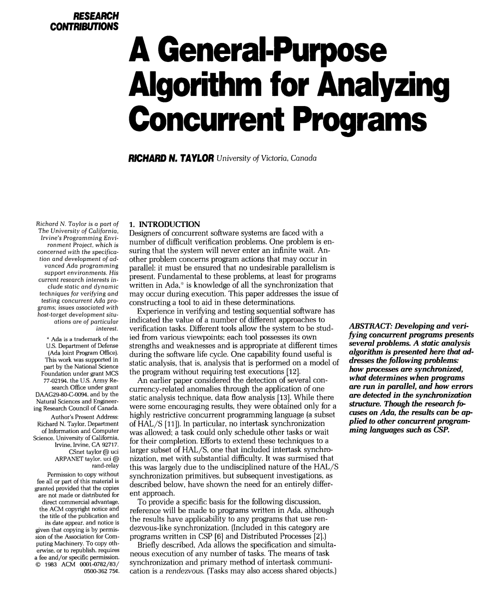
# Aiding Design

# Aiding Design

First focus: aiding in the design of concurrent programs

# Aiding Design

First focus:  aiding in the design of concurrent programs



RESEARCH CONTRIBUTIONS

## A General-Purpose Algorithm for Analyzing Concurrent Programs

RICHARD N. TAYLOR *University of Victoria, Canada*

*ABSTRACT: Developing and verifying concurrent programs presents several problems. A static analysis algorithm is presented here that addresses the following problems: how processes are synchronized, what determines when programs are run in parallel, and how errors are detected in the synchronization structure. Though the research focuses on Ada, the results can be applied to other concurrent programming languages such as CSP.*

### 1. INTRODUCTION

Designers of concurrent software systems are faced with a number of difficult verification problems. One problem is ensuring that the system will never enter an infinite wait. Another problem concerns program actions that may occur in parallel: it must be ensured that no undesirable parallelism is present. Fundamental to these problems, at least for programs written in Ada,* is knowledge of all the synchronization that may occur during execution. This paper addresses the issue of constructing a tool to aid in these determinations.

Experience in verifying and testing sequential software has indicated the value of a number of different approaches to verification tasks. Different tools allow the system to be studied from various viewpoints; each tool possesses its own strengths and weaknesses and is appropriate at different times during the software life cycle. One capability found useful is static analysis, that is, analysis that is performed on a model of the program without requiring test executions [12].

An earlier paper considered the detection of several concurrency-related anomalies through the application of one static analysis technique, data flow analysis [13]. While there were some encouraging results, they were obtained only for a highly restrictive concurrent programming language (a subset of HAL/S [11]). In particular, no intertask synchronization was allowed; a task could only schedule other tasks or wait for their completion. Efforts to extend these techniques to a larger subset of HAL/S, one that included intertask synchronization, met with substantial difficulty. It was surmised that this was largely due to the undisciplined nature of the HAL/S synchronization primitives, but subsequent investigations, as described below, have shown the need for an entirely different approach.

To provide a specific basis for the following discussion, reference will be made to programs written in Ada, although the results have applicability to any programs that use rendezvous-like synchronization. (Included in this category are programs written in CSP [6] and Distributed Processes [2].)

Briefly described, Ada allows the specification and simultaneous execution of any number of tasks. The means of task synchronization and primary method of intertask communication is a *rendezvous*. (Tasks may also access shared objects.)

*Richard N. Taylor is a part of The University of California, Irvine's Programming Environment Project, which is concerned with the specification and development of advanced Ada programming support environments. His current research interests include static and dynamic techniques for verifying and testing concurrent Ada programs; issues associated with host-target development situations are of particular interest.*

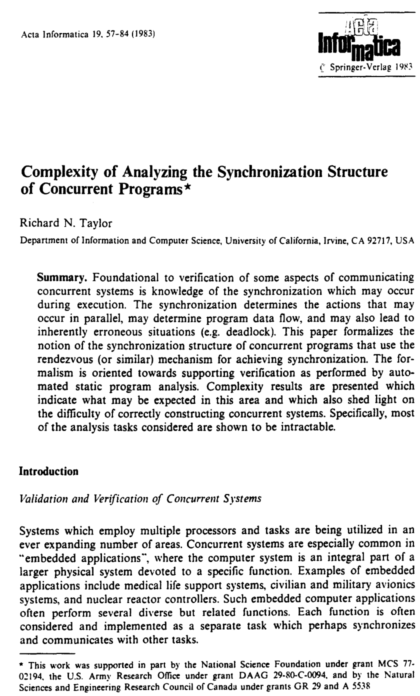
* Ada is a trademark of the U.S. Department of Defense (Ada Joint Program Office).

This work was supported in part by the National Science Foundation under grant MCS 77-02194, the U.S. Army Research Office under grant DAAG29-80-C-0094, and by the Natural Sciences and Engineering Research Council of Canada.

Author's Present Address: Richard N. Taylor, Department of Information and Computer Science, University of California, Irvine, Irvine, CA 92717. CSnet taylor @ uci ARPANET taylor, uci @ rand-relay







Acta Informatica 19, 57–84 (1983)

Acta Informatica © Springer-Verlag 1983

## Complexity of Analyzing the Synchronization Structure of Concurrent Programs*

Richard N. Taylor

Department of Information and Computer Science, University of California, Irvine, CA 92717, USA

**Summary.** Foundational to verification of some aspects of communicating concurrent systems is knowledge of the synchronization which may occur during execution. The synchronization determines the actions that may occur in parallel, may determine program data flow, and may also lead to inherently erroneous situations (e.g. deadlock). This paper formalizes the notion of the synchronization structure of concurrent programs that use the rendezvous (or similar) mechanism for achieving synchronization. The formalism is oriented towards supporting verification as performed by automated static program analysis. Complexity results are presented which indicate what may be expected in this area and which also shed light on the difficulty of correctly constructing concurrent systems. Specifically, most of the analysis tasks considered are shown to be intractable.

### Introduction

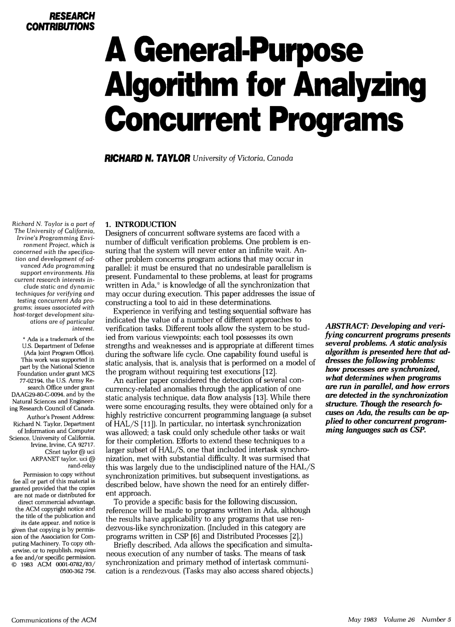
*Validation and Verification of Concurrent Systems*

Systems which employ multiple processors and tasks are being utilized in an ever expanding number of areas. Concurrent systems are especially common in "embedded applications", where the computer system is an integral part of a larger physical system devoted to a specific function. Examples of embedded applications include medical life support systems, civilian and military avionics systems, and nuclear reactor controllers. Such embedded computer applications often perform several diverse but related functions. Each function is often considered and implemented as a separate task which perhaps synchronizes and communicates with other tasks.

* This work was supported in part by the National Science Foundation under grant MCS 77-02194, the U.S. Army Research Office under grant DAAG 29-80-C-0094, and by the Natural Sciences and Engineering Research Council of Canada under grants GR 29 and A 5538

# Aiding Design

First focus: aiding in the design of concurrent programs



RESEARCH CONTRIBUTIONS

**A General-Purpose Algorithm for Analyzing Concurrent Programs**

**RICHARD N. TAYLOR** *University of Victoria, Canada*

*Richard N. Taylor is a part of The University of California, Irvine's Programming Environment Project, which is concerned with the specification and development of advanced Ada programming support environments. His current research interests include static and dynamic techniques for verifying and testing concurrent Ada programs; issues associated with host-target development situations are of particular interest.*

* Ada is a trademark of the U.S. Department of Defense (Ada Joint Program Office).

This work was supported in part by the National Science Foundation under grant MCS 77-02194, the U.S. Army Research Office under grant DAAG29-80-C-0094, and by the Natural Sciences and Engineering Research Council of Canada.

Author's Present Address: Richard N. Taylor, Department of Information and Computer Science, University of California, Irvine, Irvine, CA 92717. CSnet taylor @ uci ARPANET taylor. uci @ rand-relay



**1. INTRODUCTION**

Designers of concurrent software systems are faced with a number of difficult verification problems. One problem is ensuring that the system will never enter an infinite wait. Another problem concerns program actions that may occur in parallel: it must be ensured that no undesirable parallelism is present. Fundamental to these problems, at least for programs written in Ada,* is knowledge of all the synchronization that may occur during execution. This paper addresses the issue of constructing a tool to aid in these determinations.

Experience in verifying and testing sequential software has indicated the value of a number of different approaches to verification tasks. Different tools allow the system to be studied from various viewpoints; each tool possesses its own strengths and weaknesses and is appropriate at different times during the software life cycle. One capability found useful is static analysis, that is, analysis that is performed on a model of the program without requiring test executions [12].

An earlier paper considered the detection of several concurrency-related anomalies through the application of one static analysis technique, data flow analysis [13]. While there were some encouraging results, they were obtained only for a highly restrictive concurrent programming language (a subset of HAL/S [11]). In particular, no intertask synchronization was allowed; a task could only schedule other tasks or wait for their completion. Efforts to extend these techniques to a larger subset of HAL/S, one that included intertask synchronization, met with substantial difficulty. It was surmised that this was largely due to the undisciplined nature of the HAL/S synchronization primitives, but subsequent investigations, as described below, have shown the need for an entirely different approach.

To provide a specific basis for the following discussion, reference will be made to programs written in Ada, although the results have applicability to any programs that use rendezvous-like synchronization. (Included in this category are programs written in CSP [6] and Distributed Processes [2].)

Briefly described, Ada allows the specification and simultaneous execution of any number of tasks. The means of task synchronization and primary method of intertask communication is a *rendezvous*. (Tasks may also access shared objects.)

***ABSTRACT: Developing and verifying concurrent programs presents several problems. A static analysis algorithm is presented here that addresses the following problems: how processes are synchronized, what determines when programs are run in parallel, and how errors are detected in the synchronization structure. Though the research focuses on Ada, the results can be applied to other concurrent programming languages such as CSP.***



A Concurrency Analysis Tool Suite for Ada Programs: Rationale, Design, and Preliminary Experience

MICHAL YOUNG
Purdue University
and
RICHARD N. TAYLOR, DAVID L. LEVINE, KARI A. NIES, and DEBRA BRODBECK
University of California

CATS (Concurrency Analysis Tool Suite) 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. Meeting these objectives together is significantly more difficult than meeting any of them alone. We describe the design and rationale of CATS and report experience with an implementation. The issues addressed here are primarily practical concerns for modularizing and integrating tools for analysis of actual source programs. We also report successful application of CATS to major subsystems of a (nontoy) highly concurrent user interface system.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.6 [**Software Engineering**]: Programming Environments; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*

General Terms: Design, Reliability, Verification

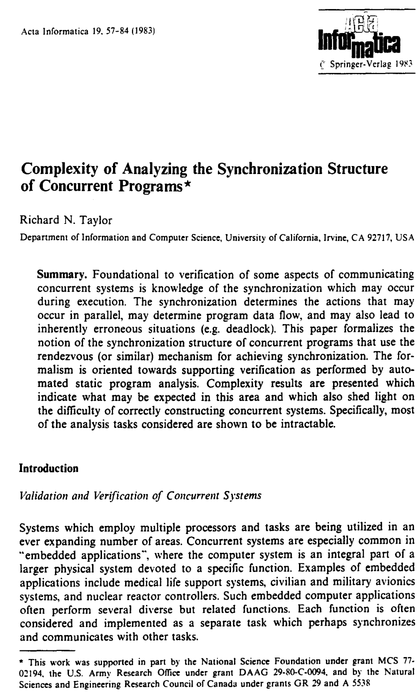
Additional Key Words and Phrases: Ada, concurrency, software development environments, static analysis, tool integration

This article is a major revision of Young et al. [1989]. This material is based on work sponsored by the Defense Advanced Research Projects Agency under grant MDA972-91-J-1010. Additional support was provided by the Software Engineering Research Center, an NSF Industry/University Cooperative Research Center, and by the National Science Foundation under grant CCR-9010135. The content of the information does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

Authors' addresses: M. Young, Software Engineering Research Center, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398; email: young@cs.purdue.edu; R. N. Taylor, D. L. Levine, K. A. Nies, and D. Brodbeck, Department of Information and Computer Science, University of California, Irvine, CA 92717-3425; email: {taylor; kari; brodbeck}@ics.uci.edu; levine@louie.timeplex.com.







Acta Informatica 19, 57-84 (1983)

Acta Informatica
© Springer-Verlag 1983

**Complexity of Analyzing the Synchronization Structure of Concurrent Programs***

Richard N. Taylor
Department of Information and Computer Science, University of California, Irvine, CA 92717, USA

**Summary.** Foundational to verification of some aspects of communicating concurrent systems is knowledge of the synchronization which may occur during execution. The synchronization determines the actions that may occur in parallel, may determine program data flow, and may also lead to inherently erroneous situations (e.g. deadlock). This paper formalizes the notion of the synchronization structure of concurrent programs that use the rendezvous (or similar) mechanism for achieving synchronization. The formalism is oriented towards supporting verification as performed by automated static program analysis. Complexity results are presented which indicate what may be expected in this area and which also shed light on the difficulty of correctly constructing concurrent systems. Specifically, most of the analysis tasks considered are shown to be intractable.

**Introduction**

*Validation and Verification of Concurrent Systems*

Systems which employ multiple processors and tasks are being utilized in an ever expanding number of areas. Concurrent systems are especially common in "embedded applications", where the computer system is an integral part of a larger physical system devoted to a specific function. Examples of embedded applications include medical life support systems, civilian and military avionics systems, and nuclear reactor controllers. Such embedded computer applications often perform several diverse but related functions. Each function is often considered and implemented as a separate task which perhaps synchronizes and communicates with other tasks.

* This work was supported in part by the National Science Foundation under grant MCS 77-02194, the U.S. Army Research Office under grant DAAG 29-80-C-0094, and by the Natural Sciences and Engineering Research Council of Canada under grants GR 29 and A 5538

# Environments

# Environments

First focus: analysis and testing environments

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 13, 697–713 (1983)

## An Integrated Verification and Testing Environment

RICHARD N. TAYLOR

*Department of Information and Computer Science, University of California, Irvine Irvine, California 92717, U.S.A.*

SUMMARY

**A verification and testing environment that includes static analysis, symbolic execution, and dynamic analysis capabilities is presented. Tool integration and co-operation are promoted through use of an intermediate program representation and a system data manager. A substantial user interface aids application of the tools. Their use is guided by a verification and testing methodology on which the system's design is based. The environment has been engineered to support the production of flight control software written in HAL/S. The environment itself is written in Pascal and is designed to be portable. Several development experiences are described. The environment demonstrates that a strong, unified verification and testing environment can be built; it serves as a basis for future investigations.**

KEY WORDS Verification and testing Programming environments Tools Reliability Software development aids Embedded software

### INTRODUCTION

During the past decade, a variety of software verification and testing techniques have appeared, each seeking to aid in the process of producing reliable code. Of course no single technique is able to satisfy all the needs in this area; different techniques are suited to different aspects of the job. Study of the various techniques has led to some important understandings, however. First, their relative efficacies and efficiencies have become known.[1] Second, it has become clear that many of the techniques have complementary characteristics. That is, the strengths of one technique are able to compensate for the weakness of another. These investigations have led to the formulation of a verification and testing (V & T) methodology: a way of applying the techniques to obtain their maximum benefit.[2–4] Indeed their co-operative application can result in a synergistic effect.

Study of the methodology and how it may be applied practically has led to another understanding: not only do the techniques functionally complement each other, but significant potential also exists for their physical integration. This paper reports, therefore, on a V & T environment designed to provide the functionality of the useful techniques, enable application of the V & T methodology, and promote efficient practical application of both. The components and integration scheme of the environment are described and a summary of the methodology is given. Some advances in fundamental technology are noted also.

This environment stands in distinct contrast with early collections of testing and verification tools. Foremost is the provision of direct support for a definite V & T methodology. This methodology enables aspects of a program's correctness to be

0038–0644/83/080697–17$01.70 *Received 7 June 1982*
 *Revised 3 November 1982 and 21 March 1983*

# Environments

First focus: analysis and testing environments

Second: programming environments

## Steps to an Advanced Ada[1] Programming Environment

RICHARD N. TAYLOR AND THOMAS A. STANDISH, MEMBER, IEEE

***Abstract*–Conceptual simplicity, tight coupling of tools, and effective support of host-target software development will characterize advanced Ada programming support environments. Several important principles have been demonstrated in the Arcturus system, including template-assisted Ada editing, command completion using Ada as a command language, and combining the advantages of interpretation and compilation. Other principles, relating to analysis, testing, and debugging of concurrent Ada programs, have appeared in other contexts. This paper discusses several of these topics, considers how they can be integrated, and argues for their inclusion in an environment appropriate for software development in the late 1980's.**

***Index Terms*–Ada, concurrency, debugging, programming environments, static analysis.**

### Introduction

In this paper we present concepts and ideas we think should characterize an advanced APSE (Ada programming support environment). These range from already proven concepts to emerging ideas currently being explored and prototyped. Several important issues remain unresolved. Our intention is to explore issues, problems, and opportunities.

*Brief Background*

Ada is a new programming language designed to promote reliability and maintainability of embedded computer software. Such software is often large (upwards of 100 000 instructions) and long-lived (15–25 years from conception to retirement). It must often meet requirements for concurrent processing and real-time performance.

If we are to produce and maintain Ada software that is *reliable, affordable,* and *adaptable,* the characteristics of Ada may not be the only important matter to consider. In addition, the characteristics of Ada software development environments may well be critical. Standish [8] makes the case that it is time to seize the opportunity to conceptualize what sort of advanced programming support tools should populate a mature APSE of high utility and effectiveness. In this context, consideration arises of support tools for modern programming practices, software reuse, interactive programming, software project management, project database support, improved testing and verification, and improved program understanding techniques.

At Irvine an experimental approach is being taken, using a prototype interactive Ada programming environment called *Arcturus.* We are using Arcturus as a platform for investigation and resolution of issues such as the following.

Manuscript received January 31, 1984. This work was supported in part by the Defense Advanced Research Projects Agency of the U.S. Department of Defense under Contract (ONR) N00039-83-C-0567 to the Irvine Programming Environment Project.

The authors are with the Programming Environment Project, Department of Information and Computer Science, University of California, Irvine, CA 92717.

[1] Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

0098-5589/85/0300-0302$01.00 © 1985 IEEE

**and testing methodology on which the system's design is based. The environment has been engineered to support the production of flight control software written in HAL/S. The environment itself is written in Pascal and is designed to be portable. Several development experiences are described. The environment demonstrates that a strong, unified verification and testing environment can be built; it serves as a basis for future investigations.**

KEY WORDS Verification and testing Programming environments Tools Reliability Software development aids Embedded software

### INTRODUCTION

During the past decade, a variety of software verification and testing techniques have appeared, each seeking to aid in the process of producing reliable code. Of course no single technique is able to satisfy all the needs in this area; different techniques are suited to different aspects of the job. Study of the various techniques has led to some important understandings, however. First, their relative efficacies and efficiencies have become known.[1] Second, it has become clear that many of the techniques have complementary characteristics. That is, the strengths of one technique are able to compensate for the weakness of another. These investigations have led to the formulation of a verification and testing (V & T) methodology: a way of applying the techniques to obtain their maximum benefit.[2–4] Indeed their co-operative application can result in a synergistic effect.

Study of the methodology and how it may be applied practically has led to another understanding: not only do the techniques functionally complement each other, but significant potential also exists for their physical integration. This paper reports, therefore, on a V & T environment designed to provide the functionality of the useful techniques, enable application of the V & T methodology, and promote efficient practical application of both. The components and integration scheme of the environment are described and a summary of the methodology is given. Some advances in fundamental technology are noted also.

This environment stands in distinct contrast with early collections of testing and verification tools. Foremost is the provision of direct support for a definite V & T methodology. This methodology enables aspects of a program's correctness to be

0038–0644/83/080697–17$01.70 *Received 7 June 1982*
 *Revised 3 November 1982 and 21 March 1983*

# Environments

First focus: analysis and testing environments

Second: programming environments

Third: software development environments

Steps to an Advanced Ada[1] Programming Environment

## Foundations for the Arcadia Environment Architecture

Richard N. Taylor, Frank C. Belz†, Lori A. Clarke*, Leon Osterweil**,
Richard W. Selby, Jack C. Wileden*, Alexander L. Wolf‡, Michal Young

Department of Information and Computer Science
University of California, Irvine[1]

*Department of Computer and Information Science
University of Massachusetts, Amherst[2]

**Department of Computer Science
University of Colorado, Boulder[3]

†TRW[4]
Redondo Beach, California

‡AT&T Bell Laboratories,
600 Mountain Avenue
Murray Hill, NJ

### Abstract

Early software environments have supported a narrow range of activities (*programming* environments) or else been restricted to a single "hard-wired" software development process. The Arcadia research project is investigating the construction of software environments that are tightly integrated, yet flexible and extensible enough to support experimentation with alternative software processes and tools. This has led us to view an environment as being composed of two distinct, cooperating parts. One is the *variant* part, consisting of process programs and the tools and objects used and defined by those programs. The other is the fixed part, or *infrastructure*, supporting creation, execution, and change to the constituents of the variant part. The major components of the infrastructure are a process programming language and interpreter, object management system, and user interface management system. Process programming facilitates precise definition and automated support of software development and maintenance activities. The object management system provides typing, relationships, persistence, distribution and concurrency control capabilities. The user interface management system mediates communication between human users and executing processes, providing pleasant and uniform access to all facilities of the environment. Research in each of these areas and the interaction among them is described.

### 1 INTRODUCTION

The purpose of a software environment is to support users in their software development and maintenance activities. Past attempts to do this have indicated the vast scope and complexity of this problem. The Arcadia project is a consortium research effort aimed at addressing an unusually broad range of software environment issues. In particular, the Arcadia project is concerned with simultaneously investigating and developing prototype demonstrations of:

- environment architectures for organizing large collections of tools and facilitating their interactions with users as well as with each other,
- tools to facilitate the testing and analysis of concurrent and sequential software, and
- frameworks for environment and tool evaluation.

This paper describes the research rationale and approach being taken by Arcadia researchers in investigating environment architecture issues. Although details concerning the tool suite and the evaluation framework are outside the scope of this paper, attempting to assemble these components into a coherent environment will provide a non-trivial test case for experimentally evaluating this architecture.

The remainder of this section presents a high-level overview of our proposed environment architecture. The major components of the architecture are described. Each of these represents a major research subarea being investigated as part of this project. The ensuing sections describe the major goals and rationale for each of these subareas. Although each subarea is an interesting research project in its own right, the most challenging questions are often raised by the interactions among subareas. Indeed, it is the importance and complexity of these interactions that requires research in the subareas be pursued cooperatively. One of the novel features of the Arcadia project is that it is synergistically exploring many of these issues.



[1]This work was supported in part by the National Science Foundation under grant CCR-8704311, with cooperation from the Defense Advanced Research Projects Agency (ARPA Order 6108, Program Code 7T10), by the National Science Foundation under grants CCR-8451421 and CCR-8521398, Hughes Aircraft (PYI program), and TRW (PYI program).

[2]This work was supported in part by the National Science Foundation under grant CCR-87-04478, with cooperation from the Defense Advanced Research Projects Agency (ARPA Order 6104), and by the National Science Foundation under grants DCR-8404217 and DCR-8408143.

[3]This work was supported in part by the National Science Foundation under grant CCR-8705162, with cooperation from the Defense Advanced Research Projects Agency (ARPA Order 6100, Program Code 7E20), by the National Science Foundation under grant DCR-0403341, and by the U.S. Department of Energy under grant DE-FG02-84ER13283.

[4]This work was sponsored in part by TRW and by the Defense Advanced Research Projects Agency/Information Systems Technology Office, ARPA Order 9152, issued by the Space and Naval Warfare Systems Command under contract N00039-88-C-0047.

1

**Recommended by:** *Stowe Boyd*

environment are described and a summary of the methodology is given. Some advances in fundamental technology are noted also.

This environment stands in distinct contrast with early collections of testing and verification tools. Foremost is the provision of direct support for a definite V & T methodology. This methodology enables aspects of a program's correctness to be

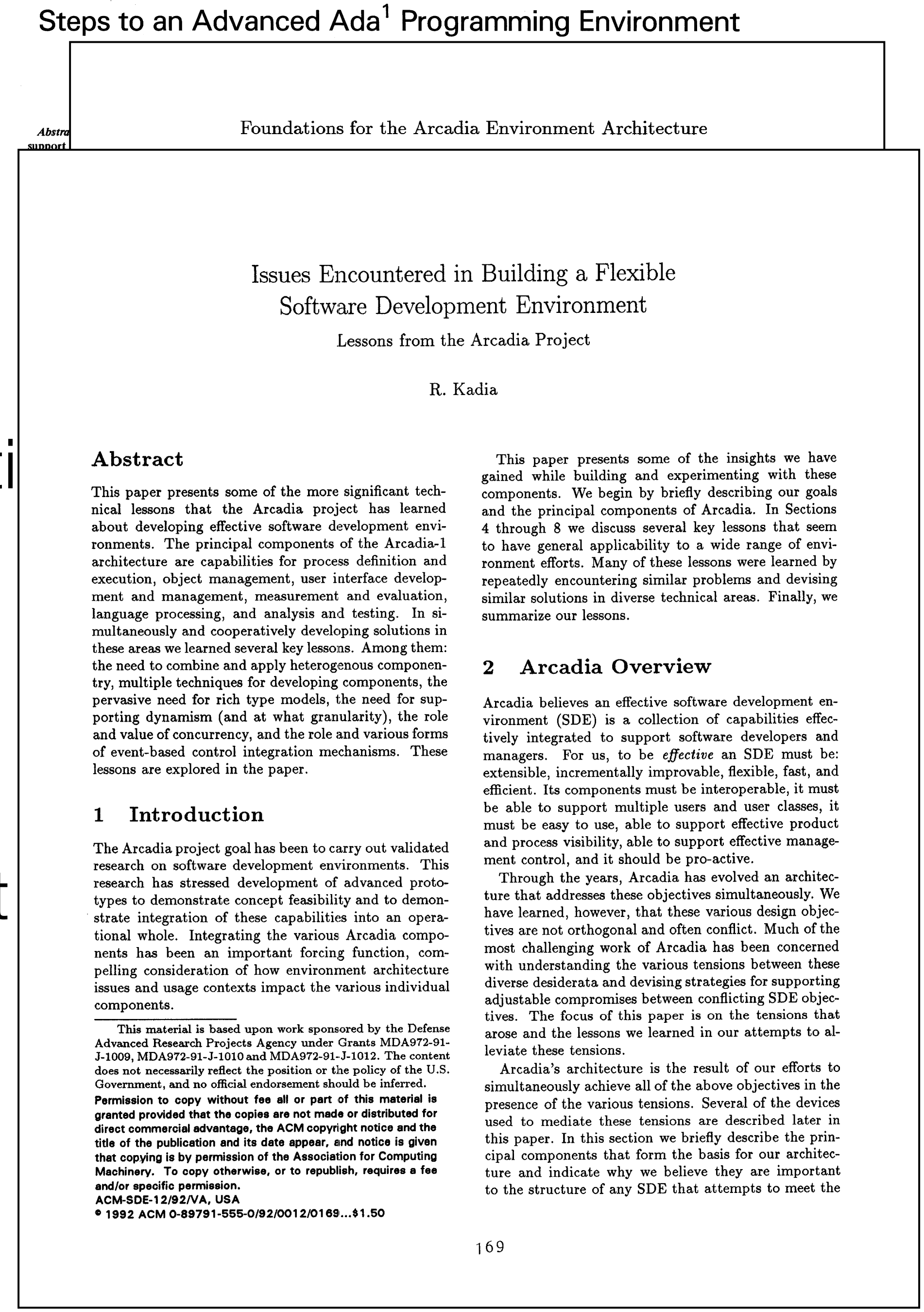
0038–0644/83/080697–17$01.70 Received 7 June 1982
© 1983 by John Wiley & Sons, Ltd. Revised 3 November 1982 and 21 March 1983

# Environments

First focus: analysis and testi
environments

Second: programming
environments

Third: software development
environments



Steps to an Advanced Ada[1] Programming Environment

Foundations for the Arcadia Environment Architecture

Issues Encountered in Building a Flexible Software Development Environment

Lessons from the Arcadia Project

R. Kadia

**Abstract**

This paper presents some of the more significant technical lessons that the Arcadia project has learned about developing effective software development environments. The principal components of the Arcadia-1 architecture are capabilities for process definition and execution, object management, user interface development and management, measurement and evaluation, language processing, and analysis and testing. In simultaneously and cooperatively developing solutions in these areas we learned several key lessons. Among them: the need to combine and apply heterogenous componentry, multiple techniques for developing components, the pervasive need for rich type models, the need for supporting dynamism (and at what granularity), the role and value of concurrency, and the role and various forms of event-based control integration mechanisms. These lessons are explored in the paper.

**1 Introduction**

The Arcadia project goal has been to carry out validated research on software development environments. This research has stressed development of advanced prototypes to demonstrate concept feasibility and to demonstrate integration of these capabilities into an operational whole. Integrating the various Arcadia components has been an important forcing function, compelling consideration of how environment architecture issues and usage contexts impact the various individual components.

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grants MDA972-91-J-1009, MDA972-91-J-1010 and MDA972-91-J-1012. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

**Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.**
**ACM-SDE-12/92/VA, USA**
**© 1992 ACM 0-89791-555-0/92/0012/0169...$1.50**

This paper presents some of the insights we have gained while building and experimenting with these components. We begin by briefly describing our goals and the principal components of Arcadia. In Sections 4 through 8 we discuss several key lessons that seem to have general applicability to a wide range of environment efforts. Many of these lessons were learned by repeatedly encountering similar problems and devising similar solutions in diverse technical areas. Finally, we summarize our lessons.

**2 Arcadia Overview**

Arcadia believes an effective software development environment (SDE) is a collection of capabilities effectively integrated to support software developers and managers. For us, to be *effective* an SDE must be: extensible, incrementally improvable, flexible, fast, and efficient. Its components must be interoperable, it must be able to support multiple users and user classes, it must be easy to use, able to support effective product and process visibility, able to support effective management control, and it should be pro-active.

Through the years, Arcadia has evolved an architecture that addresses these objectives simultaneously. We have learned, however, that these various design objectives are not orthogonal and often conflict. Much of the most challenging work of Arcadia has been concerned with understanding the various tensions between these diverse desiderata and devising strategies for supporting adjustable compromises between conflicting SDE objectives. The focus of this paper is on the tensions that arose and the lessons we learned in our attempts to alleviate these tensions.

Arcadia's architecture is the result of our efforts to simultaneously achieve all of the above objectives in the presence of the various tensions. Several of the devices used to mediate these tensions are described later in this paper. In this section we briefly describe the principal components that form the basis for our architecture and indicate why we believe they are important to the structure of any SDE that attempts to meet the

169

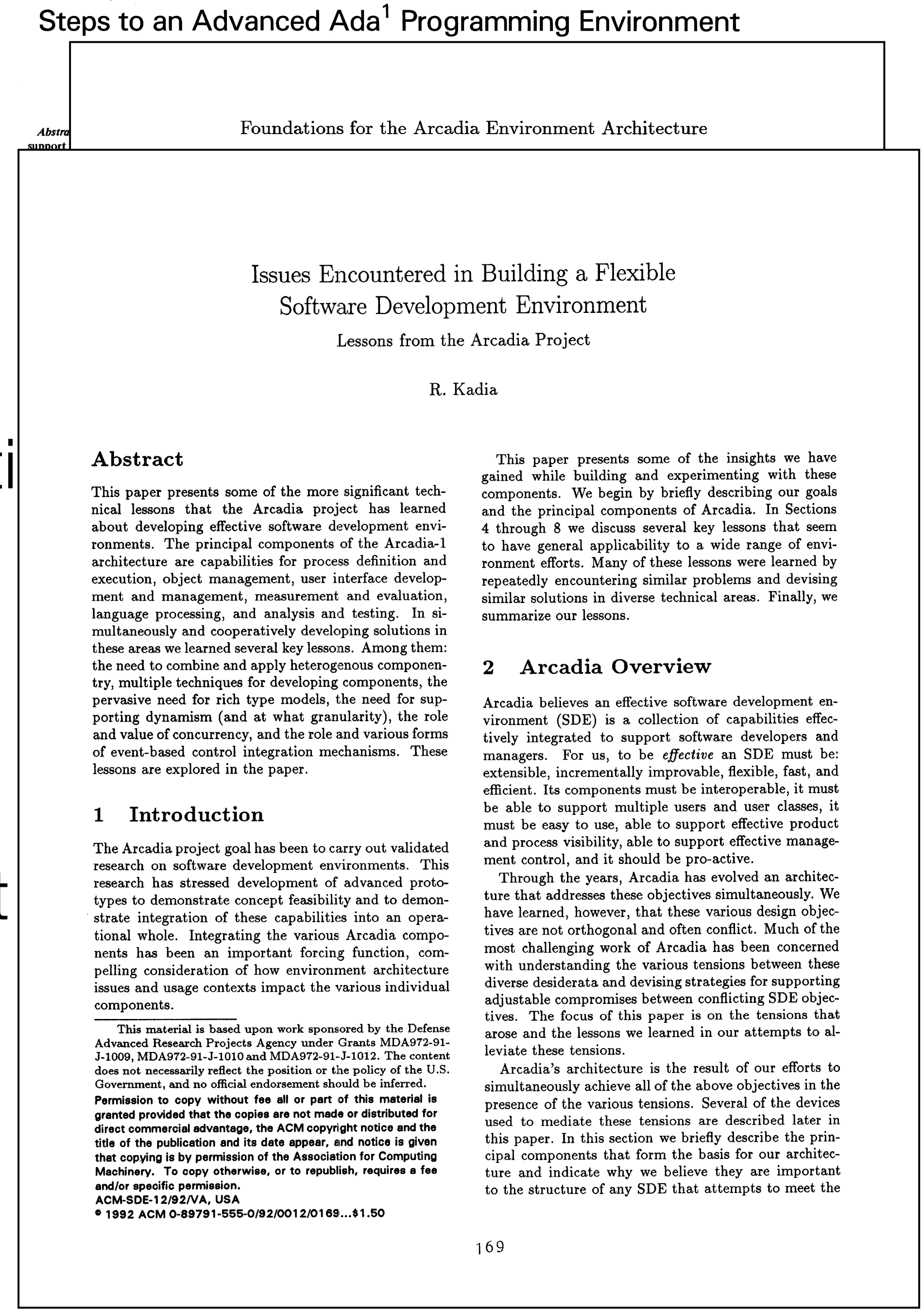
# Environments

First focus: analysis and testi
environments

Second: programming
environments

Third: software development
environments

Especially UI aspects



Steps to an Advanced Ada[1] Programming Environment

Foundations for the Arcadia Environment Architecture

Issues Encountered in Building a Flexible
Software Development Environment
Lessons from the Arcadia Project

R. Kadia

## Abstract

This paper presents some of the more significant technical lessons that the Arcadia project has learned about developing effective software development environments. The principal components of the Arcadia-1 architecture are capabilities for process definition and execution, object management, user interface development and management, measurement and evaluation, language processing, and analysis and testing. In simultaneously and cooperatively developing solutions in these areas we learned several key lessons. Among them: the need to combine and apply heterogenous componentry, multiple techniques for developing components, the pervasive need for rich type models, the need for supporting dynamism (and at what granularity), the role and value of concurrency, and the role and various forms of event-based control integration mechanisms. These lessons are explored in the paper.

## 1 Introduction

The Arcadia project goal has been to carry out validated research on software development environments. This research has stressed development of advanced prototypes to demonstrate concept feasibility and to demonstrate integration of these capabilities into an operational whole. Integrating the various Arcadia components has been an important forcing function, compelling consideration of how environment architecture issues and usage contexts impact the various individual components.

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grants MDA972-91-J-1009, MDA972-91-J-1010 and MDA972-91-J-1012. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
ACM-SDE-12/92/VA, USA
© 1992 ACM 0-89791-555-0/92/0012/0169...$1.50

This paper presents some of the insights we have gained while building and experimenting with these components. We begin by briefly describing our goals and the principal components of Arcadia. In Sections 4 through 8 we discuss several key lessons that seem to have general applicability to a wide range of environment efforts. Many of these lessons were learned by repeatedly encountering similar problems and devising similar solutions in diverse technical areas. Finally, we summarize our lessons.

## 2 Arcadia Overview

Arcadia believes an effective software development environment (SDE) is a collection of capabilities effectively integrated to support software developers and managers. For us, to be *effective* an SDE must be: extensible, incrementally improvable, flexible, fast, and efficient. Its components must be interoperable, it must be able to support multiple users and user classes, it must be easy to use, able to support effective product and process visibility, able to support effective management control, and it should be pro-active.

Through the years, Arcadia has evolved an architecture that addresses these objectives simultaneously. We have learned, however, that these various design objectives are not orthogonal and often conflict. Much of the most challenging work of Arcadia has been concerned with understanding the various tensions between these diverse desiderata and devising strategies for supporting adjustable compromises between conflicting SDE objectives. The focus of this paper is on the tensions that arose and the lessons we learned in our attempts to alleviate these tensions.

Arcadia's architecture is the result of our efforts to simultaneously achieve all of the above objectives in the presence of the various tensions. Several of the devices used to mediate these tensions are described later in this paper. In this section we briefly describe the principal components that form the basis for our architecture and indicate why we believe they are important to the structure of any SDE that attempts to meet the

169

# Software Environment Architectures and User Interface Facilities

MICHAL YOUNG, RICHARD N. TAYLOR, AND DENNIS B. TROUP

***Abstract*—User interface facilities are a crucial part of the infrastructure of a software environment. We discuss the particular demands and constraints on a user interface management system for a software environment, and the relation between the architecture of the environment and the user interface management system. A model for designing user interface management systems for large extensible environments is presented. This model synthesizes several recent advances in user interfaces and specializes them to the domain of software environments. The model can be applied to a wide variety of environment contexts. A prototype implementation is briefly described.**

***Index Terms*—Environments, interactive program structures, user interface management systems.**

## I. Introduction

A SOFTWARE environment provides automated support for some set of activities performed by software producers. An *environment architecture* is the set of basic facilities underlying the capabilities of a spectrum of environments and the rules for assembling a specific environment out of these and other facilities. Environment architectures typically deal with issues such as:

- the storage and manipulation of persistent objects,
- tool interface conventions,
- reuse of internal information and tools, and
- interaction with the user [1].

It is the last item which concerns us here: the basic facilities provided for interacting with users and how these are bound up with other environment capabilities to produce interactive tools.

The user interface management system (UIMS) is an active research area in its own right. This paper focuses on the relation between UIMS and environment architecture: the particular demands and constraints of software environments, and the interplay between UIMS issues and the other issues listed above. This paper is about the *architecture* of UIMS; it is not about graphical interaction techniques *per se*, nor about human factors in interactive programs. It describes a design which integrates several key ideas from UIMS research, adapting and extending them to serve the needs of large, flexible software environments. This design is embodied in Chiron, a UIMS for the Arcadia-1 software environment, and specific design decisions in Chiron are used to illustrate general points throughout the paper. The design is applicable to a wide range of environments.

Manuscript received January 15, 1988. This work was supported in part by the National Science Foundation under Grant CCR-8704311, with cooperation from the Defense Advanced Research Projects Agency, and by the National Science Foundation under Grants DCR-8502558 and DCR-8521398.

The authors are with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

IEEE Log Number 8820967.

## II. Context: Software Environment Architectures

Over the past decade the term "environment" has been applied to a wide variety of systems. These have ranged from editor-linker-compiler combinations to syntax-directed editors to sophisticated tool and object managers. We use the term to denote a set of facilities designed to aid its user in the performance of some job, where the "jobs" we are interested in supporting include the full spectrum of software development and maintenance activities. As such, the number of subactivities that can be supported is very large, as are the number and types of objects manipulated by the environment. The concerns which drive the following discussion, therefore, are much broader than those associated, for example, with structure editor environments.

Furthermore we are interested in integrated environments. From a user's perspective this means that certain properties of uniformity hold. While this is vital, and is addressed later in this paper, there can be much more to the notion of integration. From a tool (or environment) builder's perspective an environment is integrated if there are uniform schemes for tool invocation, tool combination/communication, data sharing, persistent object management, constraint maintenance, addition of new object types, addition of new tools, and so forth. We will not claim that techniques for achieving such integration are well known (nor that all of these issues are even well understood), but stating our general goals for environments helps to clarify some of the concerns which bear on the relationship between a user interface management system and other components of an environment.

Another pertinent interest of ours is in extensible environments. In our estimation the most useful environments are ones in which it is possible to add new capabilites to address new needs, to aid in supporting new jobs. New development and analysis techniques appear regularly; it is only sensible to integrate them into an environment. Doing so can be difficult, however, unless an en-

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 14, NO. 6, JUNE 1988 697

# Software Environment Architectures and User Interface Facilities

MICHAL YOUNG, RICHARD N. TAYLOR, AND DENNIS B. TROUP

*Abstract*—User interface facilities are a crucial part of the infrastructure of a software environment. We discuss the particular demands and constraints on a user interface management system for a software environment, and the relation between the architecture of the environment and the user interface management system. A model for designing user interface management systems for large extensible environments is presented. This model synthesizes several recent advances in user interfaces and specializes them to the domain of software environments. The model can be applied to a wide variety of environment contexts. A prototype implementation is briefly described.

*Index Terms*—Environments, interactive program structures, user interface management systems.

## I. Introduction

A SOFTWARE environment provides automated support for some set of activities performed by software producers. An *environment architecture* is the set of basic facilities underlying the capabilities of a spectrum of environments and the rules for assembling a specific environment out of these and other facilities. Environment architectures typically deal with issues such as:

- the storage and manipulation of persistent objects,
- tool interface conventions,
- reuse of internal information and tools, and
- interaction with the user [1].

It is the last item which concerns us here: the basic facilities provided for interacting with users and how these are bound up with other environment capabilities to produce interactive tools.

The user interface management system (UIMS) is an active research area in its own right. This paper focuses on the relation between UIMS and environment architecture: the particular demands and constraints of software environments, and the interplay between UIMS issues and the other issues listed above. This paper is about the *architecture* of UIMS; it is not about graphical interaction techniques *per se*, nor about human factors in interactive programs. It describes a design which integrates several key ideas from UIMS research, adapting and extending them to serve the needs of large, flexible software environments. This design is embodied in Chiron, a UIMS for the Arcadia-1 software environment, and specific design decisions in Chiron are used to illustrate general points throughout the paper. The design is applicable to a wide range of environments.

## II. Context: Software Environment Architectures

Over the past decade the term "environment" has been applied to a wide variety of systems. These have ranged from editor-linker-compiler combinations to syntax-directed editors to sophisticated tool and object managers. We use the term to denote a set of facilities designed to aid its user in the performance of some job, where the "jobs" we are interested in supporting include the full spectrum of software development and maintenance activities. As such, the number of subactivities that can be supported is very large, as are the number and types of objects manipulated by the environment. The concerns which drive the following discussion, therefore, are much broader than those associated, for example, with structure editor environments.

Furthermore we are interested in integrated environments. From a user's perspective this means that certain properties of uniformity hold. While this is vital, and is addressed later in this paper, there can be much more to the notion of integration. From a tool (or environment) builder's perspective an environment is integrated if there are uniform schemes for tool invocation, tool combination/communication, data sharing, persistent object management, constraint maintenance, addition of new object types, addition of new tools, and so forth. We will not claim that techniques for achieving such integration are well known (nor that all of these issues are even well understood), but stating our general goals for environments helps to clarify some of the concerns which bear on the relationship between a user interface management system and other components of an environment.

Another pertinent interest of ours is in extensible environments. In our estimation the most useful environments are ones in which it is possible to add new capabilites to address new needs, to aid in supporting new jobs. New development and analysis techniques appear regularly; it is only sensible to integrate them into an environment. Doing so can be difficult, however, unless an en-

Manuscript received January 15, 1988. This work was supported in part by the National Science Foundation under Grant CCR-8704311, with cooperation from the Defense Advanced Research Projects Agency, and by the National Science Foundation under Grants DCR-8502558 and DCR-8521398.

The authors are with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

IEEE Log Number 8820967.





# Chiron-1: A Software Architecture for User Interface Development, Maintenance, and Run-Time Support

RICHARD N. TAYLOR, KARI A. NIES, GREGORY ALAN BOLCER, CRAIG A. MACFARLANE, and KENNETH M. ANDERSON
University of California, Irvine
and
GREGORY F. JOHNSON
Northrop Corporation

The Chiron-1 user interface system demonstrates key techniques that enable a strict separation of an application from its user interface. These techniques include separating the control-flow aspects of the 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 data 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 that of the server, while nevertheless supporting object-oriented server concepts. The system has been used in several universities and research and development projects. It is available by anonymous ftp.

This article is a major revision and expansion of "Separations of Concerns in the Chiron-1 User Interface Development and Management System," which appeared in the Proceedings of InterCH1 93 [Taylor and Johnson 1993]. This material is based on work sponsored by the Advanced Research Projects Agency under Grant MDA972-91-J-1010. The content of the information does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.
Authors' addresses: R. N. Taylor, K. A. Nies, G. A. Bolcer, C. A. MacFarlane, and K. M. Anderson, Department of Information and Computer Science, University of California, Irvine, CA 92717-3425; G. F. Johnson, Northrop Corporation, Pico Rivera, CA.


ACM Transactions on Computer-Human Interaction, Vol. 2, No. 2, June 1995, Pages 105–144.

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 14, NO. 6, JUNE 1988 697

# Software Environment Architectures and User Interface Facilities

MICHAL YOUNG, RICHARD N. TAYLOR, AND DENNIS B. TROUP

*Abstract*—**User interface facilities are a crucial part of the infrastructure of a software environment. We discuss the particular demands and constraints on a user interface management system for a software environment, and the relation between the architecture of the environment and the user interface management system. A model for designing user interface management systems for large extensible environments is presented. This model synthesizes several recent advances in user interfaces and specializes them to the domain of software environments. The model can be applied to a wide variety of environment contexts. A prototype implementation is briefly described.**

*Index Terms*—**Environments, interactive program structures, user interface management systems.**

## I. Introduction

A SOFTWARE environment provides automated support for some set of activities performed by software producers. An *environment architecture* is the set of basic facilities underlying the capabilities of a spectrum of environments and the rules for assembling a specific environment out of these and other facilities. Environment architectures typically deal with issues such as:

- the storage and manipulation of persistent objects,
- tool interface conventions,
- reuse of internal information and tools, and
- interaction with the user [1].

It is the last item which concerns us here: the basic facilities provided for interacting with users and how these are bound up with other environment capabilities to produce interactive tools.

The user interface management system (UIMS) is an active research area in its own right. This paper focuses on the relation between UIMS and environment architecture: the particular demands and constraints of software environments, and the interplay between UIMS issues and the other issues listed above. This paper is about the *architecture* of UIMS; it is not about graphical interaction techniques *per se*, nor about human factors in interactive programs. It describes a design which integrates several key ideas from UIMS research, adapting and extending

them to ser
ronments. T
the Arcadia
decisions i
throughout
range of en

II.

Over the
applied to a
from edito
rected edit
We use the
aid its use
"jobs" we
spectrum o
tivities. As
supported i
objects ma
which drive
broader tha
editor envi
Furtherm
ments. Fro
properties
addressed l
the notion
builder's p
are unifor
tion/comm
agement, c
types, addi
claim that
well know
understood)
ments help
on the relat
system and
Another
vironments
ments are c
bilites to ad
New devel
larly; it is o
ment. Doin

Manuscript received January 15, 1988. This work was supported in part by the National Science Foundation under Grant CCR-8704311, with cooperation from the Defense Advanced Research Projects Agency, and by the National Science Foundation under Grants DCR-8502558 and DCR-8521398.

The authors are with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

IEEE Log Number 8820967.

0098-5589/88/0600-0697$01.00 ©



# Chiron-1: A Software Architecture for User Interface Development, Maintenance, and Run-Time Support

RICHARD N. TAYLOR, KARI A. NIES, GREGORY ALAN BOLCER, CRAIG A. MACFARLANE, and KENNETH M. ANDERSON
University of California, Irvine
and
GREGORY F. JOHNSON

iniques that enable a strict separation
es include separating the control-flow
urrent and may contain many threads.
from dialogue and abstract presenta-
architecture. To separate application
d *artists* are attached to instances of
DTs within the application implicitly
iple artists can be attached to ADTs,
and manipulation by either a single
n run in separate threads of control.
s to an abstract depiction hierarchy in
the specifics of particular windowing
execute in separate processes. The
stems: mechanisms are demonstrated
other than that of the server, while
The system has been used in several
available by anonymous ftp.

ions of Concerns in the Chiron-1 User
appeared in the Proceedings of In-
is based on work sponsored by the
2-91-J-1010. The content of the infor-
olicy of the U.S. Government, and no

C. A. MacFarlane, and K. M. Anderson,
ersity of California, Irvine, CA 92717-
A.

work for personal or classroom use is
r distributed for profit or commercial
on, and its date appear, and notice is
y otherwise, to republish, to post on
ermission and/or a fee.

, Vol. 2, No. 2, June 1995, Pages 105–144.

390 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 22, NO. 6, JUNE 1996

# A Component- and Message-Based Architectural Style for GUI Software

Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., *Member, IEEE*, Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow

**Abstract**—While a large fraction of application code is devoted to graphical user interface (GUI) functions, support for reuse in this domain has largely been confined to the creation of GUI toolkits ("widgets"). We present a novel architectural style directed at supporting larger grain reuse and flexible system composition. Moreover, the style supports design of distributed, concurrent applications. Asynchronous notification messages and asynchronous request messages are the sole basis for intercomponent communication. 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. Indeed, all components are oblivious to the existence of any components to which notification messages are sent. While our focus has been on applications involving graphical user interfaces, the style has the potential for broader applicability. Several trial applications using the style are described.

**Index Terms**—Architectural styles, message-based architectures, graphical user interfaces (GUIs), heterogeneity, concurrency.

✦

## 1 Introduction

Software architectural styles are key design idioms [8], [24]. UNIX's pipe-and-filter style is more than 20 years old; blackboard architectures have long been common in AI applications. User interface software has typically made use of two primary runtime architectures: the client-server style (as exemplified by X windows) and the call-back model, a control model in which application functions are invoked under the control of the user interface. Also well known is the model-view-controller (MVC) style [15], which is commonly exploited in Smalltalk applications. The Arch style is more recent, and has an associated metamodel [38].

This paper presents a new architectural style. It is designed to support the particular needs of applications that have a graphical user interface aspect, but the style clearly has the potential for supporting other types of applications. This style draws its key ideas from many sources, including the styles mentioned above, as well as specific experience with the Chiron-1 user interface development system [14], [34]. In the following exposition, the style is referred to as the Chiron-2, or C2, style.

A key motivating factor behind development of the C2 style is the emerging need, in the user interface community, for a more component-based development economy [37]. User interface software frequently accounts for a very large fraction of application software, yet reuse in the UI domain is typically limited to toolkit (widget) code. The architectural style presented supports a paradigm in which UI components, such as dialogs, structured graphics models of various levels of abstraction, and constraint managers, can more readily be reused. A variety of other goals are potentially supported as well. These goals include the ability to compose systems in which: components may be written in different programming languages, components may be running concurrently in a distributed, heterogeneous environment without shared address spaces, architectures may be changed at runtime, multiple users may be interacting with the system, multiple toolkits may be employed, multiple dialogs may be active and described in different formalisms, and multiple media types may be involved. We have not yet demonstrated that all these goals are achievable or especially supported by this style. We have examined several key properties and built several diverse experimental systems, however. The focus of this paper, therefore, is to present the style and describe the evidence we have to date. We believe our preliminary findings are encouraging and that the style has substantial utility "as is."

The new style can be informally summarized as a network of concurrent components hooked together by message routing devices. Central to the architectural style is a principle of limited visibility: a component within the hierarchy can only be aware of components "above" it and completely unaware of components which reside "beneath" it. Notions of above and below are used in this paper to support an intuitive understanding of the architectural style. As is typical with virtual machine diagrams found in operating systems textbooks, in this discussion the application code[1] is arbitrarily regarded as being at the top while

• R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead, Jr., J.E. Robbins, K.A. Nies, and P. Oreizy are with the University of California at Irvine. E-mail: Taylor@ics.uci.edu.
• D.L. Dubrow is with the Microsoft Corporation, 1 Microsoft Way, Redmond, WA, 98052. E-mail: debdu@microsoft.com.

Manuscript received July 1995; revised Jan. 1996.
Recommended for acceptance by D. Notkin and D.R. Jeffery.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S96044.

1. It is sometimes convenient to consider an application system as being subdivided into two parts: one part are those aspects of the system which do not directly perform any user interface functions (the "application"), and the other part are those aspects concerned with interacting with the user (the "user interface"). Such a distinction is rather arbitrary, and can usually be read as "those parts of an application system not constructed according to our architectural style and principles, and those parts which are."

0098-5589/96$05.00 ©1996 IEEE

# Software Architecture as a Field

If you look at the history of the research field, from Perry & Wolf and Shaw & Garlan onwards

- lots and lots of activity
- mostly concerning the structure of software systems
- but also addressing other views of systems

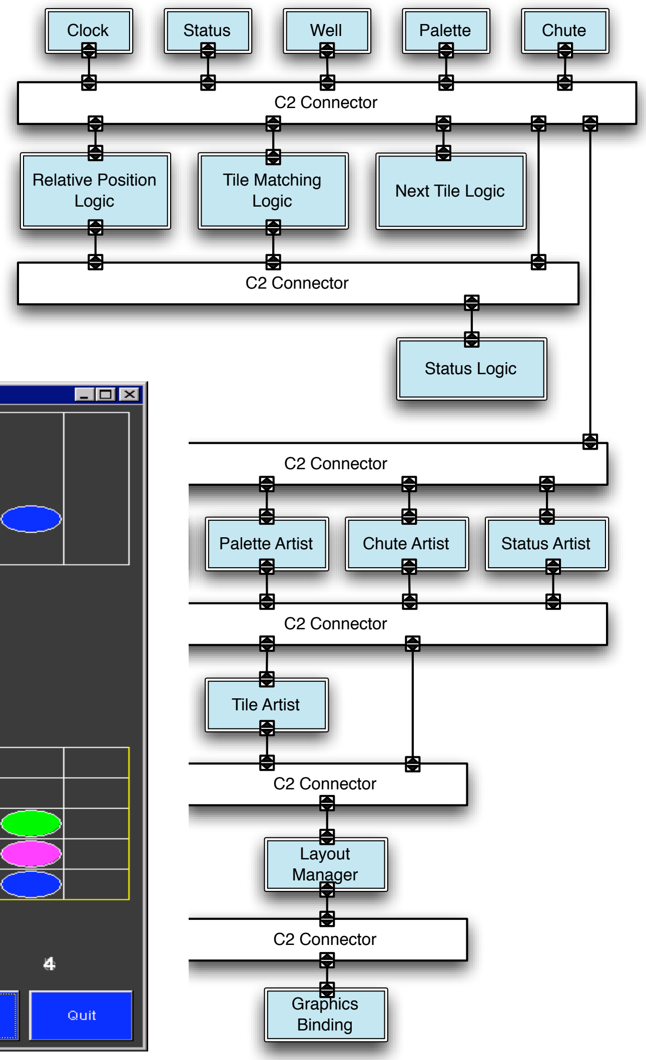
# Some of My Emphases

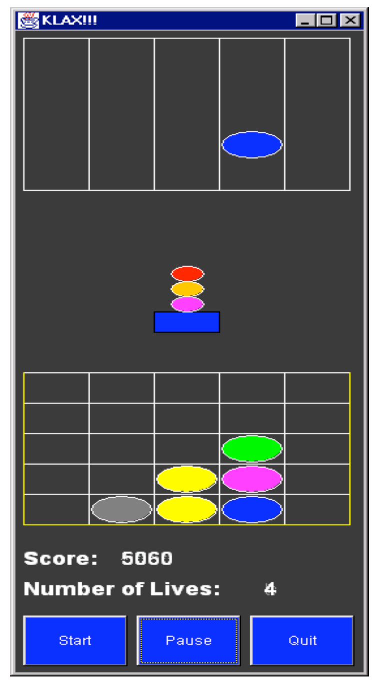
# Some of My Emphases

Structure & Style

# Some of My Emphases

## Structure & Style



# Some of My Emphases

Structure & Style

Modeling (ADLs)


70 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 26, NO. 1, JANUARY 2000

## A Classification and Comparison Framework for Software Architecture Description Languages

Nenad Medvidovic and Richard N. Taylor, *Member, IEEE Computer Society*

**Abstract**—Software architectures shift the focus of developers from lines-of-code to coarser-grained architectural elements and their overall interconnection structure. Architecture description languages (ADLs) have been proposed as modeling notations to support architecture-based development. There is, however, little consensus in the research community on what is an ADL, what aspects of an architecture should be modeled in an ADL, and which of several possible ADLs is best suited for a particular problem. Furthermore, the distinction is rarely made between ADLs on one hand and formal specification, module interconnection, simulation, and programming languages on the other. This paper attempts to provide an answer to these questions. It motivates and presents a definition and a classification framework for ADLs. The utility of the definition is demonstrated by using it to differentiate ADLs from other modeling notations. The framework is used to classify and compare several existing ADLs, enabling us, in the process, to identify key properties of ADLs. The comparison highlights areas where existing ADLs provide extensive support and those in which they are deficient, suggesting a research agenda for the future.

**Index Terms**—Software architecture, architecture description language, component, connector, configuration, definition, classification, comparison.

### 1 Introduction

Software architecture research is directed at reducing costs of developing applications and increasing the potential for commonality between different members of a closely related product family [54], [66]. Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements (software components and connectors) and their overall interconnection structure. To support architecture-based development, formal modeling notations and analysis and development tools that operate on architectural specifications are needed. Architecture description languages (ADLs) and their accompanying toolsets have been proposed as the answer. Loosely defined, "an ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module" [71]. ADLs have recently become an area of intense research in the software architecture community [11], [16], [73], [37].

A number of ADLs have been proposed for modeling architectures, both within a particular domain and as general-purpose architecture modeling languages. In this paper, we specifically consider those languages most commonly referred to as ADLs: Aesop [14], [12], ArTek [69], C2 [39], [42], Darwin [35], [36], LILEANNA [70], MetaH [6], [72], Rapide [31], [32], SADL [46], [47], UniCon [62], [63], Weaves [20], [21], and Wright [2], [4].[1] Recently, initial work has been done on an architecture interchange language, ACME [15], which is intended to support mapping of architectural specifications from one ADL to another and, hence, enable integration of support tools across ADLs. Although, strictly speaking, ACME is not an ADL, it contains a number of ADL-like features. Furthermore, it is useful to compare and differentiate it from other ADLs to highlight the difference between an ADL and an interchange language. It is therefore, included in this paper.

There is, however, still little consensus in the research community on what an ADL is, what aspects of an architecture should be modeled by an ADL, and what should be interchanged in an interchange language [43]. For example, Rapide may be characterized as a general-purpose system description language that allows modeling of component interfaces and their externally visible behavior, while Wright formalizes the semantics of architectural connections. Furthermore, the distinction is rarely made between ADLs on one hand and formal specification, module interconnection (MIL), simulation, and programming languages on the other. Indeed, for example, Rapide can be viewed as both an ADL and a simulation language, while Clements contends that CODE [49], a parallel programming language, is also an ADL [8].

Another source of discord is the level of support an ADL should provide to developers. At one end of the spectrum, it can be argued that the primary role of architectural descriptions is to aid understanding and communication about a software system. As such, an ADL must have

• *N. Medvidovic is with the Department of Computer Science, University of Southern California, Los Angeles, CA 90007-4313. E-mail: neno@usc.edu.*
• *R.N. Taylor is with the Department of Information and Computer Science, University of California, Irvine, CA 92697-3425. E-mail: taylor@uci.edu.*

*Manuscript received 13 Feb. 1998; revised 29 Dec. 1998; accepted 18 Feb. 1999.*
*Recommended for acceptance by M. Jazayeri.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 106346.*

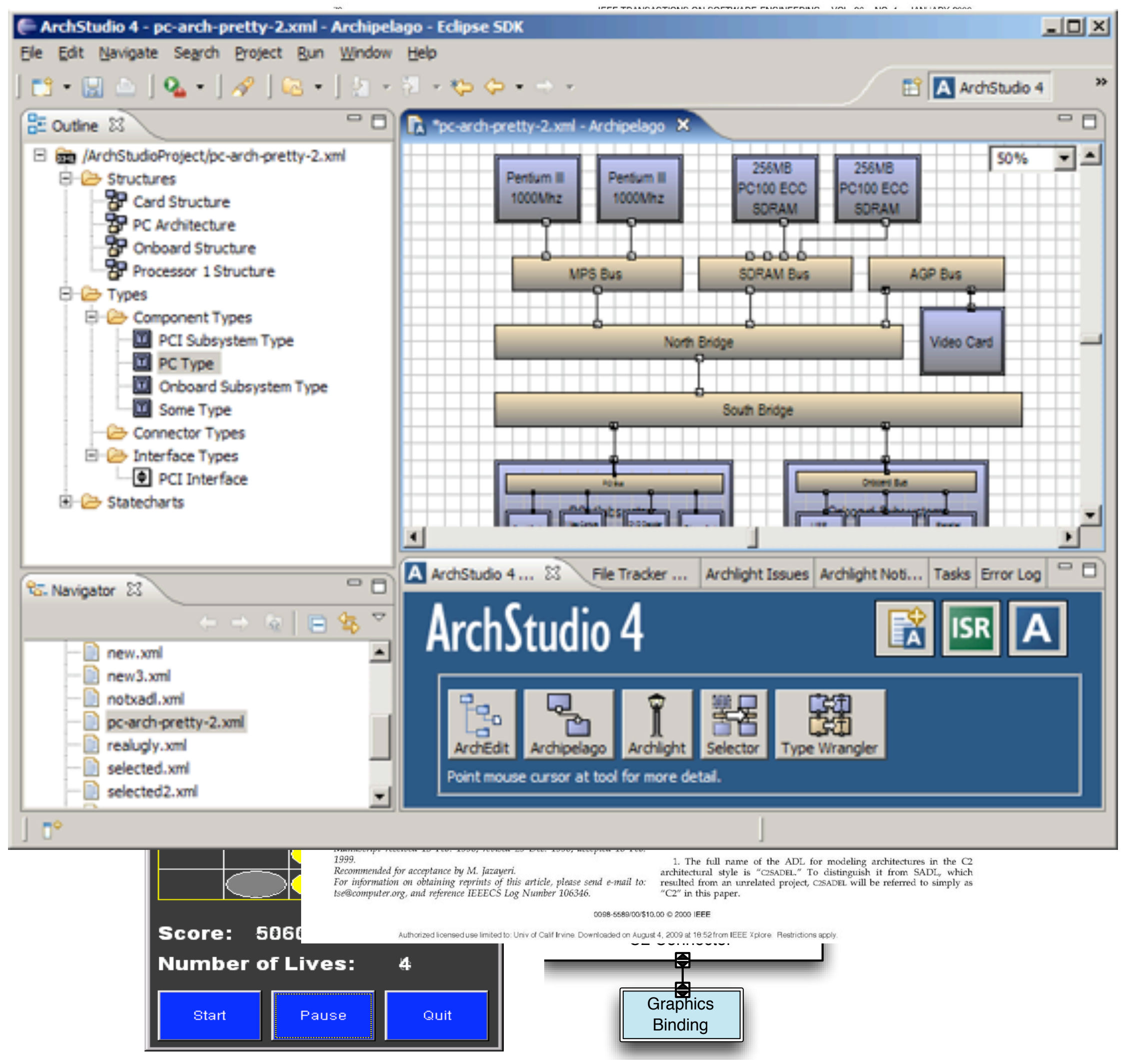
1. The full name of the ADL for modeling architectures in the C2 architectural style is "C2SADEL." To distinguish it from SADL, which resulted from an unrelated project, C2SADEL will be referred to simply as "C2" in this paper.

# Some of My Emphases

Structure & Style

Modeling (ADLs)

Support environments



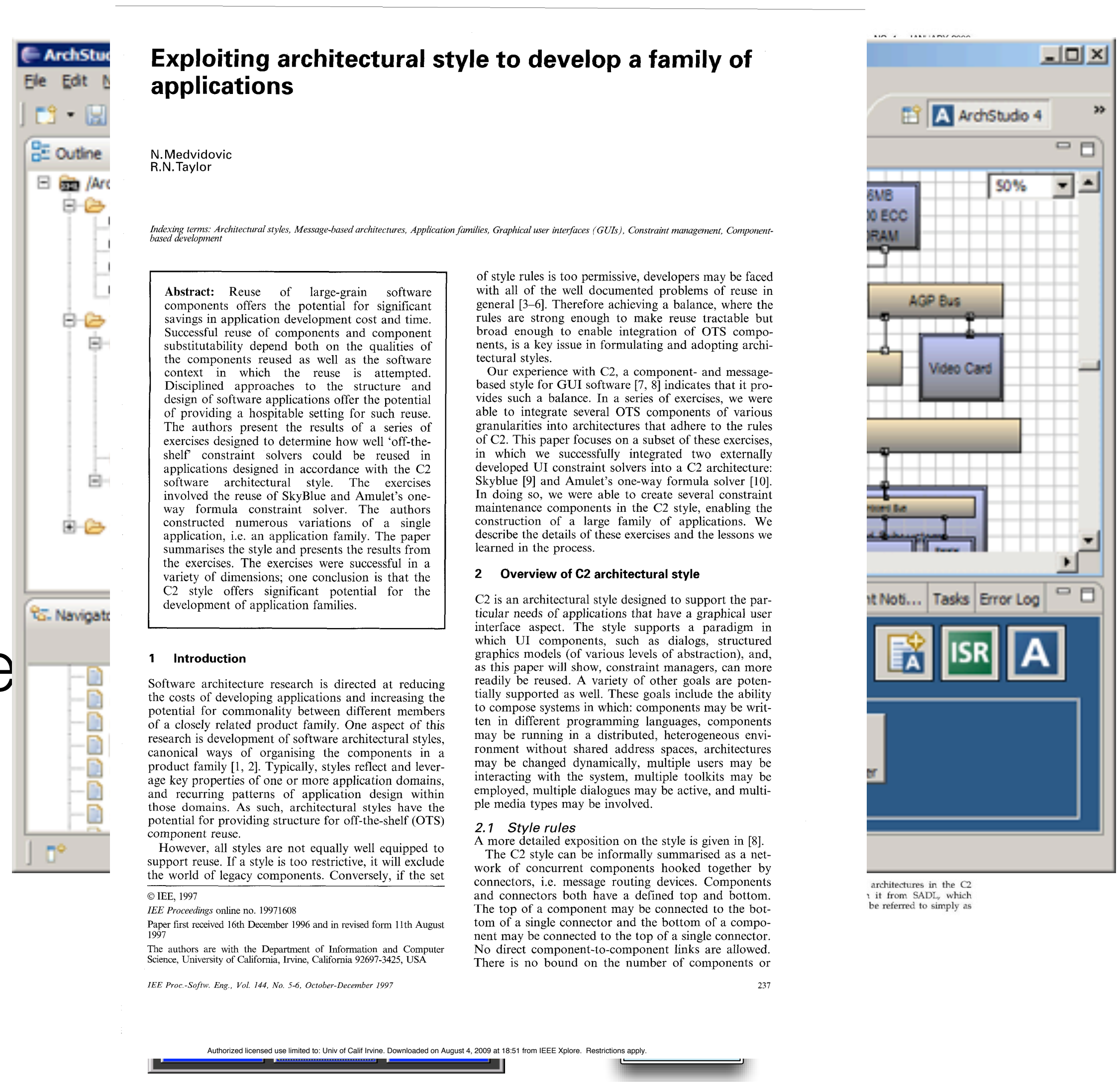
# Some of My Emphases

Structure & Style

Modeling (ADLs)

Support environments

Domain-specific software architectures




**Exploiting architectural style to develop a family of applications**

N.Medvidovic
R.N.Taylor

*Indexing terms: Architectural styles, Message-based architectures, Application families, Graphical user interfaces (GUIs), Constraint management, Component-based development*

**Abstract:** Reuse of large-grain software components offers the potential for significant savings in application development cost and time. Successful reuse of components and component substitutability depend both on the qualities of the components reused as well as the software context in which the reuse is attempted. Disciplined approaches to the structure and design of software applications offer the potential of providing a hospitable setting for such reuse. The authors present the results of a series of exercises designed to determine how well 'off-the-shelf' constraint solvers could be reused in applications designed in accordance with the C2 software architectural style. The exercises involved the reuse of SkyBlue and Amulet's one-way formula constraint solver. The authors constructed numerous variations of a single application, i.e. an application family. The paper summarises the style and presents the results from the exercises. The exercises were successful in a variety of dimensions; one conclusion is that the C2 style offers significant potential for the development of application families.

**1 Introduction**

Software architecture research is directed at reducing the costs of developing applications and increasing the potential for commonality between different members of a closely related product family. One aspect of this research is development of software architectural styles, canonical ways of organising the components in a product family [1, 2]. Typically, styles reflect and leverage key properties of one or more application domains, and recurring patterns of application design within those domains. As such, architectural styles have the potential for providing structure for off-the-shelf (OTS) component reuse.

However, all styles are not equally well equipped to support reuse. If a style is too restrictive, it will exclude the world of legacy components. Conversely, if the set of style rules is too permissive, developers may be faced with all of the well documented problems of reuse in general [3–6]. Therefore achieving a balance, where the rules are strong enough to make reuse tractable but broad enough to enable integration of OTS components, is a key issue in formulating and adopting architectural styles.

Our experience with C2, a component- and message-based style for GUI software [7, 8] indicates that it provides such a balance. In a series of exercises, we were able to integrate several OTS components of various granularities into architectures that adhere to the rules of C2. This paper focuses on a subset of these exercises, in which we successfully integrated two externally developed UI constraint solvers into a C2 architecture: Skyblue [9] and Amulet's one-way formula solver [10]. In doing so, we were able to create several constraint maintenance components in the C2 style, enabling the construction of a large family of applications. We describe the details of these exercises and the lessons we learned in the process.

**2 Overview of C2 architectural style**

C2 is an architectural style designed to support the particular needs of applications that have a graphical user interface aspect. The style supports a paradigm in which UI components, such as dialogs, structured graphics models (of various levels of abstraction), and, as this paper will show, constraint managers, can more readily be reused. A variety of other goals are potentially supported as well. These goals include the ability to compose systems in which: components may be written in different programming languages, components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple users may be interacting with the system, multiple toolkits may be employed, multiple dialogues may be active, and multiple media types may be involved.

*2.1 Style rules*

A more detailed exposition on the style is given in [8].

The C2 style can be informally summarised as a network of concurrent components hooked together by connectors, i.e. message routing devices. Components and connectors both have a defined top and bottom. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. No direct component-to-component links are allowed. There is no bound on the number of components or

© IEE, 1997

*IEE Proceedings* online no. 19971608

Paper first received 16th December 1996 and in revised form 11th August 1997

The authors are with the Department of Information and Computer Science, University of California, Irvine, California 92697-3425, USA

*IEE Proc.-Softw. Eng., Vol. 144, No. 5-6, October-December 1997* 237

Authorized licensed use limited to: Univ of Calif Irvine. Downloaded on August 4, 2009 at 18:51 from IEEE Xplore. Restrictions apply.

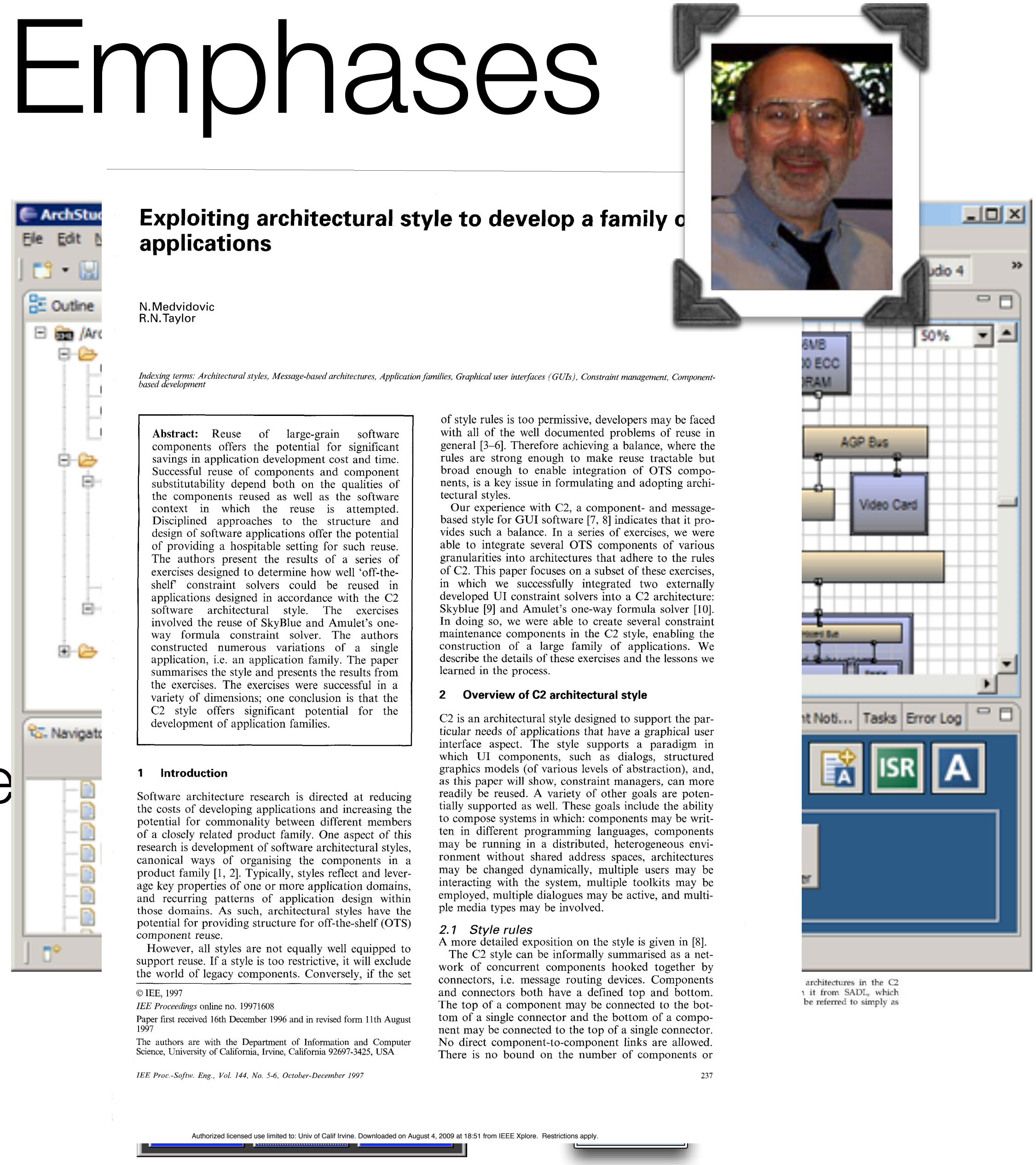
# Some of My Emphases

Structure & Style

Modeling (ADLs)

Support environments

Domain-specific software architectures




## Exploiting architectural style to develop a family o applications


N.Medvidovic
R.N.Taylor



*Indexing terms: Architectural styles, Message-based architectures, Application families, Graphical user interfaces (GUIs), Constraint management, Component-based development*



**Abstract:** Reuse of large-grain software components offers the potential for significant savings in application development cost and time. Successful reuse of components and component substitutability depend both on the qualities of the components reused as well as the software context in which the reuse is attempted. Disciplined approaches to the structure and design of software applications offer the potential of providing a hospitable setting for such reuse. The authors present the results of a series of exercises designed to determine how well 'off-the-shelf' constraint solvers could be reused in applications designed in accordance with the C2 software architectural style. The exercises involved the reuse of SkyBlue and Amulet's one-way formula constraint solver. The authors constructed numerous variations of a single application, i.e. an application family. The paper summarises the style and presents the results from the exercises. The exercises were successful in a variety of dimensions; one conclusion is that the C2 style offers significant potential for the development of application families.


### 1 Introduction

Software architecture research is directed at reducing the costs of developing applications and increasing the potential for commonality between different members of a closely related product family. One aspect of this research is development of software architectural styles, canonical ways of organising the components in a product family [1, 2]. Typically, styles reflect and leverage key properties of one or more application domains, and recurring patterns of application design within those domains. As such, architectural styles have the potential for providing structure for off-the-shelf (OTS) component reuse.

However, all styles are not equally well equipped to support reuse. If a style is too restrictive, it will exclude the world of legacy components. Conversely, if the set of style rules is too permissive, developers may be faced with all of the well documented problems of reuse in general [3–6]. Therefore achieving a balance, where the rules are strong enough to make reuse tractable but broad enough to enable integration of OTS components, is a key issue in formulating and adopting architectural styles.

Our experience with C2, a component- and message-based style for GUI software [7, 8] indicates that it provides such a balance. In a series of exercises, we were able to integrate several OTS components of various granularities into architectures that adhere to the rules of C2. This paper focuses on a subset of these exercises, in which we successfully integrated two externally developed UI constraint solvers into a C2 architecture: Skyblue [9] and Amulet's one-way formula solver [10]. In doing so, we were able to create several constraint maintenance components in the C2 style, enabling the construction of a large family of applications. We describe the details of these exercises and the lessons we learned in the process.

### 2 Overview of C2 architectural style

C2 is an architectural style designed to support the particular needs of applications that have a graphical user interface aspect. The style supports a paradigm in which UI components, such as dialogs, structured graphics models (of various levels of abstraction), and, as this paper will show, constraint managers, can more readily be reused. A variety of other goals are potentially supported as well. These goals include the ability to compose systems in which: components may be written in different programming languages, components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple users may be interacting with the system, multiple toolkits may be employed, multiple dialogues may be active, and multiple media types may be involved.

#### *2.1 Style rules*

A more detailed exposition on the style is given in [8].

The C2 style can be informally summarised as a network of concurrent components hooked together by connectors, i.e. message routing devices. Components and connectors both have a defined top and bottom. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. No direct component-to-component links are allowed. There is no bound on the number of components or




*IEE Proceedings* online no. 19971608

Paper first received 16th December 1996 and in revised form 11th August 1997



The authors are with the Department of Information and Computer Science, University of California, Irvine, California 92697-3425, USA

# Some of My Emphases

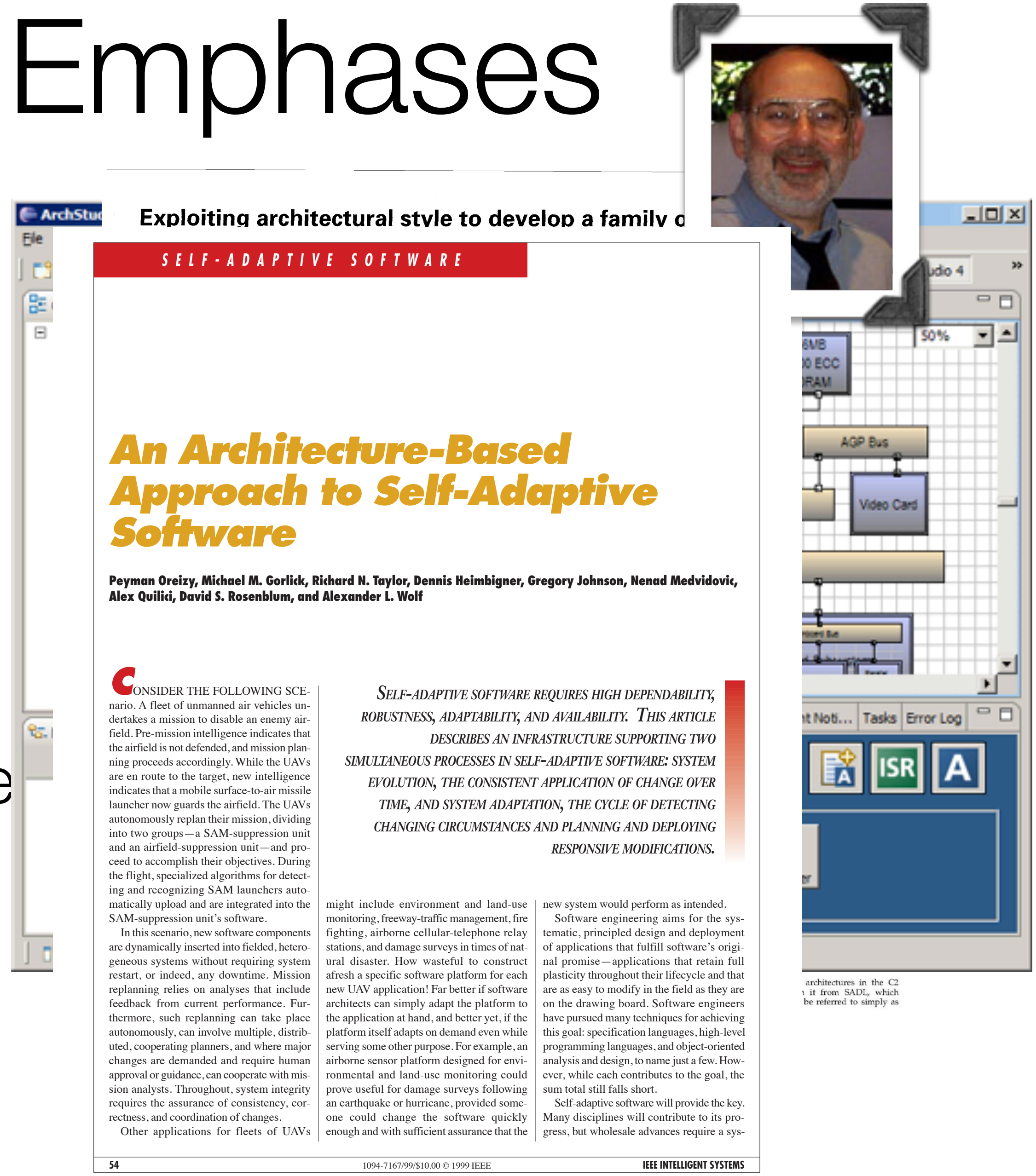
Structure & Style

Modeling (ADLs)

Support environments

Domain-specific software architectures

Dynamic adaptation





SELF-ADAPTIVE SOFTWARE

## An Architecture-Based Approach to Self-Adaptive Software

Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf

*SELF-ADAPTIVE SOFTWARE REQUIRES HIGH DEPENDABILITY, ROBUSTNESS, ADAPTABILITY, AND AVAILABILITY. THIS ARTICLE DESCRIBES AN INFRASTRUCTURE SUPPORTING TWO SIMULTANEOUS PROCESSES IN SELF-ADAPTIVE SOFTWARE: SYSTEM EVOLUTION, THE CONSISTENT APPLICATION OF CHANGE OVER TIME, AND SYSTEM ADAPTATION, THE CYCLE OF DETECTING CHANGING CIRCUMSTANCES AND PLANNING AND DEPLOYING RESPONSIVE MODIFICATIONS.*

CONSIDER THE FOLLOWING SCENARIO. A fleet of unmanned air vehicles undertakes a mission to disable an enemy airfield. Pre-mission intelligence indicates that the airfield is not defended, and mission planning proceeds accordingly. While the UAVs are en route to the target, new intelligence indicates that a mobile surface-to-air missile launcher now guards the airfield. The UAVs autonomously replan their mission, dividing into two groups—a SAM-suppression unit and an airfield-suppression unit—and proceed to accomplish their objectives. During the flight, specialized algorithms for detecting and recognizing SAM launchers automatically upload and are integrated into the SAM-suppression unit's software.

In this scenario, new software components are dynamically inserted into fielded, heterogeneous systems without requiring system restart, or indeed, any downtime. Mission replanning relies on analyses that include feedback from current performance. Furthermore, such replanning can take place autonomously, can involve multiple, distributed, cooperating planners, and where major changes are demanded and require human approval or guidance, can cooperate with mission analysts. Throughout, system integrity requires the assurance of consistency, correctness, and coordination of changes.

Other applications for fleets of UAVs might include environment and land-use monitoring, freeway-traffic management, fire fighting, airborne cellular-telephone relay stations, and damage surveys in times of natural disaster. How wasteful to construct afresh a specific software platform for each new UAV application! Far better if software architects can simply adapt the platform to the application at hand, and better yet, if the platform itself adapts on demand even while serving some other purpose. For example, an airborne sensor platform designed for environmental and land-use monitoring could prove useful for damage surveys following an earthquake or hurricane, provided someone could change the software quickly enough and with sufficient assurance that the new system would perform as intended.

Software engineering aims for the systematic, principled design and deployment of applications that fulfill software's original promise—applications that retain full plasticity throughout their lifecycle and that are as easy to modify in the field as they are on the drawing board. Software engineers have pursued many techniques for achieving this goal: specification languages, high-level programming languages, and object-oriented analysis and design, to name just a few. However, while each contributes to the goal, the sum total still falls short.

Self-adaptive software will provide the key. Many disciplines will contribute to its progress, but wholesale advances require a sys-

54 | 1094-7167/99/$10.00 © 1999 IEEE | IEEE INTELLIGENT SYSTEMS

# Architecture: A Matur(ing) Field

# Architecture: A Matur(ing) Field

Plenty of fuzzy-headed notions in the ether, however

Awareness, understanding, and uptake uneven

Hence a personal challenge:  draw the results together, present the field comprehensively, technically, deeply

# Architecture: A Matur(ing) Field

Plenty of fuzzy-headed notions in the ether, however

Awareness, understanding, and uptake uneven

Hence a personal challenge:  draw the results together, present the field comprehensively, technically, deeply

AND DON'T FORGET ABOUT THE SOURCE CODE!

# A Comprehensive Textbook

732 pages

Designing

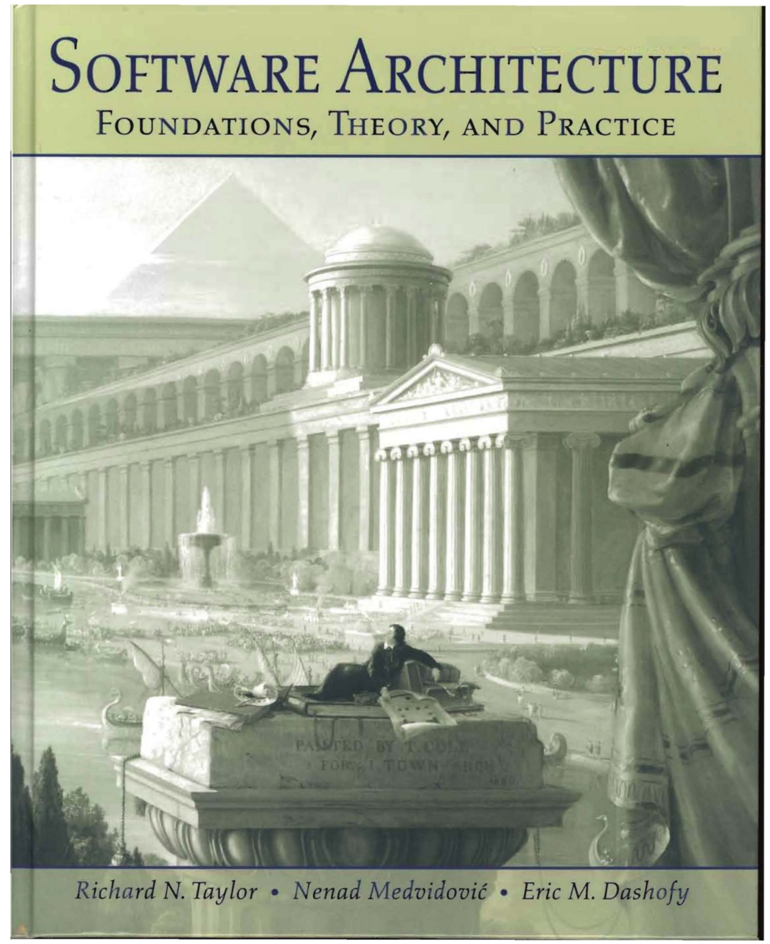
Modeling

Connectors

Implementation

Analysis

NFPs, Deployment, DSSAs, ...



# A Comprehensive Textbook

- 732 pages
- Designing
- Modeling
- Connectors
- Implementation
- Analysis
- NFPs, Deployment, DSSAs, ...

# But What *IS* Architecture?

# But What *IS* Architecture?

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

# But What *IS* Architecture?

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

Principal decisions

Whenever

Whoever

Wherever

# But What *IS* Architecture?

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

Principal decisions

Whenever

Whoever

Wherever

Encompasses structural, functional, UI decisions

Encompasses internal and external architecture

If Architecture is the set of principal design decisions - by whomever, wherever, whenever, then it is the proper central focus of software development.

When explicit, carries value not only for the present, but the system's future

# So What?

SA becomes the basis for long-term intellectual control

It is the focal point for ensuring conceptual integrity

It is an anchor for supporting reuse: of knowledge, experience, designs, and code

It is a centerpiece for effective project communication

It provides an adequate technical basis for development and management of related variant systems

# Opportunities and Challenges

# Opportunities and Challenges

- When explicit, carries value to all
  - But priorities vary, hence different models, visualizations, and tools needed
  - Interesting demands on tool support

# Opportunities and Challenges

- When explicit, carries value to all
  - But priorities vary, hence different models, visualizations, and tools needed
  - Interesting demands on tool support
- The special value of explicit connectors
  - as a way of thinking; as a way of building

# Opportunities and Challenges

- When explicit, carries value to all
  - But priorities vary, hence different models, visualizations, and tools needed
  - Interesting demands on tool support
- The special value of explicit connectors
  - as a way of thinking; as a way of building
- Going *to* code, and *with* code
  - demands support for implementation mapping, frameworks, generation, …
  - and hence provides basis for dynamic adaptation

# A Little Case Study

# A Little Case Study

Where have architectures *really* mattered?

# A Little Case Study

Where have architectures *really* mattered?

In what way have they actually mattered?

# A Little Case Study

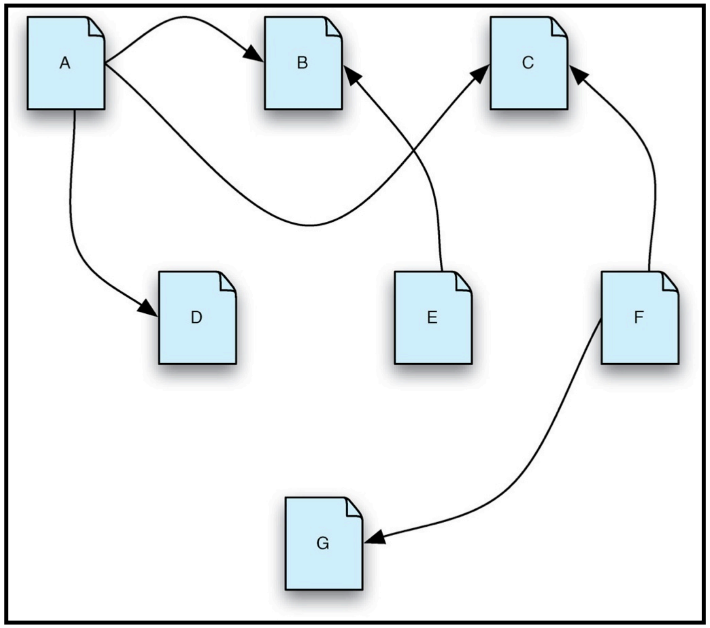
Where have architectures *really* mattered?

In what way have they actually mattered?

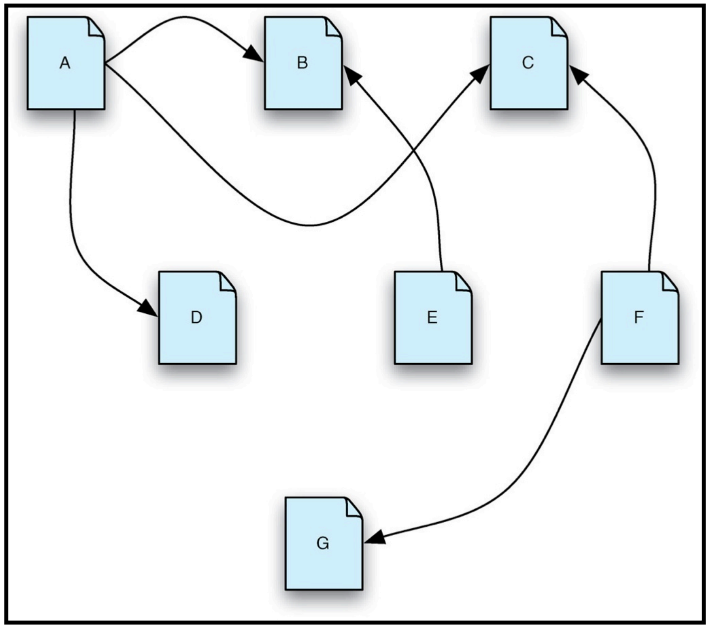
Case study: the World Wide Web

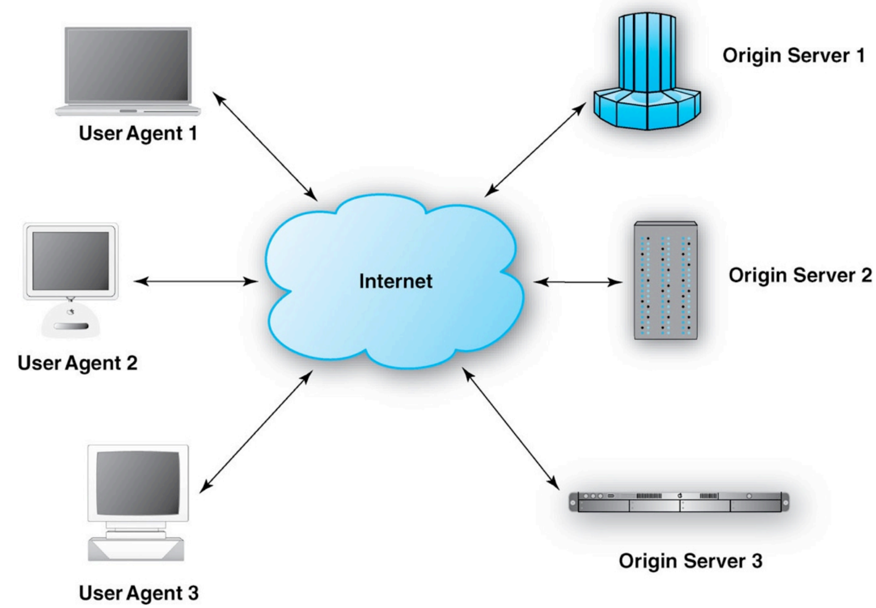
# What is the WWW?

# What is the WWW?

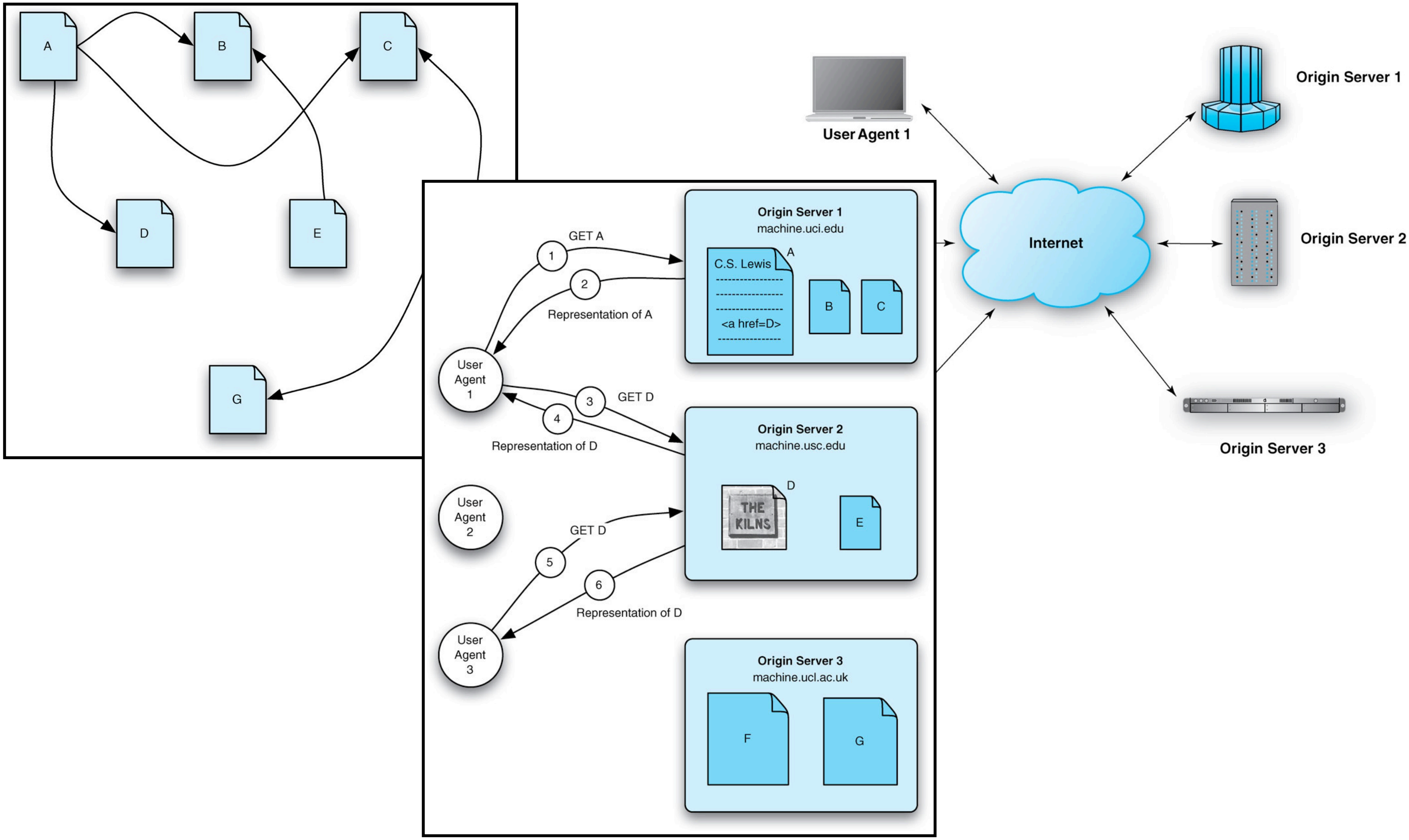


# What is the WWW?



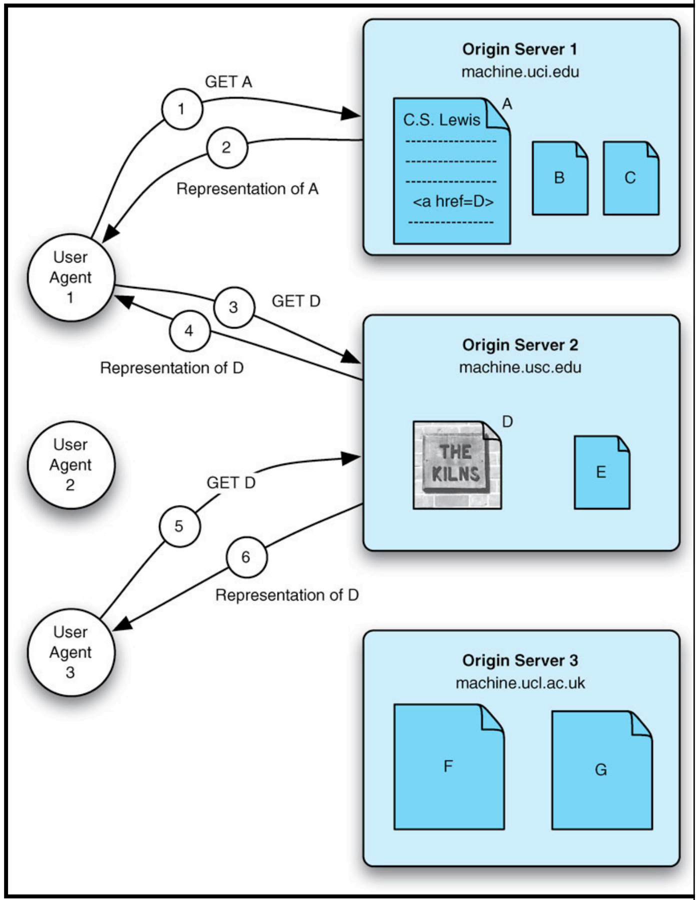



# What is the WWW?



# What is the WWW?

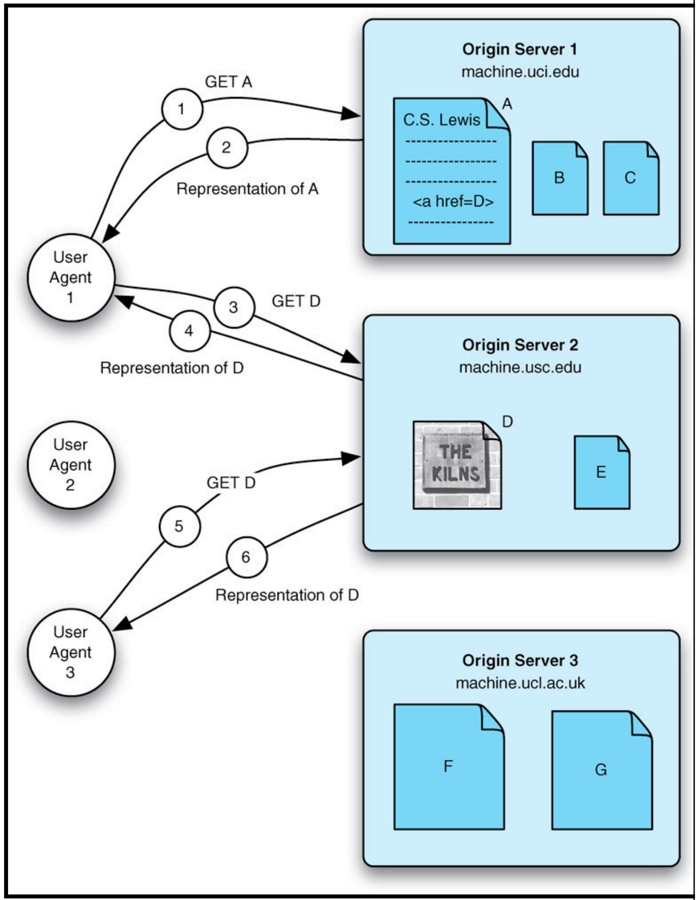
What is its structure “right now”?



# What is the WWW?

What is its structure “right now”?

What source code do I read?

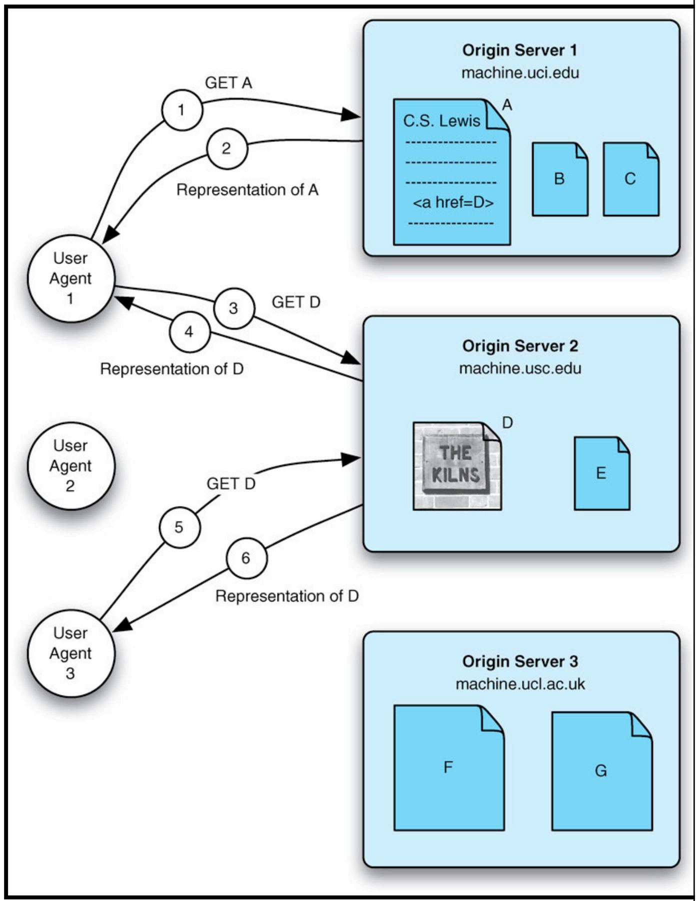


# What is the WWW?

What is its structure “right now”?

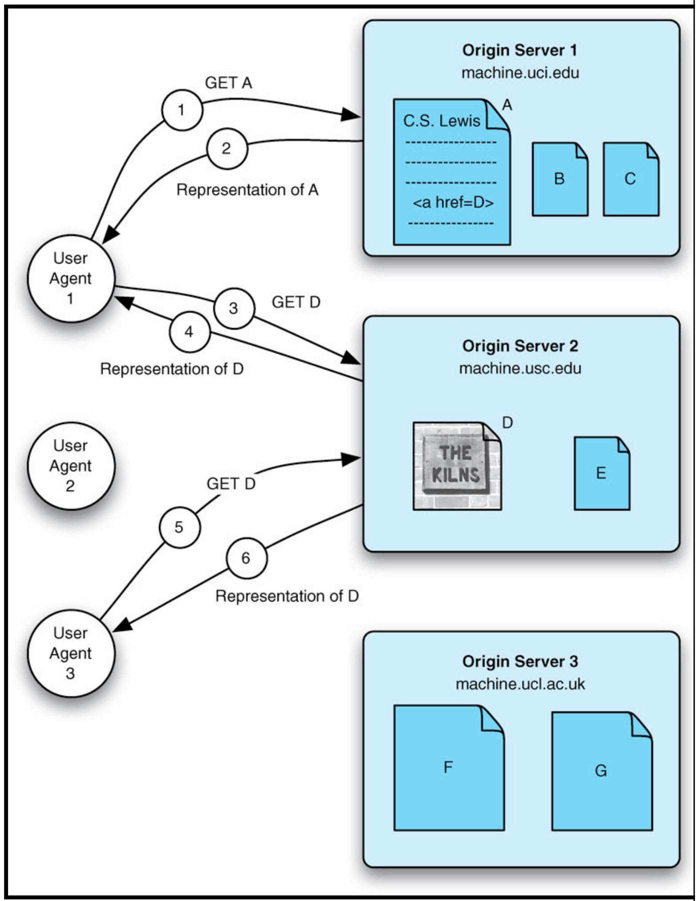
What source code do I read?

Who is in control?



# What is the WWW?

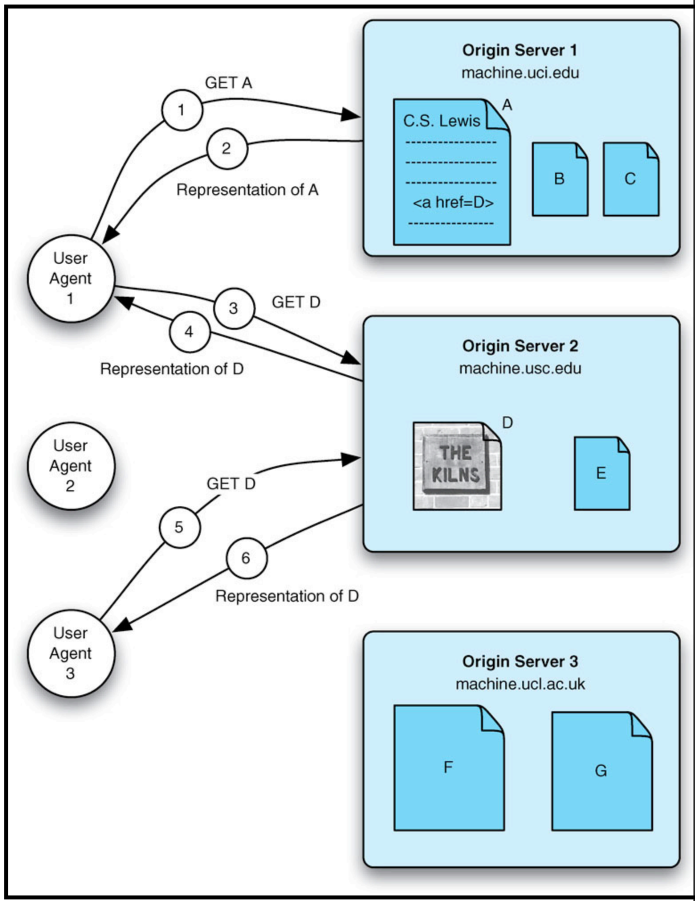
The architecture of the Web is wholly separate from the code that implements it



# What is the WWW?

The architecture of the Web is wholly separate from the code that implements it

There is no single piece of code that implements it

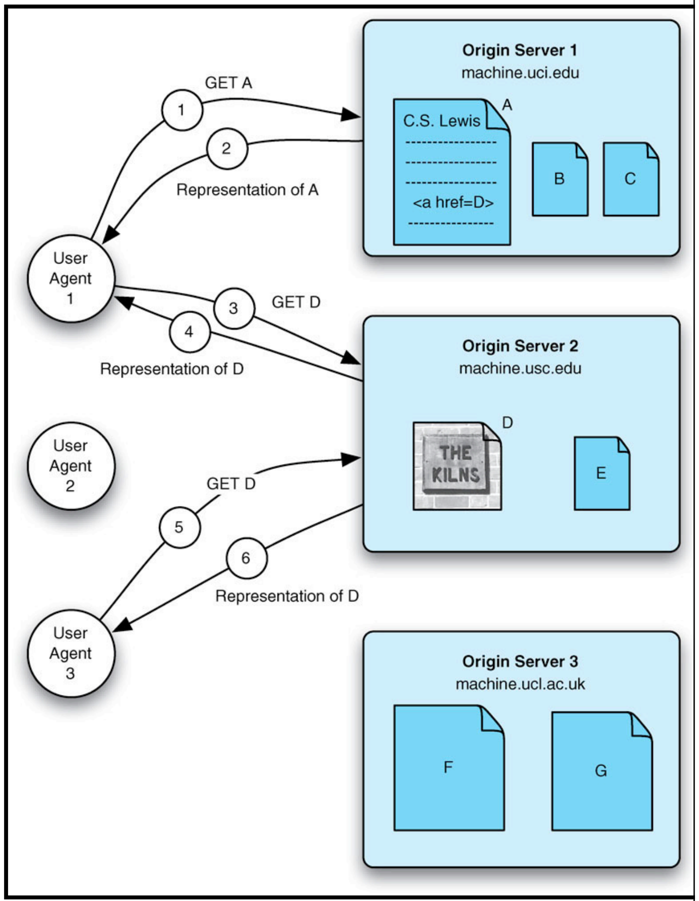


# What is the WWW?

The architecture of the Web is wholly separate from the code that implements it

There is no single piece of code that implements it

There are multiple equivalent pieces of code that implement various parts of the architecture



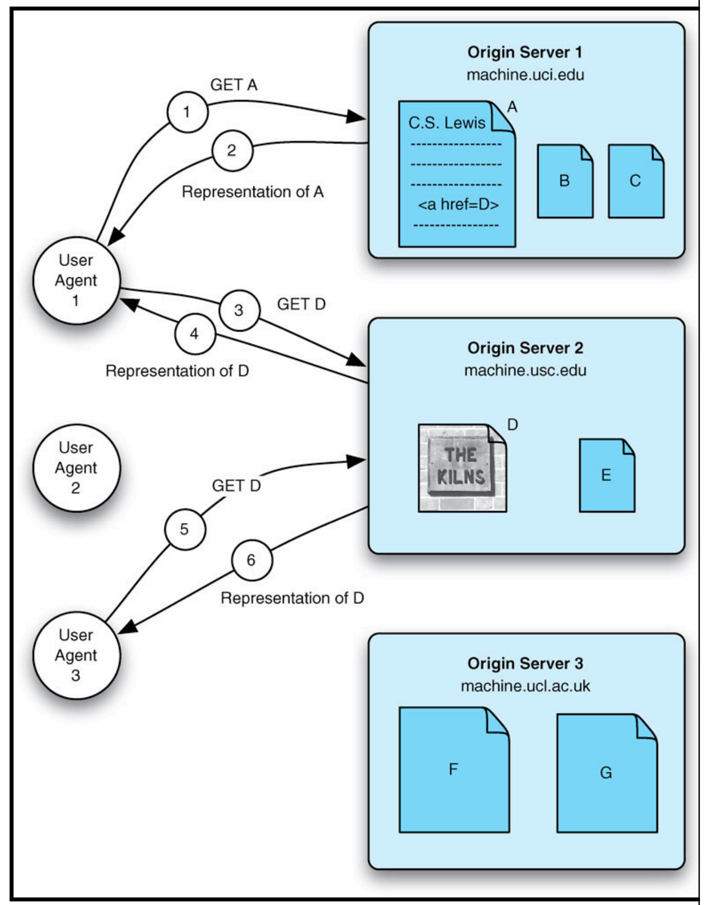
# What is the WWW?

The architecture of the Web is wholly separate from the code that implements it

There is no single piece of code that implements it

There are multiple equivalent pieces of code that implement various parts of the architecture

The stylistic constraints that constitute the Web's style are not readily apparent in the code



RFC 2616
HTTP/1.1

Network Working Group
Request for Comments: 2616
Obsoletes: 2068
Category: Standards Track

R. Fielding
UC Irvine
J. Gettys
Compaq/W3C
J. Mogul
Compaq
H. Frystyk
W3C/MIT
L. Masinter
Xerox
P. Leach
Microsoft
T. Berners-Lee
W3C/MIT
June 1999

Hypertext Transfer Protocol -- HTTP/1.1

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice



Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers [47]. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification defines the protocol referred to as "HTTP/1.1", and is an update to RFC 2068 [33].

RFC 2616
HTTP/1.1

Network Working Group
Request for Comments: 2616
Obsoletes: 2068
Category: Standards Track


R. Fielding
UC Irvine
J. Gettys
Compaq/W3C
J. Mogul
Compaq
H. Frystyk
W3C/MIT
L. Masinter
Xerox
P. Leach
Microsoft
T. Berners-Lee
W3C/MIT
June 1999


Hypertext Transfer Protocol -- HTTP/1.1

E this Memo

document specifies an Internet standards track protocol for the
net community, and requests discussion and suggestions for
vements. Please refer to the current edition of the "Internet
ial Protocol Standards" (STD 1) for the standardization state
tatus of this protocol. Distribution of this memo is unlimited.

: Notice



ypertext Transfer Protocol (HTTP) is an application-level
col for distributed, collaborative, hypermedia information
ns. It is a generic, stateless, protocol which can be used for
tasks beyond its use for hypertext, such as name servers and
ibuted object management systems, through extension of its
st methods, error codes and headers [47]. A feature of HTTP is
yping and negotiation of data representation, allowing systems
built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification defines the protocol referred to as "HTTP/1.1", and is an update to RFC 2068 [33].

RFC 1738
URL

Network Working Group
Request for Comments: 1738
Category: Standards Track


T. Berners-Lee
CERN
L. Masinter
Xerox Corporation
M. McCahill
University of Minnesota
Editors
December 1994


Uniform Resource Locators (URL)

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.


Abstract

This document specifies a Uniform Resource Locator (URL), the syntax and semantics of formalized information for location and access of resources via the Internet.


1. Introduction

This document describes the syntax and semantics for a compact string representation for a resource available via the Internet. These strings are called "Uniform Resource Locators" (URLs).

The specification is derived from concepts introduced by the World-Wide Web global information initiative, whose use of such objects dates from 1990 and is described in "Universal Resource Identifiers in WWW", RFC 1630. The specification of URLs is designed to meet the requirements laid out in "Functional Requirements for Internet Resource Locators" [12].

This document was written by the URI working group of the Internet Engineering Task Force. Comments may be addressed to the editors, or to the URI-WG <uri@bunyip.com>. Discussions of the group are archived at <URL:http://www.acl.lanl.gov/URI/archive/uri-archive.index.html>

RFC 2616
HTTP/1.1

Network Working Group
Request for Comments: 2616
Obsoletes: 2068
Category: Standards Track

R. Fielding
UC Irvine
J. Gettys

Network Working Group
Request for Comments: 1738
Category: Standards Track

T. Berners-Lee
CERN
L. Masinter
Xerox Corporation
M. McCahill
University of Minnesota
Editors
December 1994

Uniform Resource Locators (URL)

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Abstract

This document specifies a Uniform Resource Locator (URL), the syntax and semantics of formalized information for location and access of resources via the Internet.

1. Introduction

This document describes the syntax and semantics for a compact string representation for a resource available via the Internet. These strings are called "Uniform Resource Locators" (URLs).

The specification is derived from concepts introduced by the World-Wide Web global information initiative, whose use of such objects dates from 1990 and is described in "Universal Resource Identifiers in WWW", RFC 1630. The specification of URLs is designed to meet the requirements laid out in "Functional Requirements for Internet Resource Locators" [12].

This document was written by the URI working group of the Internet Engineering Task Force. Comments may be addressed to the editors, or to the URI-WG <uri@bunyip.com>. Discussions of the group are archived at <URL:http://www.acl.lanl.gov/URI/archive/uri-archive.index.html>

Berners-Lee, Masinter & McCahill [Page 1]

Hypertext Transfer Protoco

f this Memo

document specifies an Internet sta
net community, and requests discus
vements. Please refer to the curr
ial Protocol Standards" (STD 1) fo
tatus of this protocol. Distribut

t Notice

ight (C) The Internet Society (199

ypertext Transfer Protocol (HTTP)
col for distributed, collaborative
ns. It is a generic, stateless, pr
tasks beyond its use for hypertext
ibuted object management systems,
st methods, error codes and header
yping and negotiation of data repr
built independently of the data b

HTTP has been in use by the World-Wide
initiative since 1990. This specificati
referred to as "HTTP/1.1", and is an update to RFC 2068 [33].

Network Working Group
Request for Comments: 2396
Updates: 1808, 1738
Category: Standards Track

T. Berners-Lee
MIT/LCS
R. Fielding
U.C. Irvine
L. Masinter
Xerox Corporation
August 1998

Uniform Resource Identifiers (URI): Generic Syntax

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (1998). All Rights Reserved.

IESG Note

This paper describes a "superset" of operations that can be applied to URI. It consists of both a grammar and a description of basic functionality for URI. To understand what is a valid URI, both the grammar and the associated description have to be studied. Some of the functionality described is not applicable to all URI schemes, and some operations are only possible when certain media types are retrieved using the URI, regardless of the scheme used.

Abstract

A Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource. This document defines the generic syntax of URI, including both absolute and relative forms, and guidelines for their use; it revises and replaces the generic definitions in RFC 1738 and RFC 1808.

This document defines a grammar that is a superset of all valid URI, such that an implementation can parse the common components of a URI reference without knowing the scheme-specific requirements of every possible identifier type. This document does not define a generative grammar for URI; that task will be performed by the individual specifications of each URI scheme.

Berners-Lee, et. al. Standards Track [Page 1]

RFC 2396 URI Generic Syntax August 1998

RFC 1738
URL

RFC 2396
URI

RFC 2616
HTTP/1.1

```
Network Working Group                                    R. Fielding
Request for Comments: 2616                                 UC Irvine
Obsoletes: 2068                                            J. Gettys
Category: Standards Track
```

```
Network Working Group                                 T. Berners-Lee
Request for Comments: 2396                                   MIT/LCS
Updates: 1808, 1738                                       R. Fielding
Category: Standards Track                                 U.C. Irvine
                                                          L. Masinter
                                                    Xerox Corporation
                                                          August 1998

            Uniform Resource Identifiers (URI): Generic Syntax

Status of this Memo

   This document specifies an Internet standards track protocol for the
   Internet community, and requests discussion and suggestions for
```

```
Network Working Group                                 T. Berners-Lee
Request for Comments: 1738                                      CERN
Category: Standards Track                                L. Masinter
                                                    Xerox Corporation
                                                          M. McCahill
                                              University of Minnesota
                                                              Editors
                                                        December 1994

                    Uniform Resource Locators (URL)

Status of this Memo

   This do
   Interne
   improve
   Officia
   and sta

Abstract

   This do
   and ser
   resourc

1. Introdu

   This do
   represe
   strings are called "Uniform Resource Locators" (URLs).

   The specification is derived from concepts introduced by the World-
   Wide Web global information initiative, whose use of such objects
   dates from 1990 and is described in "Universal Resource Identifiers
   in WWW", RFC 1630. The specification of URLs is designed to meet the
   requirements laid out in "Functional Requirements for Internet
   Resource Locators" [12].

   This document was written by the URI working group of the Internet
   Engineering Task Force.  Comments may be addressed to the editors, or
   to the URI-WG <uri@bunyip.com>. Discussions of the group are archived
   at <URL:http://www.acl.lanl.gov/URI/archive/uri-archive.index.html>

Berners-Lee, Masinter & McCahill                                [Page 1]
```

Apache
HTTP SERVER PROJECT

```
 Notice

ight (C) The Internet Society (199

ypertext Transfer Protocol (HTTP)
col for distributed, collaborative
ns. It is a generic, stateless, pr
tasks beyond its use for hypertext
ibuted object management systems,
st methods, error codes and header
yping and negotiation of data repr
 built independently of the data b

HTTP has been in use by the World-Wide
initiative since 1990. This specificati
referred to as "HTTP/1.1", and is an update to RFC 2068 [33].
```

```
   functionality for URI.  To understand what is a valid URI, both the
   grammar and the associated description have to be studied.  Some of
   the functionality described is not applicable to all URI schemes, and
   some operations are only possible when certain media types are
   retrieved using the URI, regardless of the scheme used.

Abstract

   A Uniform Resource Identifier (URI) is a compact string of characters
   for identifying an abstract or physical resource.  This document
   defines the generic syntax of URI, including both absolute and
   relative forms, and guidelines for their use; it revises and replaces
   the generic definitions in RFC 1738 and RFC 1808.

   This document defines a grammar that is a superset of all valid URI,
   such that an implementation can parse the common components of a URI
   reference without knowing the scheme-specific requirements of every
   possible identifier type.  This document does not define a generative
   grammar for URI; that task will be performed by the individual
   specifications of each URI scheme.

Berners-Lee, et. al.        Standards Track                     [Page 1]

RFC 2396                   URI Generic Syntax                August 1998
```

RFC 1738
URL

RFC 2396
URI

ICSE 2000

```
Network Working Group                                    R. Fielding
Request for Comments: 2616                                 UC Irvine
```

# Principled Design of the Modern Web Architecture

**Roy T. Fielding and Richard N. Taylor**
Information and Computer Science
University of California, Irvine
Irvine, CA 92697–3425 USA
+1.949.824.4121
{fielding,taylor}@ics.uci.edu

**ABSTRACT**
The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an *Internet-scale* distributed hypermedia system. The modern Web architecture emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. In this paper, we introduce the Representational State Transfer (REST) architectural style, developed as an abstract model of the Web architecture to guide our redesign and definition of the Hypertext Transfer Protocol and Uniform Resource Identifiers. We describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles. We then compare the abstract model to the currently deployed Web architecture in order to elicit mismatches between the existing protocols and the applications they are intended to support.

**Keywords**
software architecture, software architectural style, WWW

## 1 INTRODUCTION

At the beginning of our efforts within the Internet Engineering Taskforce to define the existing Hypertext Transfer Protocol (HTTP/1.0) [5] and design the extensions for the new standards of HTTP/1.1 [10] and Uniform Resource Identifiers (URI) [6], we recognized the need for a model of how the World Wide Web (WWW, or simply Web) *should* work. This idealized model of the interactions within an overall Web application, what we refer to as the Representational State Transfer (REST) architectural style, became the foundation for the modern Web architecture, providing the guiding principles by which flaws in the preexisting architecture could be identified and extensions validated prior to deployment.



A software architecture determines how system elements are identified and allocated, how the elements interact to form a system, the amount and granularity of communication needed for interaction, and the interface protocols used for communication. An architectural style is an abstraction of the key aspects within a set of potential architectures (instantiations of the style), encapsulating important decisions about the architectural elements and emphasizing constraints on the elements and their relationships [17]. In other words, a style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to the style.

REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations. This is achieved by placing constraints on connector semantics where other styles have focused on component semantics. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, thereby meeting the needs of an Internet-scale distributed hypermedia system.

The modern Web is one instance of a REST-style architecture. Although Web-based applications can include access to other styles of interaction, the central focus of its protocol and performance concerns is distributed hypermedia. REST elaborates only those portions of the architecture that are considered essential for Internet-scale distributed hypermedia interaction. Areas for improvement of the Web architecture can be seen where existing protocols fail to express all of the potential semantics for component interaction, and where the details of syntax can be replaced with more efficient forms without changing the architecture capabilities. Likewise, proposed extensions can be compared to REST to see if they fit within the architecture; if not, it is more efficient to redirect that functionality to a system running in parallel with a more applicable architectural style.

This paper presents REST after the completion of six years' work on architectural standards for the modern (post-1993) Web. It does not present the details of the architecture itself, since those are found within the standards. Instead, we focus



```
ack

nternet Society (199

fer Protocol (HTTP)
buted, collaborative
neric, stateless, pr
s use for hypertext
management systems,
ror codes and header
ciation of data repr
dently of the data b

e by the World-Wide
90. This specificati
?/1.1", and is an update to RFC 2068 [33].
```

```
Network Working Group                                     T. Berners-Lee
Request for Comments: 2396                                       MIT/LCS
Updates: 1808, 1738                                           R. Fielding
Category: Standards Track                                     U.C. Irvine
                                                              L. Masinter
                                                        Xerox Corporation
                                                              August 1998


             Uniform Resource Identifiers (URI): Generic Syntax

Status of this Memo

   This document specifies an Internet standards track protocol for the
```

Apache
SERVER PROJECT

```
   functionality for URI.  To understand what is a valid URI, both the
   grammar and the associated description have to be studied.  Some of
   the functionality described is not applicable to all URI schemes, and
   some operations are only possible when certain media types are
   retrieved using the URI, regardless of the scheme used.

Abstract

   A Uniform Resource Identifier (URI) is a compact string of characters
   for identifying an abstract or physical resource.  This document
   defines the generic syntax of URI, including both absolute and
   relative forms, and guidelines for their use; it revises and replaces
   the generic definitions in RFC 1738 and RFC 1808.

   This document defines a grammar that is a superset of all valid URI,
   such that an implementation can parse the common components of a URI
   reference without knowing the scheme-specific requirements of every
   possible identifier type.  This document does not define a generative
   grammar for URI; that task will be performed by the individual
   specifications of each URI scheme.


Berners-Lee, et. al.        Standards Track                     [Page 1]

RFC 2396                   URI Generic Syntax                August 1998
```

ICSE 2000

# Principled Design of the Modern Web Architecture


**Roy T. Fielding and Richard N. Taylor**
Information and Computer Science
University of California, Irvine
Irvine, CA 92697–3425 USA
+1.949.824.4121
{fielding,taylor}@ics.uci.edu



**ABSTRACT**
The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an *Internet-scale* distributed hypermedia system. The modern Web architecture emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. In this paper, we introduce the Representational State Transfer (REST) architectural style, developed as an abstract model of the Web architecture to guide our redesign and definition of the Hypertext Transfer Protocol and Uniform Resource Identifiers. We describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles. We then compare the abstract model to the currently deployed Web architecture in order to elicit mismatches between the existing protocols and the applications they are intended to support.



**Keywords**
software architecture, software architectural style, WWW


**1 INTRODUCTION**
At the beginning of our efforts within the Internet Engineering Taskforce to define the existing Hypertext Transfer Protocol (HTTP/1.0) [5] and design the extensions for the new standards of HTTP/1.1 [10] and Uniform Resource Identifiers (URI) [6], we recognized the need for a model of how the World Wide Web (WWW, or simply Web) *should* work. This idealized model of the interactions within an overall Web application, what we refer to as the Representational State Transfer (REST) architectural style, became the foundation for the modern Web architecture, providing the guiding principles by which flaws in the preexisting architecture could be identified and extensions validated prior to deployment.



A software architecture determines how system elements are identified and allocated, how the elements interact to form a system, the amount and granularity of communication needed for interaction, and the interface protocols used for communication. An architectural style is an abstraction of the key aspects within a set of potential architectures (instantiations of the style), encapsulating important decisions about the architectural elements and emphasizing constraints on the elements and their relationships [17]. In other words, a style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to the style.

REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations. This is achieved by placing constraints on connector semantics where other styles have focused on component semantics. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, thereby meeting the needs of an Internet-scale distributed hypermedia system.

The modern Web is one instance of a REST-style architecture. Although Web-based applications can include access to other styles of interaction, the central focus of its protocol and performance concerns is distributed hypermedia. REST elaborates only those portions of the architecture that are considered essential for Internet-scale distributed hypermedia interaction. Areas for improvement of the Web architecture can be seen where existing protocols fail to express all of the potential semantics for component interaction, and where the details of syntax can be replaced with more efficient forms without changing the architecture capabilities. Likewise, proposed extensions can be compared to REST to see if they fit within the architecture; if not, it is more efficient to redirect that functionality to a system running in parallel with a more applicable architectural style.

This paper presents REST after the completion of six years' work on architectural standards for the modern (post-1993) Web. It does not present the details of the architecture itself, since those are found within the standards. Instead, we focus



```
Network Working Group                                    R. Fielding
Request for Comments: 2616
ack

Network
Request
Updates:
Category

Status o
   This
   Inte

Apach
SERVER PR

   funct
   gramm
   the
   some
   retri
nternet Society (199
Abstract
   A Un
   for
   defi
   rela
   the
fer Protocol (HTTP)
buted, collaborative
neric, stateless, pr
s use for hypertext
management systems,
ror codes and header
ciation of data repr
dently of the data b
   This
   such
   refe
   poss
   gramm
   spec

e by the World-Wide
90. This specificati
Berners-
RFC 2396                URI Generic Syntax              August 1998
?/1.1", and is an update to RFC 2068 [33].
```

# Principled Design of the Modern Web Architecture


ROY T. FIELDING
Day Software
and
RICHARD N. TAYLOR
University of California, Irvine



The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an Internet-scale distributed hypermedia application. The modern Web architecture emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. In this article we introduce the Representational State Transfer (REST) architectural style, developed as an abstract model of the Web architecture and used to guide our redesign and definition of the Hypertext Transfer Protocol and Uniform Resource Identifiers. We describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles. We then compare the abstract model to the currently deployed Web architecture in order to elicit mismatches between the existing protocols and the applications they are intended to support.



Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures; H.5.4 [**Hypertext/Hypermedia**]: Architectures; H.3.5 [**Information Storage and Retrieval**]: On-line Information Services—*Web-based services*

General Terms: Design, Performance, Standardization

Additional Key Words and Phrases: Network-based applications, REST, World Wide Web

This work was partially supported by the Defense Advanced Research Projects Agency and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021, and originated while the first author was at the University of California, Irvine.
An earlier version of this paper appeared in the *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, June 2000 (ICSE 2000), 407–416.
Authors' addresses: R. T. Fielding, Day Software, 2 Corporate Plaza, Suite 150, Newport Beach, CA 92660; email: roy.fielding@day.com; R. N. Taylor, Information and Computer Science, University of California, Irvine CA 92697-3425; email: taylor@ics.uci.edu.





ACM Transactions on Internet Technology, Vol. 2, No. 2, May 2002, Pages 115–150.

ICSE 2000

Network Working Group R. Fielding
Request for Comments: 2616

**Principled Design of the**

**Roy T. Fielding an**
Information and
University of
Irvine, CA 92
+1.949
{fielding,tayl

**ABSTRACT**
The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an *Internet-scale* distributed hypermedia system. The modern Web architecture emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. In this paper, we introduce the Representational State Transfer (REST) architectural style, developed as an abstract model of the Web architecture to guide our redesign and definition of the Hypertext Transfer Protocol and Uniform Resource Identifiers. We describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles. We then compare the abstract model to the currently deployed Web architecture in order to elicit mismatches between the existing protocols and the applications they are intended to support.

**Keywords**
software architecture, software architectural style, WWW

**1 INTRODUCTION**
At the beginning of our efforts within the Internet Engineering Taskforce to define the existing Hypertext Transfer Protocol (HTTP/1.0) [5] and design the extensions for the new standards of HTTP/1.1 [10] and Uniform Resource Identifiers (URI) [6], we recognized the need for a model of how the World Wide Web (WWW, or simply Web) *should* work. This idealized model of the interactions within an overall Web application, what we refer to as the Representational State Transfer (REST) architectural style, became the foundation for the modern Web architecture, providing the guiding principles by which flaws in the preexisting architecture could be identified and extensions validated prior to deployment.



UNIVERSITY OF CALIFORNIA,
IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

2000

Design of the Modern
ecture

nia, Irvine

as succeeded in large part because its software architecture has been de-
ds of an Internet-scale distributed hypermedia application. The modern
asizes scalability of component interactions, generality of interfaces, inde-
components, and intermediary components to reduce interaction latency,
capsulate legacy systems. In this article we introduce the Representational
architectural style, developed as an abstract model of the Web architecture
redesign and definition of the Hypertext Transfer Protocol and Uniform
describe the software engineering principles guiding REST and the inter-
en to retain those principles, contrasting them to the constraints of other
then compare the abstract model to the currently deployed Web architec-
mismatches between the existing protocols and the applications they are

Descriptors: D.2.11 [**Software Engineering**]: Software Architectures;
**ermedia**]: Architectures; H.3.5 [**Information Storage and Retrieval**]:
rvices—*Web-based services*

Performance, Standardization

nd Phrases: Network-based applications, REST, World Wide Web

supported by the Defense Advanced Research Projects Agency and Air Force
ir Force Materiel Command, USAF, under agreement number F30602-97-
while the first author was at the University of California, Irvine.
is paper appeared in the *Proceedings of the 22nd International Conference*
g, Limerick, Ireland, June 2000 (ICSE 2000), 407–416.
Fielding, Day Software, 2 Corporate Plaza, Suite 150, Newport Beach, CA
g@day.com; R. N. Taylor, Information and Computer Science, University of
2697-3425; email: taylor@ics.uci.edu.
ital or hard copies of part or all of this work for personal or classroom use is
vided that copies are not made or distributed for profit or direct commercial
pies show this notice on the first page or initial screen of a display along
opyrights for components of this work owned by others than ACM must be
ith credit is permitted. To copy otherwise, to republish, to post on servers,
or to use any component of this work in other works, requires prior specific
. Permissions may be requested from Publications Dept., ACM Inc., 1515
Y 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org.
0/02/0500–0115 $5.00

ACM Transactions on Internet Technology, Vol. 2, No. 2, May 2002, Pages 115–150.

eric Syntax August 1998

ICSE 2000

Network Working Group R. Fielding
Request for Comments: 2616

**Principled Design of the**


**Roy T. Fielding an**
Information and
University of
Irvine, CA 92
+1.949
{fielding,tayl



**ABSTRACT**
The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an *Internet-scale* distributed hypermedia system. The modern Web architecture emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. In this paper, we introduce the Representational State Transfer (REST) architectural style, developed as an abstract model of the Web architecture to guide our redesign and definition of the Hypertext Transfer Protocol and Uniform Resource Identifiers. We describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles. We then compare the abstract model to the currently deployed Web architecture in order to elicit mismatches between the existing protocols and the applications they are intended to support.



**Keywords**
software architecture, software architectural style, WWW


**1 INTRODUCTION**
At the beginning of our efforts within the Internet Engineering Taskforce to define the existing Hypertext Transfer Protocol (HTTP/1.0) [5] and design the extensions for the new standards of HTTP/1.1 [10] and Uniform Resource Identifiers (URI) [6], we recognized the need for a model of how the World Wide Web (WWW, or simply Web) *should* work. This idealized model of the interactions within an overall Web application, what we refer to as the Representational State Transfer (REST) architectural style, became the foundation for the modern Web architecture, providing the guiding principles by which flaws in the preexisting architecture could be identified and extensions validated prior to deployment.



# CHAPTER 5
# Representational State Transfer (REST)

This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles. REST is a hybrid style derived from several of the network-based architectural styles described in Chapter 3 and combined with additional constraints that define a uniform connector interface. The software architecture framework of Chapter 1 is used to define the architectural elements of REST and examine sample process, connector, and data views of prototypical architectures.

## 5.1 Deriving REST

The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture. By examining the impact of each constraint as it is added to the evolving style, we can identify the properties induced by the Web's constraints. Additional constraints can then be applied to form a new architectural style that better reflects the desired properties of a modern Web architecture. This section provides a general overview of REST by walking through the process of deriving it as an architectural style. Later sections will describe in more detail the specific constraints that compose the REST style.



sign of the Modern
ture

Irvine

cceeded in large part because its software architecture has been de- an Internet-scale distributed hypermedia application. The modern s scalability of component interactions, generality of interfaces, inde- ponents, and intermediary components to reduce interaction latency, late legacy systems. In this article we introduce the Representational tectural style, developed as an abstract model of the Web architecture sign and definition of the Hypertext Transfer Protocol and Uniform ribe the software engineering principles guiding REST and the inter- retain those principles, contrasting them to the constraints of other compare the abstract model to the currently deployed Web architec- atches between the existing protocols and the applications they are

criptors: D.2.11 [**Software Engineering**]: Software Architectures; **edia**]: Architectures; H.3.5 [**Information Storage and Retrieval**]: *—Web-based services*
ormance, Standardization
hrases: Network-based applications, REST, World Wide Web

orted by the Defense Advanced Research Projects Agency and Air Force rce Materiel Command, USAF, under agreement number F30602-97- the first author was at the University of California, Irvine.
per appeared in the *Proceedings of the 22nd International Conference* merick, Ireland, June 2000 (ICSE 2000), 407–416.
lding, Day Software, 2 Corporate Plaza, Suite 150, Newport Beach, CA y.com; R. N. Taylor, Information and Computer Science, University of 3425; email: taylor@ics.uci.edu.
r hard copies of part or all of this work for personal or classroom use is that copies are not made or distributed for profit or direct commercial how this notice on the first page or initial screen of a display along ghts for components of this work owned by others than ACM must be redit is permitted. To copy otherwise, to republish, to post on servers, use any component of this work in other works, requires prior specific missions may be requested from Publications Dept., ACM Inc., 1515 036 USA, fax +1 (212) 869-0481, or permissions@acm.org.
0500–0115 $5.00
Transactions on Internet Technology, Vol. 2, No. 2, May 2002, Pages 115–150.

Syntax August 1998

ICSE 2000

**Principled Design of the**

**Roy T. Fielding an**
Information and
University of
Irvine, CA 92
+1.949
{fielding,tayl

**ABSTRACT**
The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an *Internet-scale* distributed hypermedia system. The modern Web architecture emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. In this paper, we introduce the Representational State Transfer (REST) architectural style, developed as an abstract model of the Web architecture to guide our redesign and definition of the Hypertext Transfer Protocol and Uniform Resource Identifiers. We describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles. We then compare the abstract model to the currently deployed Web architecture in order to elicit mismatches between the existing protocols and the applications they are intended to support.

**Keywords**
software architecture, software architectural style, WWW

**1 INTRODUCTION**
At the beginning of our efforts within the Internet Engineering Taskforce to define the existing Hypertext Transfer Protocol (HTTP/1.0) [5] and design the extensions for the new standards of HTTP/1.1 [10] and Uniform Resource Identifiers (URI) [6], we recognized the need for a model of how the World Wide Web (WWW, or simply Web) *should* work. This idealized model of the interactions within an overall Web application, what we refer to as the Representational State Transfer (REST) architectural style, became the foundation for the modern Web architecture, providing the guiding principles by which flaws in the preexisting architecture could be identified and extensions validated prior to deployment.



Network Working Group
Request for Comments: 2616

**CHAPT**

**Representational State**

This chapter introduces and elaborates the R

architectural style for distributed hypermedia sys

principles guiding REST and the interaction con

while contrasting them to the constraints of oth

style derived from several of the network-based a

and combined with additional constraints that d

software architecture framework of Chapter 1 is

of REST and examine sample process, conn

architectures.

## 5.1 Deriving REST

The design rationale behind the Web architectu

style consisting of the set of constraints applied

examining the impact of each constraint as it

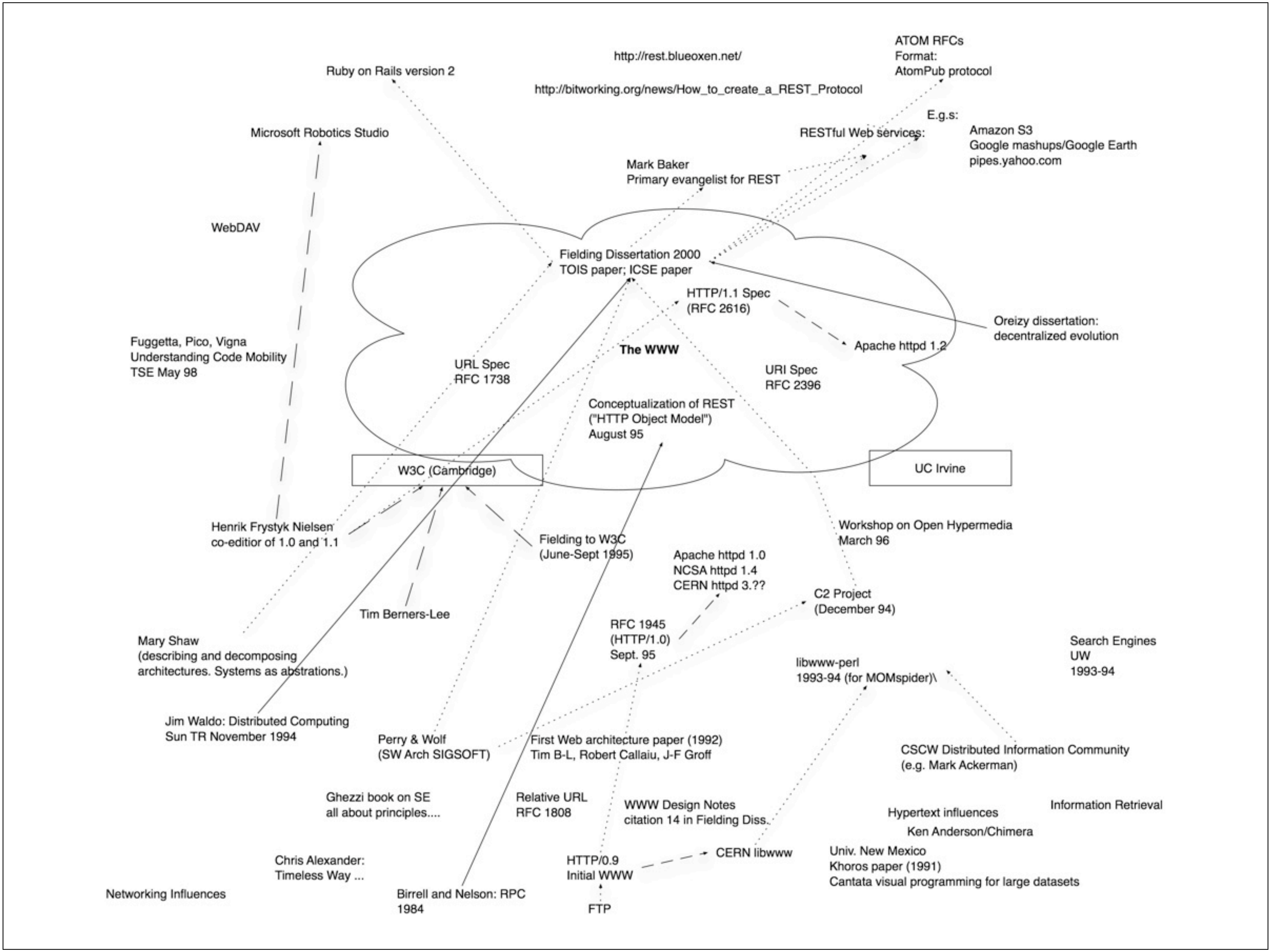
identify the properties induced by the Web's co

be applied to form a new architectural style that

modern Web architecture. This section provides

through the process of deriving it as an architect

more detail the specific constraints that compose the REST style.



ATOM RFCs
Format:
AtomPub protocol
http://rest.blueoxen.net/
Ruby on Rails version 2
http://bitworking.org/news/How_to_create_a_REST_Protocol
E.g.s:
Microsoft Robotics Studio
RESTful Web services:
Amazon S3
Google mashups/Google Earth
pipes.yahoo.com
Mark Baker
Primary evangelist for REST
WebDAV
Fielding Dissertation 2000
TOIS paper; ICSE paper
HTTP/1.1 Spec
(RFC 2616)
Oreizy dissertation:
decentralized evolution
Fuggetta, Pico, Vigna
Understanding Code Mobility
TSE May 98
The WWW
Apache httpd 1.2
URL Spec
RFC 1738
URI Spec
RFC 2396
Conceptualization of REST
("HTTP Object Model")
August 95
W3C (Cambridge)
UC Irvine
Henrik Frystyk Nielsen
co-editior of 1.0 and 1.1
Fielding to W3C
(June-Sept 1995)
Workshop on Open Hypermedia
March 96
Apache httpd 1.0
NCSA httpd 1.4
CERN httpd 3.??
C2 Project
(December 94)
Tim Berners-Lee
RFC 1945
(HTTP/1.0)
Sept. 95
Mary Shaw
(describing and decomposing
architectures. Systems as abstrations.)
Search Engines
UW
1993-94
libwww-perl
1993-94 (for MOMspider)\
Jim Waldo: Distributed Computing
Sun TR November 1994
Perry & Wolf
(SW Arch SIGSOFT)
First Web architecture paper (1992)
Tim B-L, Robert Callaiu, J-F Groff
CSCW Distributed Information Community
(e.g. Mark Ackerman)
Ghezzi book on SE
all about principles....
Relative URL
RFC 1808
WWW Design Notes
citation 14 in Fielding Diss.
Hypertext influences
Information Retrieval
Ken Anderson/Chimera
Chris Alexander:
Timeless Way ...
HTTP/0.9
Initial WWW
CERN libwww
Univ. New Mexico
Khoros paper (1991)
Cantata visual programming for large datasets
Networking Influences
Birrell and Nelson: RPC
1984
FTP

# Themes

# Themes

- Technical disciplines having an influence
  - Networking
  - Hypertext
    - Open hypermedia
  - Software architecture
  - Information retrieval

# Themes

- Technical disciplines having an influence
  - Networking
  - Hypertext
    - Open hypermedia
  - Software architecture
  - Information retrieval
- The movement of people

# Themes

- Technical disciplines having an influence
  - Networking
  - Hypertext
    - Open hypermedia
  - Software architecture
  - Information retrieval
- The movement of people

# Themes

Technical disciplines having an influence

- Networking
- Hypertext
  - Open hypermedia
- Software architecture
- Information retrieval

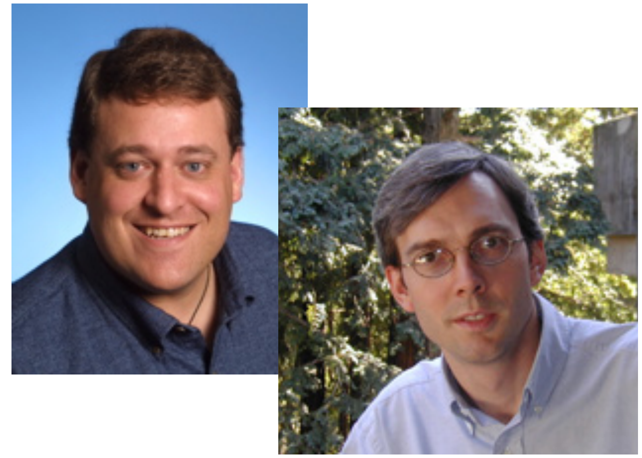
The movement of people

"When I ask you a question, you give me the right answer" -- Roy Fielding to Dick Taylor, 1994

# Themes

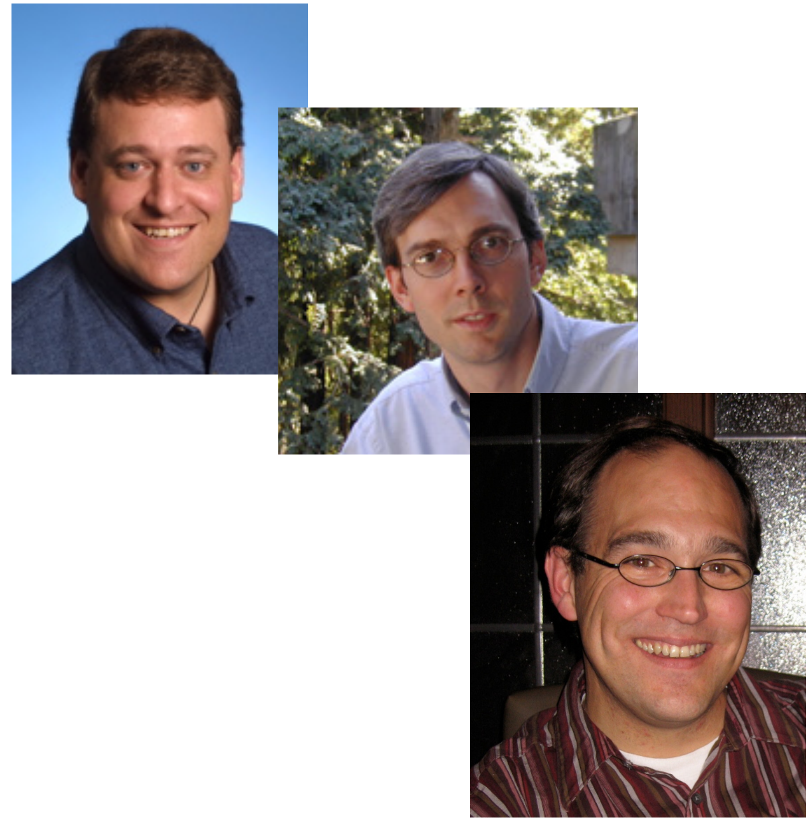
- Technical disciplines having an influence
  - Networking
  - Hypertext
    - Open hypermedia
  - Software architecture
  - Information retrieval
- The movement of people
- The social milieu Roy was in

# Themes



- Technical disciplines having an influence
  - Networking
  - Hypertext
    - Open hypermedia
  - Software architecture
  - Information retrieval
- The movement of people
- The social milieu Roy was in

# Themes



Technical disciplines having an influence

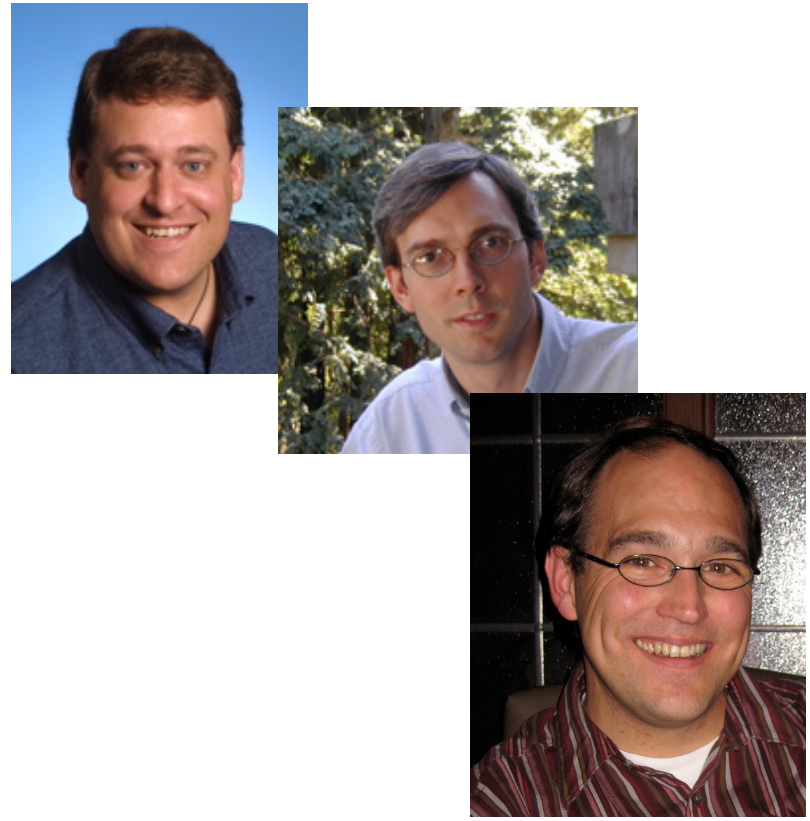
- Networking
- Hypertext
  - Open hypermedia
- Software architecture
- Information retrieval

The movement of people

The social milieu Roy was in

# Themes

- Technical disciplines having an influence
    - Networking
    - Hypertext
        - Open hypermedia
    - Software architecture
    - Information retrieval
- The movement of people
- The social milieu Roy was in



# Themes

- Technical disciplines having an influence
    - Networking
    - Hypertext
        - Open hypermedia
    - Software architecture
    - Information retrieval
- The movement of people
- The social milieu Roy was in

# The REST Principles

R1. Information is a resource, named by a URL

R2. Representation of a resource is a set of bytes, plus meta-data describing them

R3. All interactions are context-free

R4. Only a few operations available, and act in accordance with the other REST principles

R5. Idempotent operations and representation meta-data support caching

R6. Presence of intermediaries is promoted

# The REST Principles

R1. Information is a resou

R2. Representation of a
meta-data describing them

R3. All interactions are context-free

R4. Only a few operations available, and act in accordance with the other REST principles

R5. Idempotent operations and representation meta-data support caching

R6. Presence of intermediaries is promoted

These are the principal design decisions.
This* is the core architecture of the WWW.

*

*

The as-built architecture varies from the as-designed

*

The as-built architecture varies from the as-designed

The principles do not suffice to describe today’s Web applications

*

The as-built architecture varies from the as-designed

The principles do not suffice to describe today's Web applications

Crystallizing these principles (largely by Justin) came while writing the textbook

*

The as-built architecture varies from the as-designed

The principles do not suffice to describe today's Web applications

Crystallizing these principles (largely by Justin) came while writing the textbook

"Hypertext" doesn't appear in the REST Principles, per se

- If the *representation* is a hypertext then ...

# Research, Funding, Pubs

# Research, Funding, Pubs

Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others

# Research, Funding, Pubs

Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others

Should ease of publication direct what you do?

# Research, Funding, Pubs

Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others

Should ease of publication direct what you do?

The first paper on HTML and the Web was rejected by the 1991 ACM Hypertext conference

# Research, Funding, Pubs

Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others

Should ease of publication direct what you do?

The first paper on HTML and the Web was rejected by the 1991 ACM Hypertext conference

DARPA continually challenged my funding of Roy's and Jim's work (until it was a success!)

# Research, Funding, Pubs

Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others

Should ease of publication direct what you do?

The first paper on HTML and the Web was rejected by the 1991 ACM Hypertext conference

DARPA continually challenged my funding of Roy's and Jim's work (until it was a success!)

Roy took 9 years to get his Ph.D

# Research, Funding, Pubs

Some areas of software engineering are much easier to get pubs in (ICSE, FSE) than others

Should ease of publication direct what you do?

The first paper on HTML and the Web was rejected by the 1991 ACM Hypertext conference

DARPA continually challenged my funding of Roy’s and Jim’s work (until it was a success!)

Roy took 9 years to get his Ph.D

Exploratory development in SE research should be valued

# “Just One More Thing”

Isn’t all this stuff about the REST principles just trying to dress up development work that happened over a decade ago? Make “architecture” look good? Just trying to get more mileage out of old work?

# What Can You Do If...

You think through the crystallized REST principles,

See where they fail in modern applications, and then

Make an elegant generalization?

# Computational REST (CREST)

C1. A resource is a locus of computations, named by an URL

C2. The representation of a computation is an expression plus metadata to describe that expression.

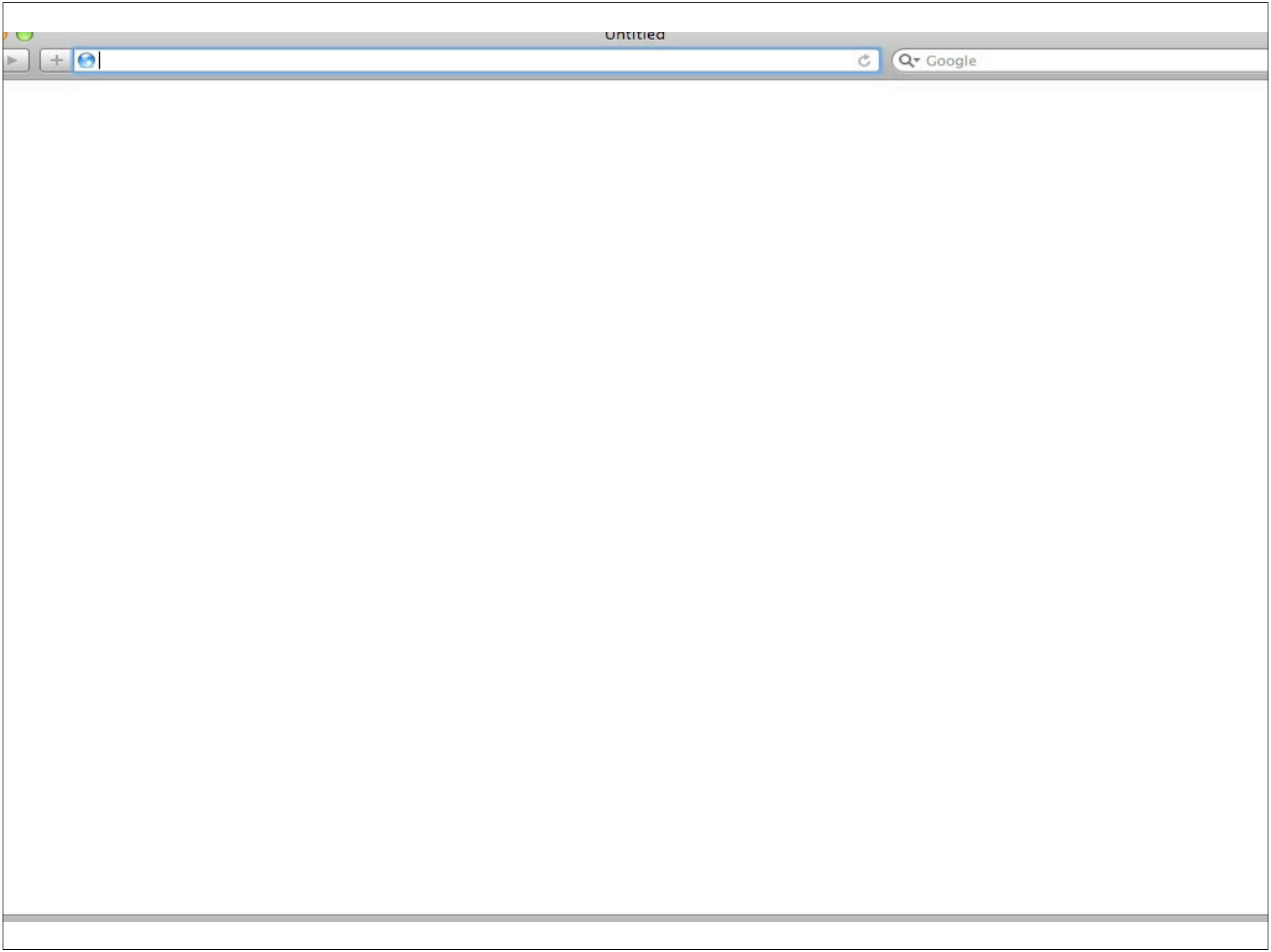
C3. All computations are context-free.

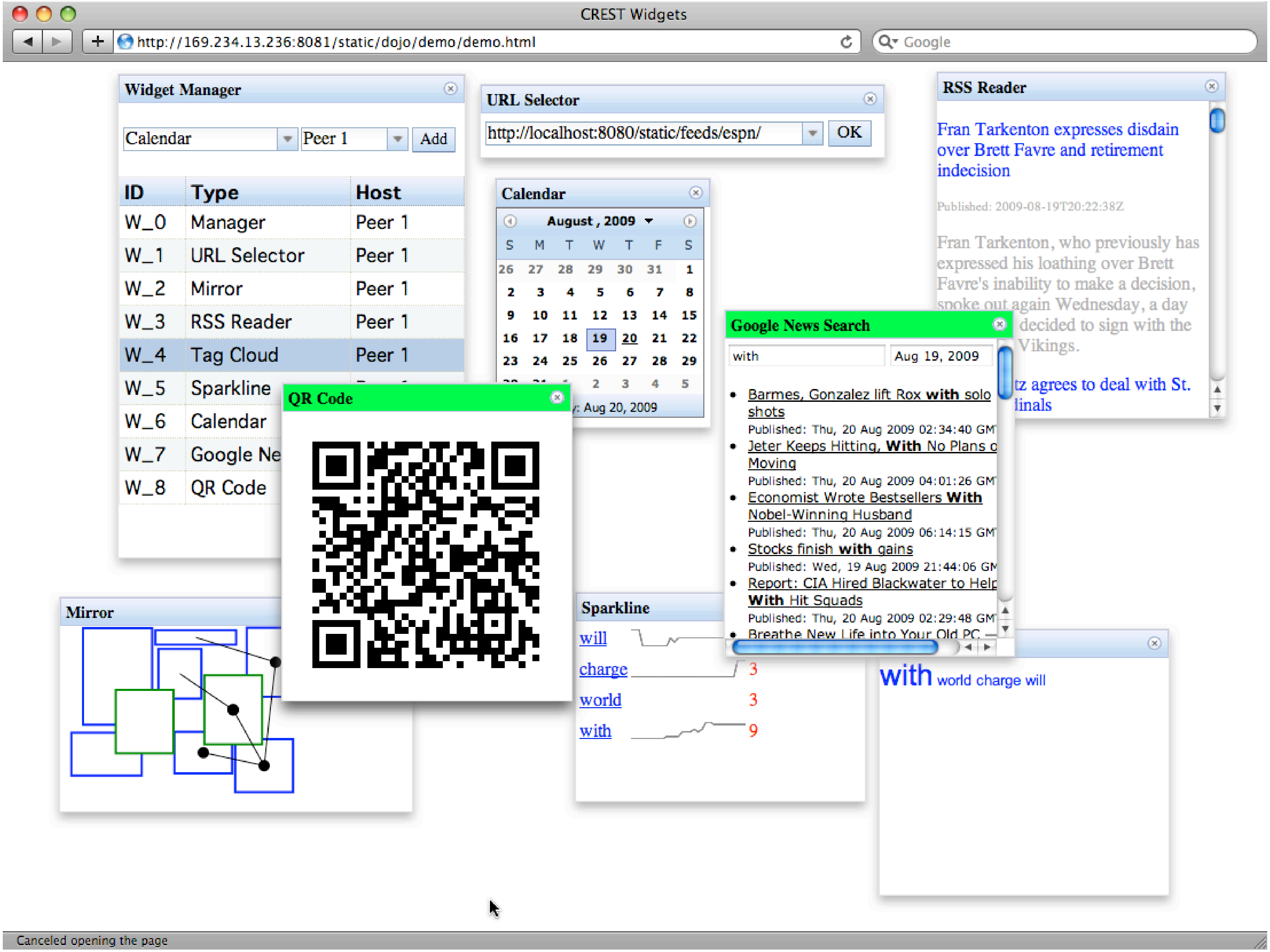
C4. Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged.

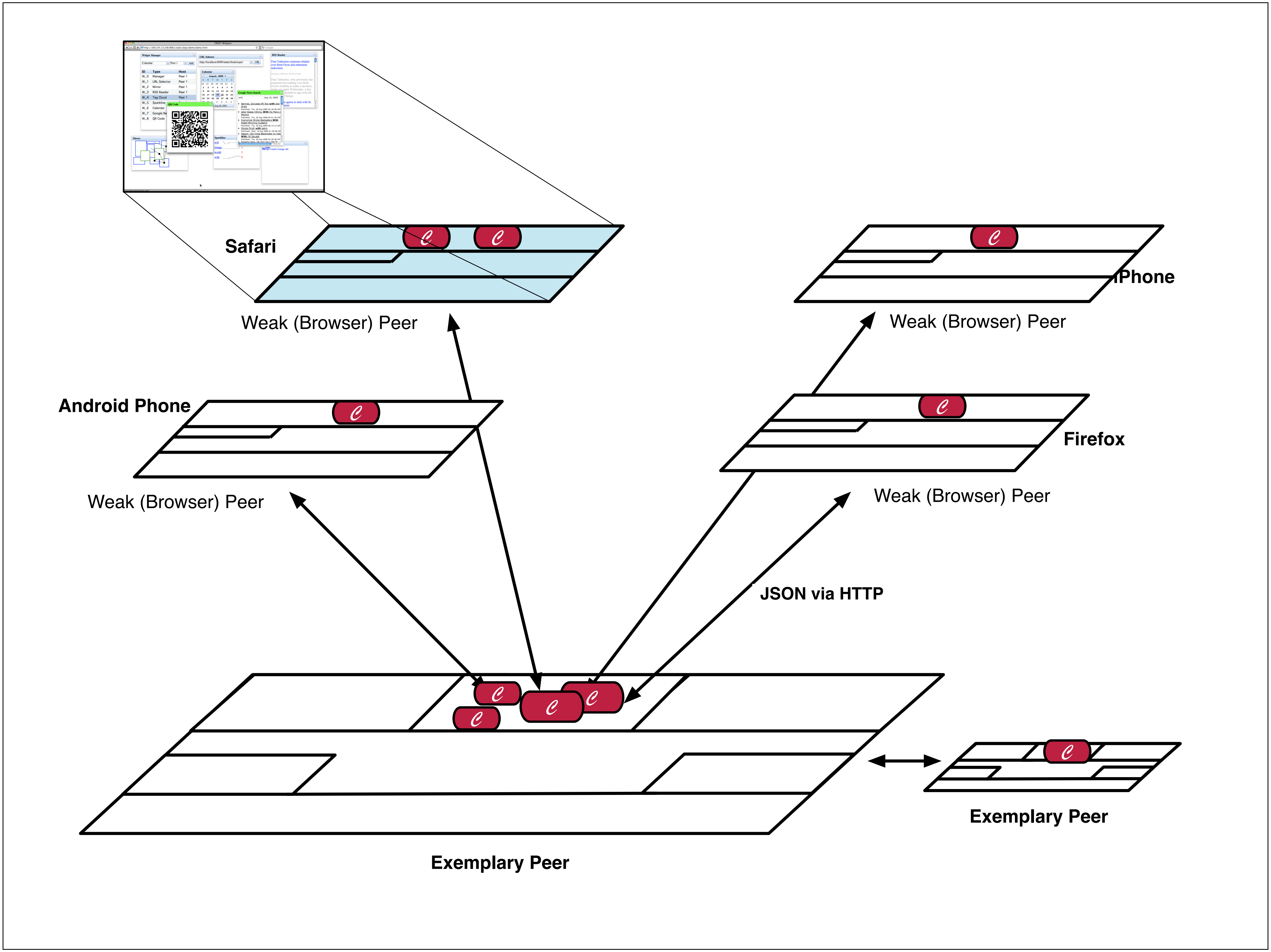
C5. The presence of intermediaries is promoted.

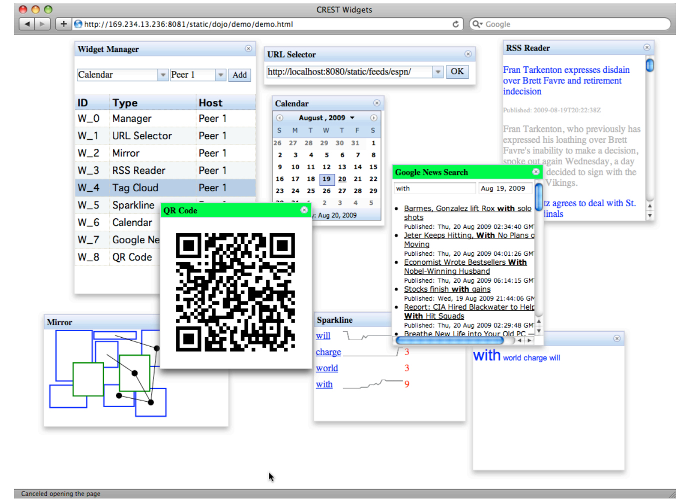
# Demo

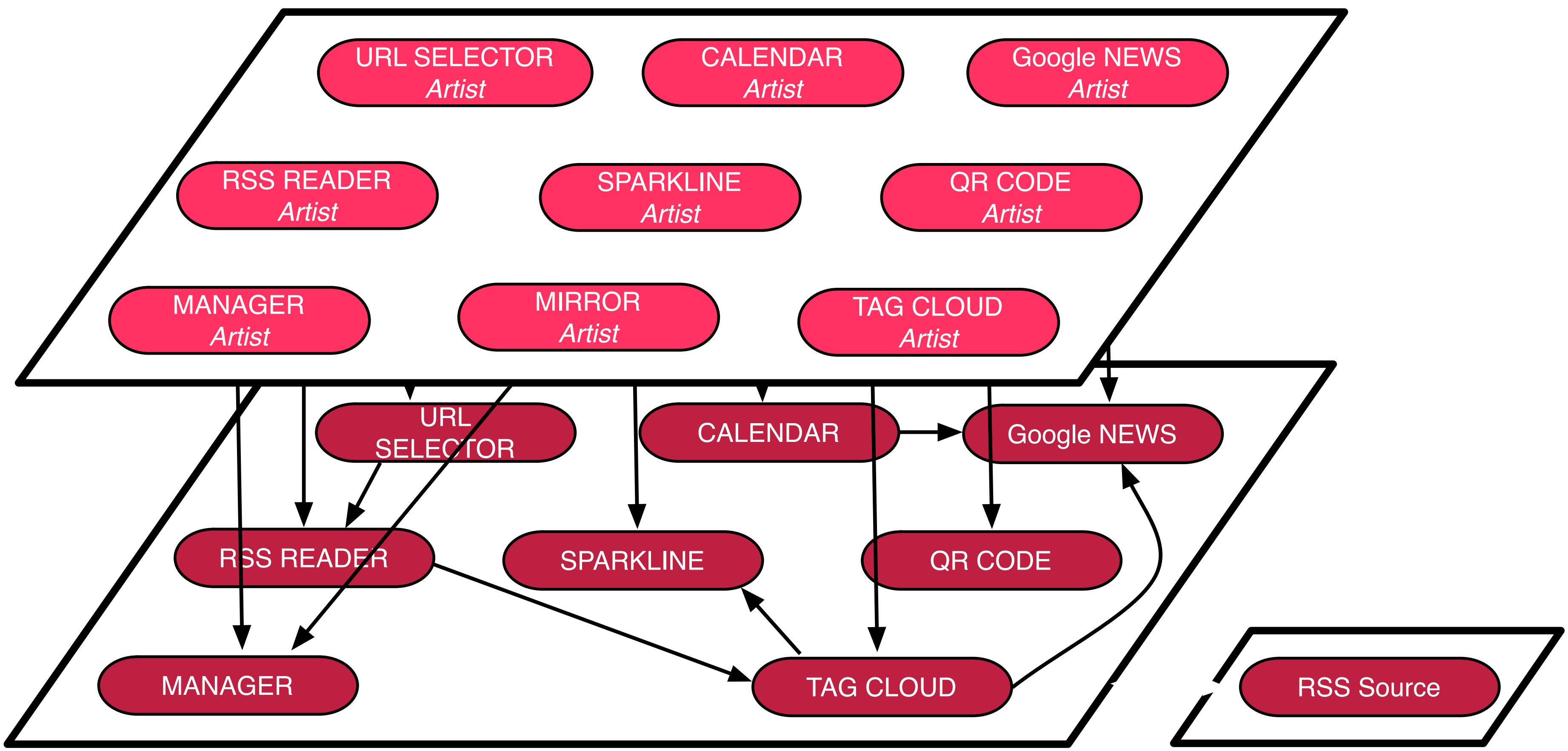
A dynamic, customizable news reader

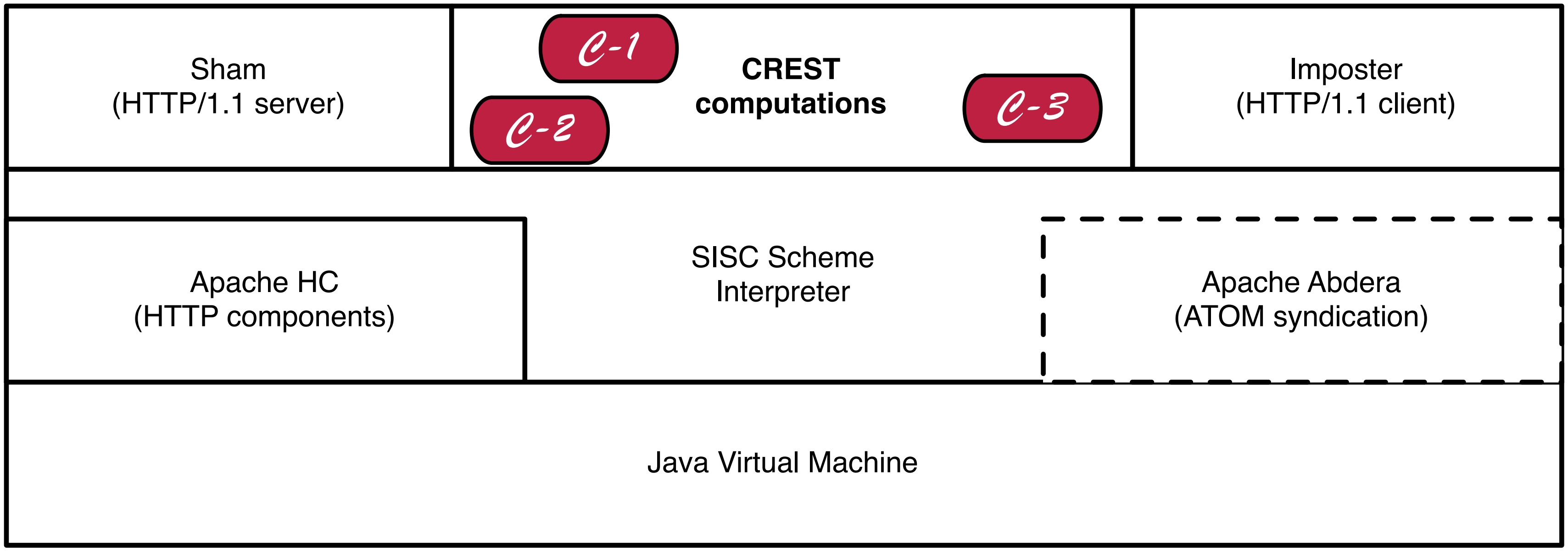

Untitled
Google


CREST Widgets
http://169.234.13.236:8081/static/dojo/demo/demo.html
Widget Manager
Calendar
Peer 1
Add
ID Type Host
W_0 Manager Peer 1
W_1 URL Selector Peer 1
W_2 Mirror Peer 1
W_3 RSS Reader Peer 1
W_4 Tag Cloud Peer 1
W_5 Sparkline
W_6 Calendar
W_7 Google Ne
W_8 QR Code
URL Selector
http://localhost:8080/static/feeds/espn/
OK
Calendar
August , 2009
RSS Reader
Fran Tarkenton expresses disdain over Brett Favre and retirement indecision
Published: 2009-08-19T20:22:38Z
Google News Search
with
Aug 19, 2009
Barmes, Gonzalez lift Rox with solo shots
Published: Thu, 20 Aug 2009 02:34:40 GMT
Jeter Keeps Hitting, With No Plans o
Economist Wrote Bestsellers With Nobel-Winning Husband
Published: Thu, 20 Aug 2009 06:14:15 GMT
Stocks finish with gains
Published: Wed, 19 Aug 2009 21:44:06 GM
Report: CIA Hired Blackwater to Help With Hit Squads
Published: Thu, 20 Aug 2009 02:29:48 GMT
QR Code
Mirror
Sparkline
will
charge 3
world 3
with 9
with world charge will
Canceled opening the page

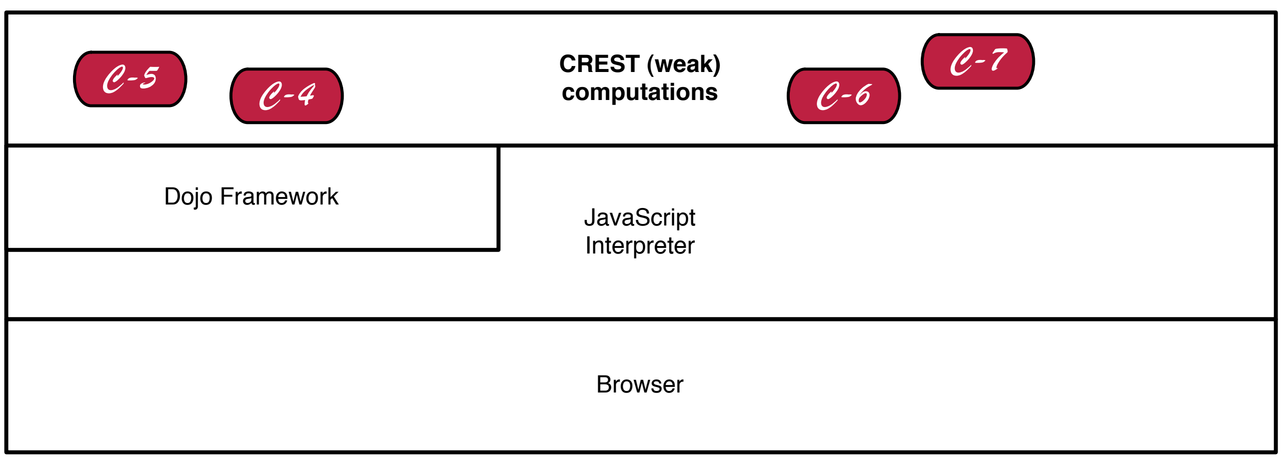

Safari
Weak (Browser) Peer
iPhone
Weak (Browser) Peer
Android Phone
Weak (Browser) Peer
Firefox
Weak (Browser) Peer
JSON via HTTP
Exemplary Peer
Exemplary Peer


Widget Manager
URL Selector
RSS Reader
Calendar
Google News Search
QR Code
Mirror
Sparkline


URL SELECTOR
Artist
CALENDAR
Artist
Google NEWS
Artist
RSS READER
Artist
SPARKLINE
Artist
QR CODE
Artist
MANAGER
Artist
MIRROR
Artist
TAG CLOUD
Artist
URL
SELECTOR
CALENDAR
Google NEWS
RSS READER
SPARKLINE
QR CODE
MANAGER
TAG CLOUD
RSS Source




Exemplary CREST peer




Weak CREST peer

# Demo FAQ's

What's going on in the browser?

"Just graphical rendering" Dojo and Javascript for drawing **only**

What are those visible widgets?

(1) Artists (2) Highly restricted CREST computations on restricted peers

What's being transported across the network?

HTML and JSON as weak CREST *expressions* describing state changes in the underlying computations

What's the transport protocol? HTTP/1.1

How many CREST computations were there? ≈12

How many nodes/servers were active? 2 laptops, 2 phones, Mac mini

How was time fudged? Mini had stored one month of RSS feed data

What kind of compatibility w/ existing infrastructure?

Completely backwards compatible with all WWW browsers & servers

# The Demo, redux

The point:  analysis of the **essential architectural decisions** of the WWW, followed by generalization, opens up an entirely new space of decentralized, Internet-based applications based on computations as the fundamental entity

# CREST Credits

Justin Erenkrantz

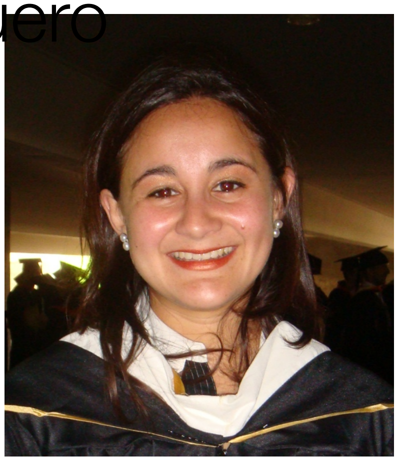
(Final defense: next Thursday)

Michael Golick

Demo support:

Yongjie Zheng

Alegria Baquero



Thank you!