Succeedings of the Second International Software Architecture Workshop (ISAW-2)

Alexander L. Wolf

Organizing Committee Chair

1 Introduction

Interest in software architecture as an area of research, education, and practice within software engineering has been growing steadily over the past decade. Software architecture is concerned with the principled study of large-grained software components, including their properties, relationships, and patterns of combination. It is becoming clear that one key to the effective development, operation, and evolution of software systems is the design and evaluation of appropriate architectures.

As a complement to the Fourth Symposium on the Foundations of Software Engineering (FSE-4), whose theme was software architecture, the Second International Software Architecture Workshop (ISAW-2) brought together practitioners and researchers for two intense days of discussion and work. The ISAW-2 proceedings [6] were published in the Joint Proceedings of the SIGSOFT '96 Workshops, ACM Press, ISBN 0-89791-867-3.

The participants were organized into three parallel working groups each focused on a different topic within software architecture.

- *Styles and Patterns*—Techniques and models for centering software architecture activities on generally useful design methods, components, and assemblages of components.
- *Architecture Description*—Languages and methods for the capture of architectural designs.
- *Tools and Methods*—Automated aids for designing, evaluating, validating, implementing, and evolving software architectures.

Tying these work groups together was the use of a common case study derived from a real-world architecture found in industry. Each working group studied this architecture from its particular perspective and, to a greater or lesser extent, used the case study to organize their discussions. It is interesting to note, for example, that the three groups created specialized depictions of the architecture for their discussion (see figures 3, 5, and 7).

The case study was the architecture of the Call Center Customer Care (C4) System, which was developed by Andersen Consulting. A description of the architecture and some of its more interesting challenges appears in Section 2. It is reproduced here in the hopes that it might be prove of continued use to the community.

Also appearing here are reports from each working group. The Styles and Patterns Working Group was led by Frances Paulisch, of Siemens, and Mary Shaw, of CMU. The group tried to uncover the styles and patterns that underlay various components of the C4 architecture. In the process, they defined a new kind of architectural style that they named the Data Ooze. The Architectural Description Working Group was led by Paul Clements, of the SEI, and Jeff Magee, of Imperial College. The group concentrated on identifying critical aspects of architectures that require description and on identifying important areas where further work in architectural description is needed. The Tools and Methods Working group was led by William Griswold, of UCSD, and Philippe Kruchten, of Rational. The group pretended to go through a development cycle for the C4 system in order to uncover various architectural tool and method needs.

At the conclusion of the workshop, the working group chairs held a panel session in which each group asked a "challenge" question of the other two groups. The questions and responses appear in Section 6.

We hope that these succeedings capture at least some essence of the very fruitful discussion that occurred at the workshop. Of course, this report cannot replace the benefits of actual attendance. The workshop will indeed continue, and we hope that the reader will be interested in attending a future ISAW.

Acknowledgements First, I thank the 40 participants for their stimulating position papers and conversation. I gratefully acknowledge the work of the Organizing Committee, who contributed a great deal of effort in selecting the participants and running the working groups. I thank Voytek Kozaczynski for suggesting the C4 case study. Voytek and Joel Heinke, architect of the C4 system, worked hard to distill and capture the C4 architecture in a form usable by the workshop participants. I thank the General Chair of SIGSOFT '96, Mark Moriconi, and the Local Arrangements Coordinator, Olga Karobkoff, for the helpful and efficient way in which they organized and ran the meeting. Finally, I thank Laura Vidal for her invaluable editorial and organizational assistance.

2 C4 Case Study

Joel Heinke (Architect) and Voytek Kozaczynski

The Call Center Customer Care (C4) System has been developed by Andersen Consulting for a large US telecommunication company. The primary function of the system is to support interactions with customers requesting new services (e.g., new phone lines), changes in the configuration of the existing services (e.g., phone number changes, long-distance company changes, or relocation), or reporting problems.

A.L. Wolf is with the Department of Computer Science, University of Colorado, Boulder, CO 80309 USA (e-mail: alw@cs.colorado.edu).

The phone company has over 19 million customers. Considering how often, on average, a customer changes their service configuration, the system has to support up to 400 service representatives simultaneously at a near 7X24 (seven days per week, 24 hours per day) availability level. However, these representatives are not the only means by which a customer can request a change or report problems. For example, there exists a phone service (Quick Service) by which customers can communicate with the system. This is discussed in more detail below.

2.1 C4 Functional Description

C4 is an OLTP (On-Line Transaction Processing) system that handles an interesting type of transaction. A transaction is initiated by a customer call, or so-called Business Event. There are three types of events:

1. service negotiations;
2. account management; and
3. trouble-call management.

Each event is then divided into *tasks* that are in turn broken into *activities*. Tasks are groups of related activities that the representative and the customer have to complete. For example, if the negotiation is about the customer moving from one address to another, there will be a set of activities related to terminating some of the existing services, a set of activities related to obtaining the new address, and a set of activities related to negotiating new services and activation dates.

C4 must provide:

- support for multiple concurrent tasks (e.g., the customer should be able to negotiate a number of different services during the same call);
- integrated support for completing activities (e.g., screen sequences, to-do lists, context-sensitive data fields, etc.);
- validation of availability of requested service;
- completion of activities and tasks;
- integrity of customer data;
- integrity of the final requested configuration;
- advice on available products and product "bundles";
- resolution of conflicting events; and
- support for interrupted and long-lasting conversations.

The last two points are particularly interesting. Conflicting events come from multiple so-called *authors* of events. For example, while someone is negotiating a new phone line, the spouse is using the phone company kiosk at a bank to request an ISDN line that will have an extra two phone lines. In general, events can come from:

- direct conversations with service representatives (the case described here);
- automated call center (the phone menu-type system);
- kiosks (in the future); and
- Direct customer connections, such as Internet (in the future).

Before C4 sends a service request to NOSS (see below), it has to make sure that all related events have been combined or conflicts resolved.

The other requirement comes from the fact that a conversation with a customer can be interrupted (for technical reasons, for example) or suspended by the customer or the representative. The first case is rather obvious. An example of the second case is when a customer says something like "let me talk to my spouse and I will call you back". In any case, C4 has to manage the context that persists and can be recalled.

2.2 System Interactions

The C4 system interacts with a number of other systems:

- a network provisioning system that makes physical changes to the network configurations and supports network management;
- a billing system;
- a host of corporate databases; and
- a number of downstream systems.

A high-level structure of the system interactions is show in Figure 1.

2.2.1 Interactions with NOSS

NOSS (Network Operations Support System) is the network management and provisioning system. It is being developed concurrently by a large third-party hardware/software company specializing in communication networks. Its functionality includes:

- *workforce management*—management of maintenance crews;
- *provisioning*—putting physical network components in place such as connections from the curb to the house;
- *network creation*—a peculiar name for maintaining information about new and existing physical networks;
- *activation*—automatic turning on and off of services;
- *network management* that provides
  - status monitoring and measurements,
  - proactive maintenance,
  - diagnostics, and
  - problem reporting;

  and
- *field access*—a subsystem providing up-to-the-minute information for field technicians.

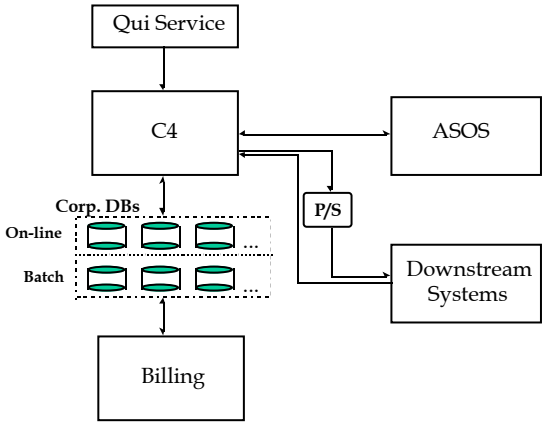

Figure 1: Components and Interactions in the C4 Architecture.

The NOSS architecture is based on the OSI CMIS (Common Management Information Service). The interface between C4 and NOSS is a large set of messages in the named-tag/value format. Each message interchange is technically a synchronous pair of messages composed of a request and a reply. At the application level there are two types of interactions:

1. a request followed by a response that contains the requested information and/or confirmation of NOSS action; and
2. an overall interaction consisting of two message pairs:

   (a) initial C4 request followed by a NOSS reply containing a request identifier, and
   (b) an unsolicited message from NOSS containing the previously issued request identifier along with related data followed by a C4 reply acknowledging receipt of the message.

The basic interactions between C4 and NOSS are as follows.

- C4 sends a Service Request (service order message) to NOSS to perform service reconfigurations.
- C4 sends queries to NOSS about existing network status or capabilities. This is usually done while a service representative is talking to a customer. For example, a request may be sent to NOSS about availability of service at a particular geographic address/service location or possible service activation dates.
- C4 may request NOSS to "lock" resources such as phone numbers for a service request.

2.2.2 Interactions with Downstream Systems

Downstream systems are systems such as Long-distance Carrier Services, 911 Service, Voice Mail, Phone Directory, and Revenue Collections (credit scoring and checking). C4 interacts with these systems in two ways.

1. It publishes requests, via the Publish/Subscribe (P/S) component, to which the downstream systems should react. For example, all new phone connections are published so that the 911 service is connected to them in the required number of hours.
2. The downstream systems notify C4, via a direct asynchronous message, about significant business events that affect customer configuration. For example, the Long-distance Carrier system may notify C4 that a client should be connected to a new long-distance company.

One of the design challenges that we discuss below is that direct requests from a customer may conflict with a similar request coming from an external system. For example, a customer may call to request a change to their long-distance carrier at the same time that a third carrier sends notification that the same customer has now selected them as their long-distance carrier.

2.2.3 Interactions with Corporate DBs and Billing

Some of the major functions of the Billing System include:

- invoice calculation for local services;
- invoice printing for both local and long-distance services;
- revenue reporting; and
- bill inquiry and adjustment.

C4 does not interact with the Billing System, but it works against the same set of databases. More specifically, C4 interacts with the on-line part of the corporate databases, while the Billing System works with a batch copy of the on-line databases (this is shown in Figure 1). Monthly billing is done in a number of cycles. Each cycle processes the billing information of a set of customers. All updates to the data of the current cycle are halted (applied only to the on-line data) until the end of the cycle. The updates at the end of the cycle are done in batch and are transparent to C4.

All customer data are stored in over 100 tables. C4 does not use all of them. During any customer conversation, C4 obtains a basic customer profile from about a dozen tables. Other tables are read and/or updated on demand.

2.3 Execution Architecture

The system execution architecture is a standard three-tier client/server configuration, as shown in Figure 2. All service

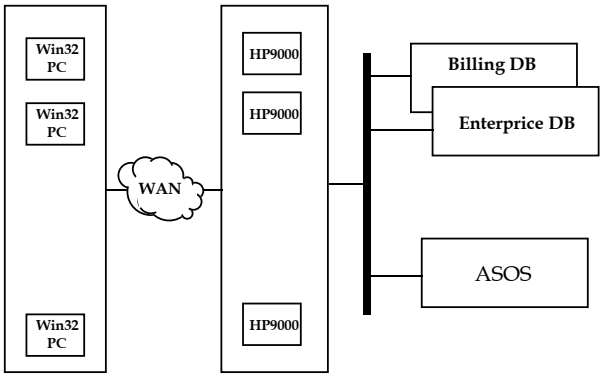

Figure 2: Execution Architecture of the C4 System.

representatives work on PCs running the WinNT OS connected to a LAN that is interconnected to a WAN. The middle layer is a cluster of HP 9000 servers running UNIX and a TP Monitor that can load-balance among them. The TCP/IP communications protocol is used throughout the network. The back end runs on a dedicated high throughput LAN. This LAN connects the Enterprise and Billing databases to the servers as well as a TCP/IP connection to NOSS via a module that marshals messages between C4 and NOSS. C4 runs on the cluster of PCs and HP 9000s.

In light of this configuration, the following are additional architectural constraints.

- No persistent data caching on the service representative workstations to limit the implications of local failures.
- No databases at office locations.
- No administrators at local offices.
- No maintenance down time.
- The middle-layer server cluster must be tuned for performance.
- It must be possible to add servers to increase throughput.
- The back end must be tuned for database performance.
- There must be a preferred place to maintain persistence.
- The operations architecture must be well engineered.
- High availability cannot be achieved by utilizing fault-tolerant hardware (this option is not economically viable).

2.4 Key Architectural Challenges

Included here is a partial list of architectural challenges for C4. The challenges are not completely independent, so there is some repetition in the list. A number of challenges are implied by the execution architecture selected for the system.

- Managing time and date effectively. Customers may want service changes in the future and this implies:
  - some form of a "tickler" system; and
  - ability to inform the customer about future changes of rates and/or services.
- Interfacing with multiple authors of business events. As explained above, the main source of business events is direct conversation by a service representative with a customer. However, other sources, such as downstream systems, kiosks, etc., have to be accommodated. Some important points about this include:
  - business events from different authors may conflict with regard to the requested configuration at a service location; and
  - business events from different authors may be received and processed at the same point in time—business events from different authors become different service requests that must be cross-validated to ensure a valid resulting configuration.
- Architect something that is economically viable with a small set of customers and yet can grow to a very large network (i.e., 15 million customers).
  - it should not require high initial equipment investment;
  - it should allow for "leaner" growth (with respect to a cost/capacity function);
  - it should be able to grow at a rapid rate (e.g., 1000+ new customers per day); and
  - it should allow the identification, monitoring, and elimination of processing bottlenecks.
- Validation of a requested service configuration should be done at near real-time. This implies that:
  - C4 has to communicate with NOSS and other systems, while guiding service representatives through tasks and activities; and
  - C4 may request a "lock" on some of the network resources (such as a few phone numbers) for a fixed amount of time.
- Support for long-lasting, interrupted sessions. This issues has been described above.
- Integrated (smart) performance support for service representatives.
- Support a large number (e.g., 400+) service representatives concurrently, with a minimum of 100 service representatives to start.
- Near 7X24 application availability.

3 Styles and Patterns Group

Group Chairs: Frances Paulisch and Mary Shaw

When architecting a system, a software architect will typically rely on knowledge about how similar systems have been designed in the past. Recurring design problems and their solutions can be viewed as architectural styles. A taxonomy of such styles is beginning to emerge e.g., dataflow systems such as pipes and filters and data-centered systems such as blackboards [5]. An architectural style defines a family of systems by describing their component and connector types together with a set of constraints on how they can be combined. Ideally, certain extra-functional properties (e.g., robustness, maintainability, scalability, etc.) are associated with the style. A pattern is a prose form for recording architectural styles as problem/solution-pairs and is particularly valuable as a communication mechanism for the software architect and other persons involved in developing the system (designers, users, etc.). In addition to the seminal text on design patterns that describe micro-architectures for software [3], there is a text describing patterns at various levels of scale, including the architectural level [2].

The topic our working group addressed was "styles and patterns" for software architecture. We explicitly chose to focus our work on the case study at hand (the C4 system described above) and found this practical approach valuable. We started our working group with a short round of introductions and looked at the C4 problem from our various perspectives using the case study as a target. This allowed us to reach both a better understanding of each others viewpoints as well as of the case study. The industrial software architects among us confirmed that meetings and discussions of this kind on software architecture are typical for them.

3.1 Working Group Participants

The following gives a very short summary of each participant's position; for more detailed information, refer to the position papers in the workshop proceedings [6].

- Jean-Marc DeBaud points out that most work in software architecture focuses on solutions rather than on problems. He outlines a domain-oriented method for the definition of problems and their mapping to architectures.
- Robert DeLine describes an architecture description language that supports user-defined element types and has an extensible compiler architecture based on Microsoft's OLE.
- Voytek Kozaczynski proposes a categorization of components common to business systems, which is based on their extra-functional characteristics. In his position paper he argues that this categorization can help architects think about how to decompose and structure complex systems.
- Spiros Mancoridis describes a technique for computing the interfaces of subsystems that were created during a reverse engineering process, and a visual formalism, called ASF, that enables designers to define styles of subsystem interfaces with precision.
- Jim Ning describes how the architecture specification language (ASL) would be used to describe the interfaces of the C4 components. Support for evolution is given through the automatic generation of "glue" code.
- Carlo Montangero introduces some refinement patterns that correspond to well-known architectural styles. The use of architectural refinement is valuable because it ensures that only instances of correct steps are used and it makes the architecture explicit. The motivation for this work is software process modeling.
- Peyman Oreizy shows how the C2 style of architecture can be used to describe the architecture of the case study.
- Frances Paulisch describes various aspects of tool support for software architecture, particularly for design patterns. A major challenge is to promote reuse, yet support the variability inherent in designing with patterns.
- Jeffrey Poulin describes the evolution of a software architecture for Management Information Systems. In particular, he focuses on a "dual baseline" architectural description: the objective architecture we seek to achieve, and the current architecture we actually can implement based on the realities.
- Will Tracz, although not directly in our group, was "virtually" present through his position paper focusing on the relationship between properties of software architectures and architectural styles.
- Alexander Ran makes the case that just as components are structured as compositions of lower-level components, interfaces must be structured as compositions of lower-level interfaces. He thus aims to avoid a granularity mismatch between the level of abstraction of components and their interfaces.
- Jason Robbins presents an approach to architectural analysis that supports evolution by providing feedback as design decisions are being made by the software architect.
- Mary Shaw and Paul Clements' joint position paper gives a two-dimensional classification strategy for architectural styles focusing on control issues, data issues, as well as control/data interaction issues. The goal is to establish a uniform descriptive standard for architectural styles, provide a systematic organization to support retrieval of information about styles, discriminate among different styles, and, eventually, organize advice on selecting a style for a given problem.
- Kevin Sullivan proposes that we should view software design as a process of deciding how to make irreversible capital investment in software assets of uncertain value, and that financial options theory provides a firm, unifying, simplifying, and well-developed basis for such decision-making.

To document the better understanding of the case study, we jointly derived a high-level diagram of the system that served as a basis for our discussions (see Figure 3). We focused

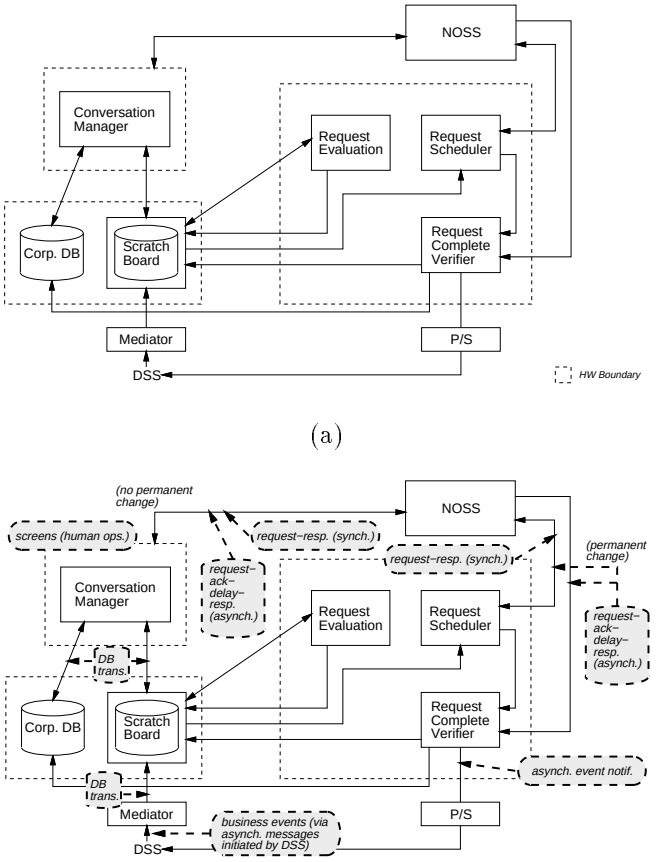

Figure 3: View of the C4 Architecture Before (a) and After (b) Examining the Different Kinds of Interactions.

our discussions on the topics of "interfaces and interactions", "scalability", and "styles and patterns", each of which is addressed individually below.

3.2 Interfaces and Interactions

We took a look at the nature of the lines in the informal "box-and-arrow" diagram of the architecture as shown in Figure 3a. We found that by focusing not only on what the components do but also on what the connectors do we get valuable insights into the problem. As shown in Figure 3b, there are a number of different kinds of interactions involved in this system, including asynchronous and synchronous messages from C4 to NOSS, asynchronous event notification, asynchronous messages ("Business Events") initiated by the downstream systems, and standard database transactions between C4, the database, and the billing component.

We reached agreement that the different types of communication imply design constraints for C4 and that at the architectural design level it is beneficial to distinguish between these various types of communication.

3.3 Scalability

One of the most important architectural challenges in the C4 system was its scalability in the number of customers the system is able to support. Ideally, one would want the "vision" (as Poulin said) of the architecture to remain stable despite a significant increase in scale. In general, we recognized that scalability is such that the costs of increasing the scale are initially "linear" but they may reach a point eventually in which a major change in the architecture is needed to continue to scale up. When architecting a system, one should try to identify early, for each of the critical extra-functional properties, where these discontinuities in the scale-up are likely to be and to plan accordingly. For example, consider at what point will such a system have to scale from a single database to a cluster of replicated databases to a set of geographically distributed databases and what effect will that have on the software architecture? In the terms of the options theory promoted by Sullivan, it is important to examine early how likely it is that a change will be necessary, what the "up front" cost to build in that flexibility is, and what the long-term benefits are.

3.4 Styles and Patterns

As is typical for large systems, numerous styles can be identified in the C4 system. We chose to focus on one of them—one that we view as important in that it helps us get a better understanding of how an essential part of the system works.

During early discussion of C4 we "tried out" several architectural styles, including data flow, communication systems, events, and blackboards, to see how well they helped to explain the system and how well they might guide subsequent development. None fit well, so we asked what other systems we knew that resembled C4. We recognized a variety of other applications that share the characteristic that they process requests, but they must gather information incrementally before finally satisfying the requests. We recognized a new underlying style that describes these systems, which we named "Data Ooze" (see Figure 4).

To make this new style concrete, we describe the Data Ooze style here in pattern form.

- *Name:* The "Data Ooze" architecture; thus named because data does not flow, but oozes gradually through the system. Also known as the "Smart Filter".

  The "Data Ooze" architectural pattern is useful in transaction-processing applications in which a set of possibly incomplete and possibly inconsistent transactions are gradually and in a non-deterministic order built up over time into a set of valid transactions that are then processed.
- *Example:* Consider the Call Center Customer Care System (C4) described above. In this system, a set of possibly incomplete and possibly inconsistent transactions enter the C4 system either through telephone conversations with service representatives, through the automated call center,

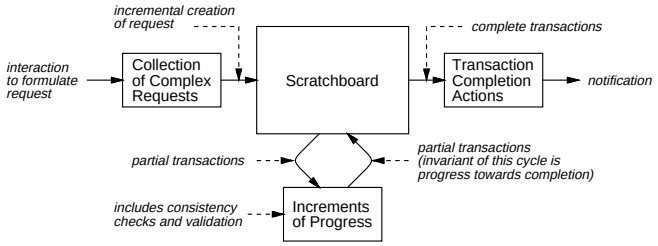

Figure 4: The Data Ooze Style.

through phone company kiosks, or through direct customer connections (e.g., through the internet). An example of an incomplete transaction is when, during a lengthy conversation between the customer and the service representative in which a significant amount of information about the requested service is gathered, the customer suddenly breaks off the conversation (humorously referred to during the workshop as the "my cat is on fire" scenario). The incomplete transaction may reserve some system resource (e.g., temporarily assign a particular phone number). At some future point in time, the information recorded in this conversation (the "context") should be recalled from the system and completed (or alternatively the information would be removed from the system after reaching a "timeout" threshold). An example of an inconsistent transaction is when a husband and wife request different and conflicting services at approximately the same time (e.g., one via a telephone conversation with a service representative and the other via a phone company kiosk).

Through various actions, the transactions make gradual progress towards completion. This can be through the arrival of additional information completing the transaction, through timeout, resolution of conflicting events, or information from the downstream systems described in the C4 case study (i.e., notification that a resource is now available).

Once complete, the transaction is handed off to the NOSS-System to be carried out and the billing process is initiated.

- *Context:* Processing data streams (transaction processing).
- *Problem:* The following lists essential distinguishing characteristics that are common to transaction-processing problems of this kind:
  - the transactions may be incomplete and/or inconsistent;
  - the transactions make gradual progress towards completion; and
  - the ordering of the transactions is not relevant.
- *Solution:* The idea behind the Data Ooze architecture is to store the incomplete and possibly inconsistent transactions in a buffer known as the "scratchboard" (see Figure 4). As the various parts of the system make incremental improvements on the set of transactions (e.g., through consistency checks and validation), their status is updated in the scratchboard. The incoming transactions flow (although indefinitely and non-deterministically delayed) gradually (or "ooze") through the system until the transaction is complete.
- *Related to:* It is somewhat like a dataflow system in that data flows through the system, but the transactions are not ordered or complete. It is somewhat like a blackboard system in that gradual progress to completion is made through changes in the shared data and these changes come in no predetermined order.
- *Second example:* An additional example of where such a style would apply is a travel reservation system in which (possibly multiple) customers tentatively book (possibly multiple) flights, specifying various complex constraints (a particular airline, best price, frequent flyer bonus, over weekend, etc.) and only once the transaction is completed and confirmed with the customer does the transaction to book the trip get processed (followed by issuing tickets and billing).

3.5 Conclusions

We are grateful to Andersen Consulting for having provided us with a case study to focus our work on. We thank also all of the participants for their active participation in the working group.

Together, we found that looking at the problem from the perspective of different styles was an effective way of getting a better understanding of the problem. We identified a particular style, the "Data Ooze" style, which allows us to communicate the essence of an important part of the system. Hopefully having this style in the repertoire will make it easier for software architects designing similar systems to focus their attention on the particularly interesting aspects of the application. We believe that it is worthwhile to continue collecting and classifying styles, possibly in pattern-form, so that software architects can build on their own experience as well as on that of experienced architects.

4 Architecture Description Group

Group Chairs: Paul Clements and Jeff Magee

The group used its time by identifying the architectural description goals and issues of interest to the group, and by discussing these goals and issues where possible using the C4 case study as a source of problems and examples. However, the group decided that it would not limit the scope of the discussion to those problems raised directly by the case study.

The group tried to avoid discussion of the detailed differences of specific architecture description languages and to concentrate on the general requirements for architecture description. Following brief position presentations, discussion centered on architectural views, styles, constraints, and variability.

4.1 Position Presentations

Each participant gave a brief talk summarising their position and raising issues that they wanted to table for a more general discussion. The talks (in order of presentation) were as follows.

- Robert Allen reported on his experience with applying the Wright ADL to the DoD "High Level Architecture for Simulations (HLA)". The talk raised issues of interface definition concerned with consistency and precision, and how compliance with a standard can be ensured while permitting the variation required by different simulation models.
- David Garlan addressed the question of when is it possible to implement a design in one style using a design in a different style. He suggested an approach to style-based architectural refinement that identifies subsets of systems in a particular style amenable to specialised treatment. The approach simplifies the problem of refinement by removing the requirement that a refinement technique be applicable to all instances of a particular style.
- Bob Balzer described the use of Relational Abstraction for specifying architectural constraints and outlined an approach to enforcing constraints at run time using instrumented connectors. The talk raised the general question of "When should we enforce architectural constraints".
- Ugo Montanari outlined new work on a "Tile Model" for concurrent systems that emphasises composability. The model shows promise in its ability to capture aspects of the dynamic structure. The talk raised the general issue of description of dynamic architectures.
- Richard Hilliard talked about the industrial requirements for architectural description and outlined an extensible OO framework for architectural description. The talk discussed the need for multiple view architectural descriptions.
- Larry Howard reported on his experience with developing structural models for air vehicle simulations. He identified the need for ADLs to express reusable structural abstractions.
- Jeff Kramer presented work on Self Organising architectures in which component interconnection is constrained by an architecture description but is not precisely prescribed by it. The description permits dynamic reorganisation consistent with the architecture. The talk raised the issue of enforcing global constraints in a distributed setting.
- Will Tracz raised the challenge of creating architecture descriptions that can be used to verify the extra-functional qualities of designs such as scalability, adaptability, extendibility, reliability, fault tolerance, safety, and implementability.
- Rick Mugridge reported on work on event-based software architectures. The talk raised the issues of tool integration, the impact of changes, and the traceability of architectural decisions.
- Nenad Mevidovic described an approach using the C2 ADL to performing dynamic architecture changes. The approach is to extend the ADL with an architecture construction notation (ACN) that can be used to describe changes.
- Dirk Riehle described an OO architectural framework for distributed business applications. The talk empasised the need for such an architecture to be reflective in all its key abstractions. He argued that this facilitated evolution.
- Richard Taylor argued that generalisation from domain experience was a superior paradigm for software architecture research. He supported this thesis by examples that demonstrate that restricting the application domain facilitates analysis that leads both to greater understanding and to more immediate results.
- David Wile reported on work on semantics for the architecture interchange language ACME. He argued that a structural or topological description is insufficient. Architectural description must also include elements of a behavioural specification to be useful.

4.2 Discussion

The presentations revealed that, collectively, we hold the following goals and uses for architecture representation:

- to ensure that behavioural requirements and global constraints are satisfied;
- to aid in the synthesis of systems; and
- to analyse for both functional and extra-functional properties.

The kinds of information that participants' technologies require and represent include dynamic (behavioural) models of systems and components, communication patterns of components, finite state machine models, component and system interfaces, constraints, structures, and styles. Subsequent discussion touched on many issues but centred around the following themes.

4.2.1 Architectural Views

A view of an architecture is a presentation that suppresses certain architectural information and exposes other information. All architectural representations are necessarily views, since none can adequately express all architectural information about a system (especially since there is no fixed line between architectural and non-architectural information). Some industrialists in attendance at the workshop (notably Philippe Kruchten and Robert Nord) have written about common architectural views that are useful in practice. A view may, for example, show information flow, process synchronisation, hierarchical decomposition of components, or other structure-based information.

Discussion centred around whether there was a central or core model of software architecture that could be used as a carrier for the additional information required by different views. It was agreed that for most current ADLs this core model

presented a structural view consisting of a configuration of components and connectors. Indeed the two descriptions of the C4 case study examined by the working group were both structural views—one in the C2 style and one in an ad hoc style. It was not agreed that this current state-of-the-art was necessarily the way forward and there were some strongly held opinions that the structural view is not the most important view in many application domains. This of course raised the issue of whether architectural description languages should only attempt to deal with a limited application domain (see below).

4.2.2 Architectural Styles

An Architectural Style is a set of constraints on the componentry and interconnections in an architecture; it defines a family of architectures, each of which is consistent with that style. Styles may be the architectural equivalent of design patterns, in that they provide a vocabulary for discussing common solutions to recurring design problems, and provide a basis for concentrated effort to achieve analytic results and supporting developmental infrastructure. Our group discussed the place of style in representation, and noted the need for representational techniques that acknowledge the existence of the role of style and accommodate changes in (evolution of) style over the life time of a system. We also need representation techniques that acknowledge that a single system can simultaneously exhibit multiple styles: either hierarchically (where the internal structure of a component in one style exhibits a second style), locationally (where one part of a system is in one style and a different part is in a different style), or simultaneously (where the entire system may be described equally well by different styles; for example, layered, object-oriented, and multi-processing).

A presentation by Richard Taylor of a description of the C4 case study architecture in the C2 style (see Figure 5) prompted the discussion of how to select an appropriate style for a particular problem. Many participants expressed the opinion that the C2 style was not natural for the case study problem and was an example of applying a notation successful in one domain to an inappropriate domain. Voytek Kozaczynski, as the case study expert, was encouraged that the C2 description captured many of the elements of the architecture actually used.

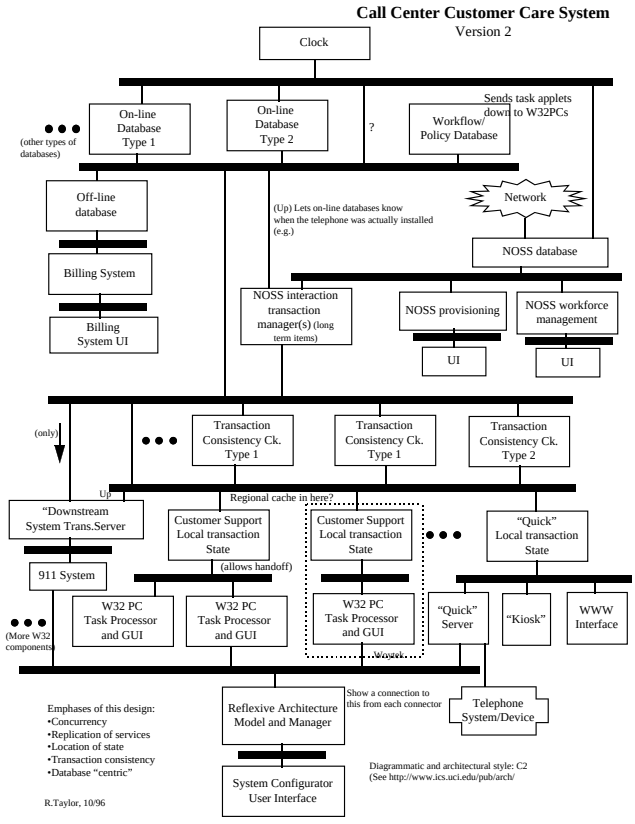

Figure 5: C4 Architecture Described Using the C2 Style.

4.2.3 Architectural Constraints

An architecture is an embodiment of a set of design decisions, each one of which limits the space of possible system solutions. Each design decision admits one set of subsequent possibilities and throws out all others, and is almost always in response to a constraint of some sort. Constraints may be levied on the behaviour of the system, its resource consumption, its interaction with itself and with other systems, or on developmental issues such as cost and schedule. The representation of an architecture can be said to be a representation

of the architect's response to all of the operative constraints, and those should not be lost in the process.

Under the heading of constraints, the group discussed the issue of traceability—what part of the architecture description reflects required constraints. The issue of when architecture constraints should be enforced was discussed. Type systems for components, connectors, and interfaces permit early enforcement of constraints. For dynamic architectures it was held that some constraints have to be enforced at run time. However, it was clear that where possible, constraints should be enforced statically, since run-time enforcement, particularly in distributed systems, can be expensive in communication and computation costs.

4.3 Conclusions

At the end of the session, there was a feeling that the following areas are ripe for more work.

- *Artifact-to-artifact traceability.* Formal representation can help with architectural refinement (including design to code). Issues of valid refinements, allowable variation from artifact to artifact, the importance of system-specific context for judging the validity of a refinement, and component (or artifact) interchangeability are all areas that would bear deeper investigation.
- *Reflective models.* Architectural models that embody knowledge about themselves provide new capabilities. Such models are useful in deriving measures of suitability about the architecture, and for managing architectural dynamism at system run time.
- *Variability.* Architectural representations must handle variability, for they by nature are abstractions of a system, and abstractions imply variability. In architecture, variability at run time implies dynamism. Variability at development time includes dealing with product variants and with different component implementations. Architectural description appears to be an appropriate level to capture and express system variation. To date, most ADLs to not have the capability of describing architecture variations.
- *Style.* As discussed previously, the role of style in architecture representation is currently unclear. Styles appear to be the key to capturing the experience of software architects and to providing the basis for selecting amenable analysis techniques.

5 Tools and Methods Group

Group Chairs: William Griswold and Philippe Kruchten

This working group consisted of researchers and practitioners from every "phase" (what will be called *activities* here to avoid any supposition of ordering) of software development: requirements acquisition (S. Bot, J. Kuusela), specification (P. Inverardi), design (R. Kazman, R. Monroe), testing and analysis (D. Richardson, N. Eikelmann), maintenance and evolution (W. Griswold, N. Mendonca, G. Murphy, W. Scherlis, G. Trausmuth), and process and planning (P. Kruchten, R. Nord). Most of the members had an interest in tools for one or more of these activities, and most were interested in the problems posed by evolving a system after its initial development.

Because the work of this group depended as much on *how* C4 was built as on its architecture, we spent an additional hour with Joel Heinke, the C4 architect. Joel explained how the architecture was conceived and how it related to the implementation of the C4 system. He also explained how tools were used in the process. He noted that the management of communication amongst team members was the single greatest challenge in getting C4 implemented. In particular, a lot of time was spent communicating in order to avoid misunderstandings or correct problems that arose because of misunderstandings. These problems occurred between the different activities of the project as well as amongst subteams within an activity (particularly implementation). Joel's belief was that these problems were due to ambiguities in the interfaces between subsystems. He acknowledged, in particular, that there was a tension between delaying design decisions to keep options open and creating unforeseen ambiguities. For instance, the frequency of interactions between components might be left undetermined until a late phase. Yet performance concerns might demand completely changing the interface between two components, resulting in changing component interfaces late in development.

5.1 Working Group Activities

Because of the spectrum of interests and skills in this group, it was decided that it would be best to explore the interactions and interdependencies of these activities. Such an approach would help working group members to learn about each other's work and how each activity relates to the others. False assumptions about how one activity is situated with the others would be exposed. New insights for the community or new research directions were secondary goals.

Initially we formed four activity groups: requirements, specification, design, and evaluation (e.g., testing), with the working group leaders acted as nominal managers (and hence in charge of process). The groups were to move through one "spiral" of architectural development, based on Boehm's *Lifecycle Architecture* concept [1]. However, the specification group quickly folded in with the testing group due to the time constraints and research interests. It was also decided to suppose that the initial C4 had already been built and deployed, so the current cycle was concerned with evolving the system to keep pace with new communication services.

Before beginning this spiral, each subgroup made a sales pitch

to the project leaders about why their approach to their activity was superior, and hence why they should be hired to carry out that activity. The purpose was to allow each subgroup to establish a strategy, help subgroup members to learn about each other's work, and to allow the working group participants to talk about their work in the context of the case study. This is what each group proposed.

- The requirements subgroup advocated an approach emphasizing stakeholder analysis and scenarios. Quality Function Deployment (QFD) was to be used to record each stakeholder's requirements, their relative importance, and the conflicts between requirements. High-level scenarios are then used to describe the requirements in a situated fashion. The advantages of this approach is that conflicting requirements are identified and resolved early and the resulting requirements are described in a fashion that exposes potential user and system dynamics.
- The design subgroup advocated a tools-based approach that records the system's design in a repository-based system with a graphical front-end. The unique characteristic of this system is that it is possible to record scenarios in this system so that design components can be linked to particular requirements in an intuitive fashion. Moreover, this system has the capability to visualize these scenarios dynamically, analyze the design for timing properties and the like, and also constrain the designer's modification activities with design rules.

  One problem faced by this subgroup was that the initial C4 was not developed within this system, so current design and scenario information had to be injected into the tool somehow. The evolution specialists in this subgroup proposed using their tools to translates the designers' informal design into a concrete design suitable for the repository. Software restructuring tools could also be used to better represent the design directly in the code. These techniques are unique in that they can use the informal information provided by designers to make an informed "recovery" of design information from the code.
- The evaluation subgroup focused primarily on testing and analysis. This subgroup proposed encoding the architecture in terms of partial operational specifications written in the Chemical Abstract Machine language (CHAM). "Test" cases and previous results of testing would be stored in a repository. Storage of previous results and connecting test cases to particular system requirements allows selective (and hence lower cost) testing of evolutionary increments of the system. Description of the system in a partial operational specification language allows early "testing" of the system (i.e., in absence of code) by executing the specification with test case inputs or proving properties of the specification with respect to the inputs. As a consequence, testing takes on a flavor of error prevention, rather than error detection. Because the specification language abstracts away non-architectural issues, state-space explosion can be avoided.

After the project managers hired these subgroups to carry out the evolution of C4, the actual architectural spiral was carried out (Figure 6). The subgroups proceeded in the order above. While one subgroup worked, the others watched or helped, as was appropriate. As stated earlier, the goal was to help the participants understand each other's work. Each group had limited time (about an hour), so focusing on just one new requirement was necessary.

The requirements subgroup held a stakeholder-based requirements session. The subgroup operated as facilitators, while the rest of the working group pretended to be stakeholders. The decision was to focus on two kinds of stakeholders, customers and software architects. First, the "customers" said what they wanted from future communication systems. About a dozen wishes were recorded, such as "I want access to the internet at any time, anywhere." Second, the "architects" stated their desires, including such statements as "I don't want the architecture of the system to change." The requirements were numbered and put in QFD matrices and prioritized.

It was agreed to focus on the mobility requirement, which requires C4 to register new cellular phones. Unlike regular phones, cellular phones are not associated with a phone company line that is "activated" to begin service. Instead, the phone itself must be registered with the phone company. Such a change could have a substantive impact on C4's architecture. Consequently, a scenario was developed for this requirement: The customer, using a web browser, navigates to the "add service" page, and selects "new cellular service". The system then prompts for the cellular phone's ID, which the customer types in.[1] The system then verifies that it can provide the service, informs the customer of the costs, etc. In the requirements phase, this scenario is recorded at the customer/C4 interface. It is refined in later phases where the architect's (potentially conflicting) requirements can be analyzed.

The design subgroup used this scenario to show how it would be connected to the design, and how the tool's analysis capabilities could be applied. This group also described in more detail how the informal design could be translated into a concrete form suitable for storage in the design tool's repository. For the sake of discussion, the tools group introduced an "execution architecture" based on an extended discussion with Joel Heinke, the C4 architect (Figure 7). Using this, they then elaborated the add-cellular-service scenario directly on the diagram (note "acs" markings in figure). Such an elaboration depends upon relating the actions that realized the scenario with system components. One way to achieve this is to relate the new scenario to an existing scenario. In this case, a supposed "add phone service" scenario was used to link in the new scenario. With this "linked" scenario, then, it is possible to identify the scope of change, likely performance characteristics, and the like. Moreover, the design rules of C4,

[1] No one in the group knew how a cellular phone's "address" gets communicated to a service provider. This interaction was invented to allow the requirements process to continue.

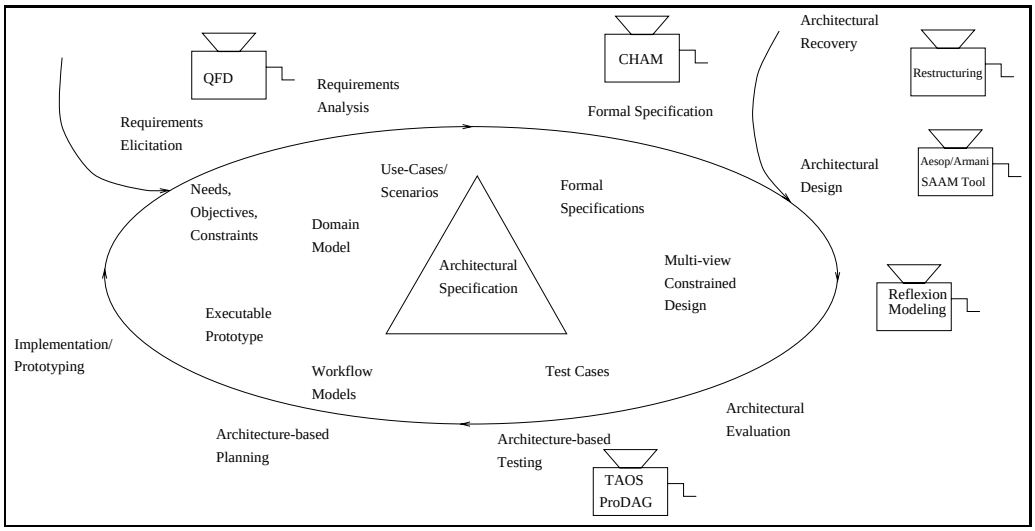

Figure 6: The Architecture Process. The oval depicts the flow of activities. Text inside represents the data created and modified. Text outside represents activities. "Grinders" correspond to the researchers' tools that can assist these activities.

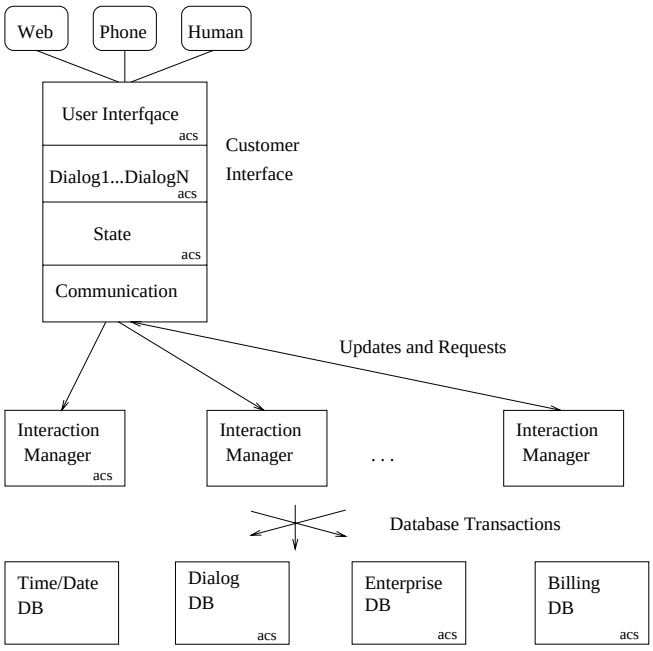

Figure 7: Architecture of C4 Used by Design Subgroup.

which are recorded in the tool as design expertise, specify that the service cannot be added by circumventing the layering of the customer interaction component.

The design subgroup then described how this architecture would have been "recovered" from the informal design in the first place. Of particular concern is relating the existing code to the components described in the architecture. They described how they would ask the system designers and implementors about how the system's architecture was manifested in the code. For instance, assignment of code and data to components is often manifested by naming conventions and placement in files. Connections between components may be manifested by procedure calls, data access, or other communication mechanisms. Given a specification of such properties for a given system, it is possible for tools to create the mappings between source code and architecture, and to report how well the postulated architecture represents the implemented system.

The evaluation group spoke of how scenarios could serve as a driver for testing and analysis. For instance test cases would be developed to exercise the scenarios. As scenarios are changed and added, new test cases are developed and obsolete ones removed. At the architecture stage, the test cases run on the operational specifications. As an example, the evaluation group drew up a short CHAM specification [4], which describes the interaction between the customer interaction component and the interaction manager component (Figure 8). This specification can be exercised by providing a specific start state, which corresponds to a scenario elicited in the requirements phase. By doing so, it is possible to prove or disprove properties of the architecture before coding has even begun.

C4 Tools	Tools Desired	Working Group Tools
Microsoft Office	"Traceability"	
E-mail	Workflow	Reflexion Models
Design1	Version & config. mgmt	SAAM tool
Flowcharter	Collaboration	Rational Rose
Queuing Simulator	Component & connector specification	Stakeholder-centric traceability
GUI & client code generator		CHAM
Programmer's workbench	Partial specifictions	Armani
Visual C++	Domain specific languages	Star Diagram Restructuring Tool
Testcase capture (no coverage)	Arch. recovery & restructuring	TAOS & ProDAG
code framework (library)	Testing	

Table 1: List of Tools Mentioned in Working Group. Tools are not intended to line up by kind or task.

5.2 Conclusions

Grammar:

P ::= IM | CI
D ::= add-service
C ::= output(D) | input(D)
M ::= M,M | P | D | C | M PAR M

Rewrite Rules:

output(D).M, input(D).M′ ⟶ M, M′
M PAR M′ ⟶ M, M′
CI ⟶ output(add-service).input(D′).CI

Initial State:

{output(add-service).CI PAR
input(add-service).IM PAR ...}

Figure 8: Simple CHAM Description for the Communication Between the Customer Interaction Component and the Interaction Manager Component.

One insight of this working group was the preoccupation on what many members referred to as "traceability". In particular, there was a preoccupation about how changes propagated from one activity to the next. Moreover, there was concern about how a single requirements change might propagate throughout the system implementation. These concerns were addressed by working through three methods: repositories, scenarios, and reverse engineering techniques. Repositories are used to establish links between different views of the system such as requirements, specification, design, etc. For our working group, scenarios served as a unifying concept for understanding component interactions and design dependencies. Lastly, reverse engineering tools were required to establish the initial connections between views, since C4 was initially developed using traditional tools. This is not a "traditional" reverse engineering task, in that this was not an attempt to recover a lost design, and could well have occurred even earlier in the development process. Also, it was observed that it is not possible to anticipate all the dependencies that might be of interest, since new dependencies can arise out of new system features or alternative architectures can be postulated. Consequently, reverse engineering tools play an important role in capturing and visualizing new dependencies when they become important, before or after deployment. One challenge faced by this work is capturing architectural information from a combination of source code, out of date documentation, and system developers' informal knowledge.

Lastly, we recorded all the tools that were mentioned during the workshop, and classified them into three categories: tools used to build C4, tools that would be beneficial to architectural development, and tools that our working group were developing (Table 1). We offer no comment on the disparities, except to note that the transition from research to practice often takes 15 years.

6 Intergroup Debate

In this section we reproduce the questions and answers from the integroup debate that occurred at the closing plenary session.

6.1 Styles and Patterns

Question from Tools and Methods Group

"Conventions" are a fundamental aspect of engineering practice. In software architecture, these conventions transcend code, including requirements, specification, process, etc. To what extent can styles and patterns incorporate these elements?

Styles are not intended to capture all information about a system—they are intended to capture structural information such as the parts from which a system is composed, the interaction among these parts, and constraints that shape the character of the system.

Patterns, on the other hand, are intended to carry contextual information as well as the definition of the core idea of the pattern. So they can capture these external conventions.

Question from Architecture Description Group

Complex systems incorporate multiple styles. How do we mix/compose architectural styles?

Most real systems rely on multiple styles. They are combined in a variety of ways. A couple of these are easy to handle.

- One style describes the system at some level; different styles are used to implement the components. Consider, for example, a UNIX filter system, represented as a shell script. Nothing constrains the implementations of the individual filters. When you look inside them you may find more pipes, objects, or perhaps spaghetti.
- A system partitions cleanly into two or more segments, with each segment in some style and clearly delineated boundaries between segments. In this case it is possible to write "glue" components that take care of the style crossings at the segment boundaries.

These are not the really interesting cases, though. The interesting (and hard!) case arises when multiple styles are used concurrently to describe different aspects of a system. For example, a study of 11 cruise control systems found 8 different styles used to explain the systems; in most cases two or three styles were used concurrently. The Viewpoints Workshop, meeting concurrently with this one, may be addressing this problem.

6.2 Architecture Description

Question From Tools and Methods Group

Our guest architect identified the management of communication amongst teams to be the major problem in the development of their system. This issue operates throughout the lifecycle. The costly communication occurs between phases (e.g., requirements to specification), crosses features and modules, etc. To what extent can ADLs manage and describe these wide-spectrum dependencies?

The communication problem is essentially an interface problem among teams. Communication flows from team to team, teams must be coordinated and synchronised, the performance of teams must be predicted and resources provided. ADLs can manage and describe these dependencies to the extent that the "team interface" issues reflect interface issues among the software components that the teams are building. For example, inter-team communication flow often involves negotiation about the behaviour of each other's components, or ways in which the components will work together (integrate) in the context of broader system context such as performance requirements or adherence to a style. ADLs serve precisely the purpose of formalising these decisions and capturing them for future reference. To the extent that inter-team communication does not involve such decisions, then ADLs are not appropriate and should not be expected to help. For example, helping to decide when to baseline a document or which team next gets use of the biggest conference room is better handled by a workflow manager or process enactment tool, of which there is no shortage.

Question from Styles and Patterns Group

When, if ever, will there be a concrete, complete, usable description of a real, large-scale system in an ADL? And when will it be more cost-effective to use an ADL than not to?

Some purveyors of ADLs would argue that such descriptions in fact exist today. Aircraft simulators, embedded missile software, and computer chips have all been modeled to a high degree of industrial fidelity with ADLs in environments that are at least not completely academic. The question of cost-effectiveness is more difficult. We posit that the greatest benefit of formal architectural representation is the early prediction and analysis that it enables. Analysing for schedulability, predicting performance, generating simulations to check for desired behaviour, help in building prototypes, and automatic production of components such as process schedulers are all compelling reasons to use an ADL, arguably much more so than their other main use as an unambiguous vehicle for communicating the architecture to stakeholders. As always, measuring cost-effectiveness first requires measuring cost, something at which our field continues to be poor. When we can quantify the benefits of early simulation, code generation, and the like, we will be able to quantify the benefits of ADL usage.

6.3 Tools and Methods

Question from Styles and Patterns Group

Given that the tool of choice is Microsoft Office and that the adoption of a tool should be easy and incremental, how can we provide software architects with effective tools?

Microsoft Office is successful in part because of its flexibility and performance. It can be learned quickly, used for a wide variety of tasks, and adapted to new tasks with virtually no effort. Moreover, these tools can be integrated with other tools with lightweight integration technologies like OLE to preserve openness.

For new tools to be successful, they must be readily adaptable to the tasks at hand, rather than forcing architects to adapt to the tool. Moreover, such tools should be fast. Tools that are slow, despite other advantages, will be ignored by time-pressured architects. Finally, such tools should readily connect to other tools in the environment and not be dependent on running in a proprietary programming environment.

Question from Architecture Description Group

Why is there no tool or method available to propagate design change at the architectural level?

Such tools exist, but are not yet in wide use because the technology in most cases is not yet mature. Repositories, which can be used to trace the effects of a change through much of a project's documents, are not a young technology, but their high cost, low performance, and lack of flexibility have slowed their adoption. Kazman's new SAAM Tool provides support for tracing design changes with scenarios. QFDs are used in requirements analysis to detect conflicts between requirements, and hence expose problems in design changes early. Reverse engineering tools exist solely to understand how changes will propagate through a system in terms of its design.

References

[1] B. Boehm. Anchoring the Software Process. *IEEE Software*, 13(4):73–82, July 1996.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, New York, 1996.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, editors. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[4] P. Inverardi and A.L. Wolf. Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[5] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, New Jersey, 1996.

[6] L. Vidal, A. Finkelstein, G. Spanoudakis, and A.L. Wolf, editors. *Joint Proceedings of the SIGSOFT '96 Workshops*. ACM Press, New York, New York, 1996.