SAAM: A Method for Analyzing the Properties of Software Architectures

Rick Kazman

Department of Computer Science
University of Waterloo
Waterloo, ON Canada N2L 3G1
rnkazman@watcgl.uwaterloo.ca

Len Bass, Gregory Abowd

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, U.S.A. 15213
{ljb,gda}@sei.cmu.edu

Mike Webb

Texas Instruments Inc.
Dallas, TX, U.S.A. 75265
{smw}@sei.cmu.edu

Abstract

While software architecture has become an increasingly important research topic in recent years, insufficient attention has been paid to methods for evaluation of these architectures. Evaluating architectures is difficult for two main reasons. First, there is no common language used to describe different architectures. Second, there is no clear way of understanding an architecture with respect to an organization's life cycle concerns—software quality concerns such as maintainability, portability, modularity, reusability, and so forth. This paper addresses these shortcomings by describing three perspectives by which we can understand the description of a software architecture and then proposing a five-step method for analyzing software architectures called SAAM (Software Architecture Analysis Method). We illustrate the method by analyzing three separate user interface architectures with respect to the quality of modifiability.

1 Introduction

Software architecture is a topic of growing concern within both the academic and industrial communities. Despite the popularity of this topic, little attention focuses on methods for analyzing a software architecture to show that it satisfies certain properties. Software architectures are primarily motivated by software engineering considerations, or software quality factors, such as maintainability, portability, modularity and reusability. In this paper, we address architectural description and a method of analysis with respect to software quality. The method is demonstrated by means of a case study examining architectures for developing the user interface portion of interactive systems.

Over the past decade, changes in the software architecture characterize the advances in support environments for the development of user interfaces. However, it is often difficult to assess a developer's claims of qualities inherent in a software architecture. Examples of such claims are:

> We have developed … user interface components that can be reconfigured with minimal effort. [20]

> Serpent … encourages the separation of software systems into user interface and "core" application portions, a separation which will decrease the cost of subsequent modifications to the system. [22]

> This Nephew UIMS/Application interface is better that [sic] traditional UIMS/Application interfaces from the modularity and code reusability point of views. [26]

The difficulty in assessing the validity of these claims arises for two reasons. First, the various architectural descriptions do not use a common vocabulary. When people develop new architectures, they typically develop new terms to describe them, or use old terms in a new way. It is, therefore, difficult to compare these new architectures with existing ones—there is no level playing field. Second, it is often difficult to link architectural abstractions with system development concerns. Developers tend to concentrate on the functional features of their architectures, and seldom address the ways in which their architectures support quality concerns within the system development life cycle.

The main goal of this paper, therefore, is to establish a method for describing and analyzing software architectures, called the Software Architecture Analysis Method (SAAM). Before analyzing an architecture, we must first have a way to provide a clear description of it which exposes its main features. We define three perspectives for understanding and describing architectures—functionality, structure and allocation. We also provide a simple language for describing the structural perspective. This language is important because it permits different architectures to be described consistently, forcing a common level of understanding for comparing different architectures.

After we have a way to describe an architecture, we can apply the SAAM to analyze it. The main activities involved in the SAAM are enumerated below.

1. Characterize a canonical functional partitioning for the domain.

2. Map the functional partitioning onto the architecture's structural decomposition.

3. Choose a set of quality attributes with which to assess the architecture.

4. Choose a set of concrete tasks which test the desired quality attributes.

5. Evaluate the degree to which each architecture provides support for each task.

It is a secondary goal of this paper is to demonstrate the utility of this general architectural analysis method for the evaluation of user interface architectures.

1.1 Background

Our approach to the analysis of software at the architectural level reflects a growing trend in software engineering. That trend emphasizes a better understanding of general architectural concepts as a foundation for the proof that a system meets more than just functional requirements [12]. We agree with Perry and Wolf [18] that an architectural description provides multiple perspectives of different information, though their perspectives would rightly be considered as perspectives on structure in our terms. They also promote the use of a small lexicon of elements to portray structure as do Abowd, Allen and Garlan [2] and Garlan and Shaw [8]. Our small lexicon, however, resulted more because it was sufficient for the task at hand and not out of any conviction that a small vocabulary was essential.

We are not original in our desire to predict the quality of a software product from a higher level design description. For example, Parnas [17] motivated the use of modularization or information hiding as a means of high-level system decomposition to improve flexibility and comprehensibility. Stevens, Myers and Constantine [25] introduced the notions of module coupling and cohesion to evaluate alternatives for program decomposition. The software metrics community has used these notions to define predictive measures of software quality (see, for example, the work by Briand, Morasca and Basili [6]). Our analysis method appeals to a more abstract evaluation of how the architecture fulfills the domain functionality and other nonfunctional qualities. We do not present any metrics for predictive evaluation, but instead present an example-based method for performing a more qualitative evaluation.

Garlan and Shaw demonstrated how the examination of significant architectural case studies can help to define the field of software architecture, and this is why we chose to ground our general analysis method in a concrete domain. Although we have extensive experience in the field of Human-Computer Interaction and user interface development and analysis, our main interest in writing this paper was not so much for the HCI community but for the general software engineering community interested in understanding how clear and coherent software architectures can aid in analysis. Having said that, there is a clear contribution in our work for the user interface development community. There is an abundance of literature which takes a taxonomic approach to classifying the many user interface development tools ([4], [11]). Rather than simply classify various user interface development tools, we provide a means of evaluating their suitability for producing modifiable interactive systems.

1.2 Overview

In Section 2, we define the three distinct perspectives, functionality, structure and allocation, used to describe a software architecture and we introduce the simple lexicon for the structural perspective. The remainder of the paper demonstrates the application of the SAAM to the case study of user interface software. In Section 3, we define a canonical functional partitioning for user interface software and examine how that functionality is allocated for three distinct user interface development environments. In Section 4, we discuss one quality attribute, modifiability, important for user interface software. We then identify two benchmark modification tasks which are used in Section 5 to evaluate the architectures assumed by the different development environments. In Section 6, we summarize the results of this paper and point to further work on this topic.

2 Perspectives on architectures

The architectural design of a software system can be described from (at least) three perspectives—the functional partitioning of its domain of interest, its structure, and the allocation of domain function to that structure. These perspectives reflect a consensus within the software engineering community, as witnessed by the literature ([8], [13], [21], [24]). We will discuss each of these perspectives in turn.

2.1 Functionality

A system's functionality is what the system does. It may be a single function or a bundle of related functions which together describe the system's overall behavior. For large systems, a partitioning divides the behavior into a

collection of functions which together comprise the system's function but which are individually simple to describe or otherwise conceptualize.

Typically, a single system's functionality is decomposed through techniques such as structured analysis [30] or object oriented analysis [22], but this is not always the case.When discussing architectures for a collection of systems in some common domain, such as we are doing here, the functional partitioning is often the result of a domain analysis. In a mature domain (e.g., databases, user interfaces, flight simulators, and VLSI design), the partitioning of functionality has been exhaustively studied and is typically well understood, widely agreed upon, and canonized in implementation-independent terms. Another common name for a canonical functional partitioning is a reference architecture, but we choose not to use that name for fear of overloading the term architecture.

2.2 Structure

A system's software structure reveals how it is constructed from smaller connected pieces. The structure is described in terms of the following parts:

1. A collection of components which represent computational entities (e.g., a process) or persistent data repositories (e.g., a file);

2. A representation of the connections between the components, that is, the communication and control relationships among the components.

In our analysis of architectures, we make use of a small and simple lexicon for describing structure, as indicated in Figure 1.
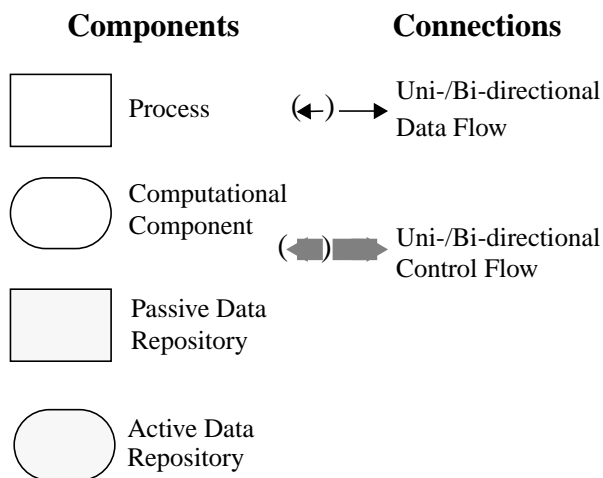

Figure 1: Architectural notations

Square-cornered boxes with solid lines represent processes, or independent threads of control. Round-cornered

boxes represent computational components which only exist within a process or within another computational component (e.g. procedures, modules). Square cornered shaded boxes represent passive data repositories (typically files). Round-cornered shaded boxes represent active data repositories (e.g., an active database). Solid arrows represent data flow (uni- or bi-directional) and grey arrows represent control flow (also uni- or bi-directional). This lexicon has been defined to meet the descriptive needs of this paper; it should not be construed as a complete vocabulary for all structural descriptions.

To understand the overall behavior of a system, we would need to provide more detailed descriptions of the computations possible within components and the overall coordination of a collection of components with various connection relationships between them. Such computational and coordination models should be explicit in an architectural description [1]. However, we do not include such information in our lexicon.

2.3 Allocation

The allocation of function to structure identifies how the domain functionality is realized in the software structure. The purpose of making explicit this allocation is to understand the way in which the intended functionality is achieved by the developed system. There are many structural alternatives and many possible allocations from function into that structure. Software designers choose structural alternatives on the basis of system requirements and constraints which are not directly implied by the system's functional description. It is the allocation of function to structure which we will be examining most closely in this paper, because the allocation choices most strongly differentiate architectures within a given domain.

3 Description of architectures for user interfaces

We can now describe a number of software architectures for user interfaces in terms of a canonical functional partitioning for the domain, the individual structural decomposition and the allocation from functionality to that structure. Once we have these architectural descriptions, we will have the basis for a more intellectually satisfying comparison of different user interface software architectures. We begin with a description of a canonical functional partitioning for this domain before describing three influential non-commercial user interface architectures—Serpent, Chiron and Garnet.

3.1 The Arch/Slinky metamodel

The Arch/Slinky metamodel of user interface architec-

tures [29] will serve as our canonical functional partitioning for this domain. It is an extension of the Seeheim model of user interface management systems [19], and is the result of wide discussion and agreement amongst user interface researchers and developers. Arch/Slinky identifies the following five basic functions of user interface software:

- *Functional Core* (FC). This component performs the data manipulation and other domain-oriented functions. It is these functions that the user interface is exposing to the user. This is also commonly called the Domain Specific Component, or simply the Application.

- *Functional Core Adapter* (FCA). This component aggregates domain specific data into higher-level structures, performs semantic checks on data and triggers domain-initiated dialogue tasks.

- *Dialogue* (D). This component mediates between the domain specific and presentation specific portions of a user interface, performing data mapping as necessary. It ensures consistency (possibly among multiple views of data) and controls task sequencing.

- *Logical Interaction* (LI) component. This component provides a set of toolkit independent objects (sometimes called virtual objects) to the dialogue component.

- *Physical Interaction* (PI) component. This component implements the physical interaction between the user and the computer. It is this component which deals with input and output devices. This is also commonly called the Interaction Toolkit Component.

In the remainder of this section, we will provide structural descriptions of the Serpent, Chiron and Garnet user interface development environments. Each of these development environments assumes a structure for application built within that system. We will first show that graphical structure assumed by the authors of the development environment. We will then recast that structure using the lexicon defined in Section 2.2. Superimposed on that structural description will be the allocation of domain function given by the Arch/Slinky canonical functional partitioning.

3.2 Serpent

Serpent identifies a dialogue controller, the presentation and the application as three distinct processes in its architecture [3], as shown in Figure 2.

Application modules contain the computational semantics required for the application. Although there can theoretically be many different applications contained within a given run-time instance of Serpent, there is typically only
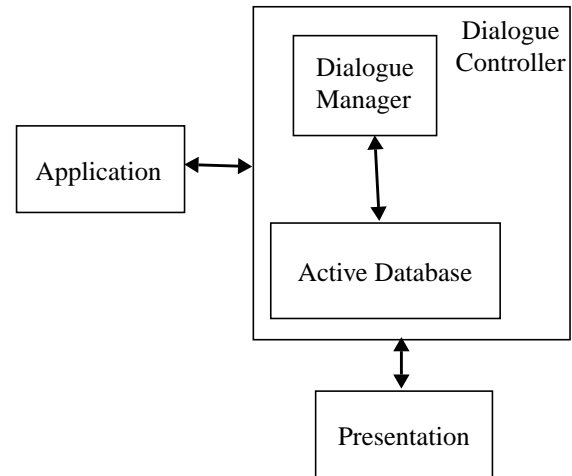

Figure 2: The Serpent architecture (adapted from [3])

one. Presentation modules provide techniques for supporting interaction at both the logical and physical level completely independent of application semantics. Different presentation modules in a given run-time instance are possible, although once again, not typical. Given that application and presentation modules are separate, there must be a way to coordinate a given application component with a presentation component. That is the purpose of the dialogue controller. The dialogue component mediates the user's interaction with an application, through the control of the presentation.

All communication between Serpent components is mediated by constraints on shared data in the database shown in Figure 2. This structure is implemented as an active database—when values in the database change, they are automatically communicated to any component which is registered as being interested in the data. This global database physically resides in the same process as the dialogue controller but is logically independent from *all* of the Serpent components. A dialogue manager sits within the dialogue controller process and mediates the connection between application and presentation. The dialogue manager is further decomposed into a collection of view controllers (not shown in Figure 2) which provide a finer grain of correspondence between application and presentation objects.

Figure 3 indicates how the five basic functional roles of a user interface architecture, as identified in section 3.1, are mapped onto Serpent's software structure. Dotted lines are used, in this and following figures, as a visual indication of the allocation of functionality to portions of the structure. This is done for analysis purposes only and should not be interpreted as having any other structural implications.

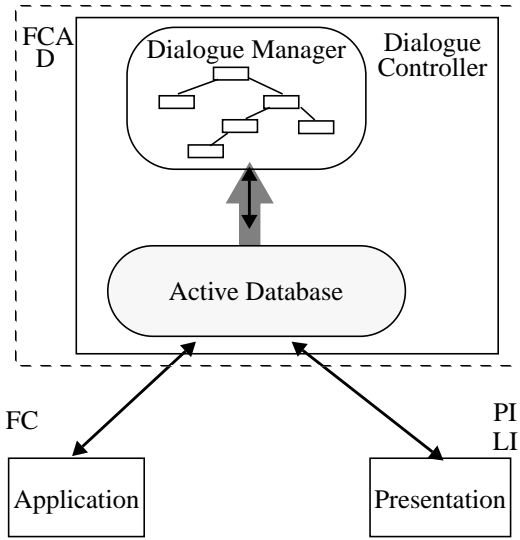Several things have changed between Figure 2 and Fig-

Figure 3: Serpent architecture (annotated)

ure 3. The architecture has been redrawn using the notation given in section 2.2, and the functional roles have been overlaid on the structure of Serpent, in order that the allocation of functionality to structure may be illustrated.

There are several points of interest to note in this recharacterization of Serpent's structure:

- there is no structural separation between the PI and LI functions in the presentation process;

- there is no structural separation between the FCA and D functions in the dialogue controller process;

- except for the dialogue manager, all of Serpent's components are monolithic—that is, they provide no *architectural* support for subdivision of functionality within a component. Within the dialogue manager, a *view controller* hierarchy provides an architectural decomposition.There are other forms of support that allow for further subdivision of functionality. For example, in

Serpent a glue language is used within the presentation process to separate physical aspects of a window system/toolkit from logical aspects. This glue, therefore, supports the subdivision of the presentation process into physical interaction and logical interaction functions.

This last point is important to remember. We are only concerned with architectural support in this paper. We do not discuss other forms of support, such as language support as exemplified by the use of glue in Serpent, though we recognize their potential importance.

3.3 Chiron

The Chiron architecture [24] was built with the expressed goals of addressing life cycle concerns of maintainability and sensitivity to environmental changes. A Chiron architecture consists of a client and a server. The client consists of an application, which exports a number of abstract data types (ADTs) which Chiron encapsulates within Dispatchers. Dispatchers communicate with Artists, which maintain abstract representations of their associated ADTs in terms of an abstract depiction library (ADL).

A Chiron server consists of the ADL, a virtual window system, and an instruction/event interpreter. The virtual window system translates abstract interface depictions into concrete ones. The instruction/event interpreter responds to requests from Artists to change the abstract description and translates those requests into changes to the presentation. It also responds to events from users and translates those back into Artist requests.

The architecture of a typical system developed under Chiron is shown in Figure 4.

The Chiron architecture clearly separates the application (functional core) from the rest of the system, as would be expected in a system which was built with the expressed goal of minimizing sensitivity to environmental
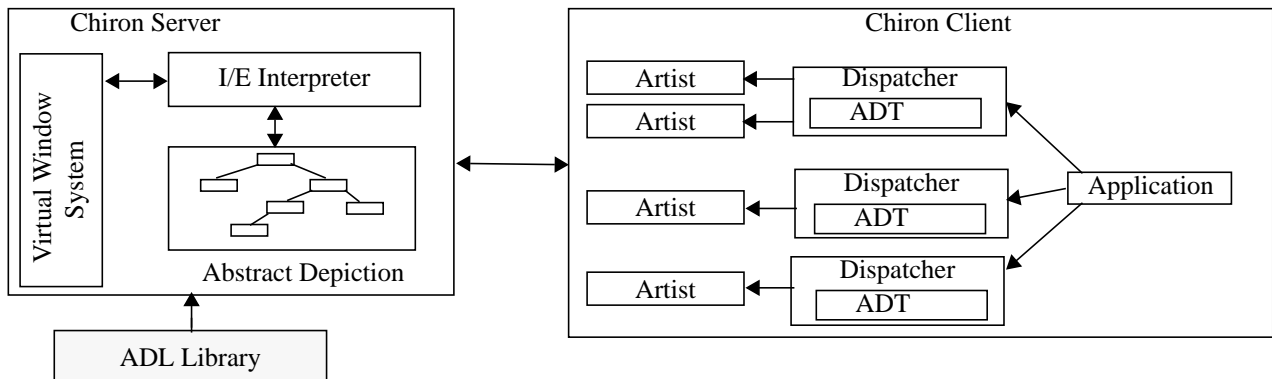

Figure 4: A typical Chiron architecture (adapted from [24])
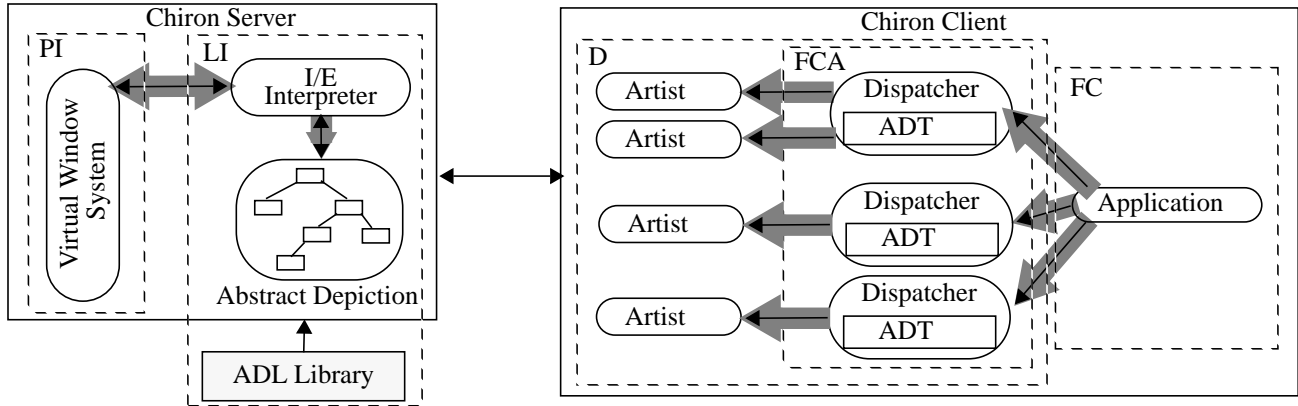
Figure 5: Chiron architecture (annotated)

changes. The dispatchers/ADTs together form the functional core adapter. It seems clear that the Artists contain some of the dialogue, for example, maintaining a correspondence between objects from the application domain and interface objects from the presentation domain. However, what is less clear, from the architectural description, is where the "state" of the dialogue lives. For example, where does one put the information that the "paste" option in an edit menu should be grayed out unless something has previously been cut or copied? Another type of dialogue issue is maintaining relationships among the interface objects. For example, when a user selects the "Save As" option in a file menu, something in the dialogue must cause a file selection box to be created. Once again, the location of these sorts of dialogue issues is not clear from Chiron's architectural description. These dependencies might exist in the Artists or the ADTs. [28]

The physical interaction component appears to be located in Chiron's Virtual Window System component, and the logical interaction component is encapsulated within the ADL. As a result of this characterization, we can provisionally annotate the Chiron architecture as shown in Figure 5.

This re-characterization indicates the logical division of functionality in Chiron, according to the Arch/Slinky meta-model. Note that by re-characterizing Chiron's architecture in this way, we can now begin to understand its relationships to other architectures, such as Serpent's. This task is considerably more difficult when trying to compare architectures based upon their own representations and claims. What we have done is to develop a common language for making architectural comparisons.

3.4 Garnet

The emphasis for systems developed under Garnet [15] is on control of the runtime behavior of interaction objects and the visual aspects of the interface. This is a different emphasis from Serpent and Chiron, which were expressly interested in maintainability and separation of concerns.

The architecture of applications built with Garnet is a single process, with Common Lisp and the X11 window system as its foundations (above an assumed operating system). The Garnet toolkit is built on top of this foundation. At the very top are high-level Garnet tools (which we call Garnet applications). Garnet is thus commonly presented as a layered system, as shown in Figure 7.
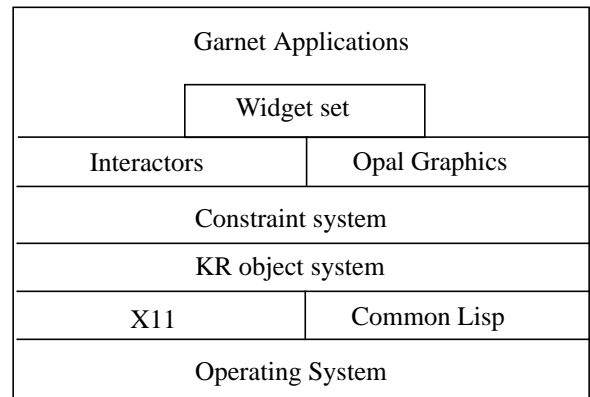

Figure 6: The Garnet architecture (adapted from [15])

However, Garnet applications are not strictly layered. In a strictly layered system, the *n*th layer hides all details of the layers *n-1* and lower, and is available only to layer *n+1*. In Garnet applications, Common Lisp, the KR object system and the Constraint system are services used directly by all layers. There are no prototypical patterns of usages for the object services. Consequently, we choose not to reveal it as an architectural features of Garnet in Figure 7, just as we have chosen not to represent the operating system in our architectural description.

The Interactors and Opal Graphics together strictly hide all of the X11 calls. Thus, Garnet widgets and applications do not have any window manager calls in them, only calls

to the interactors, object-oriented graphics, constraints and object system. [14]

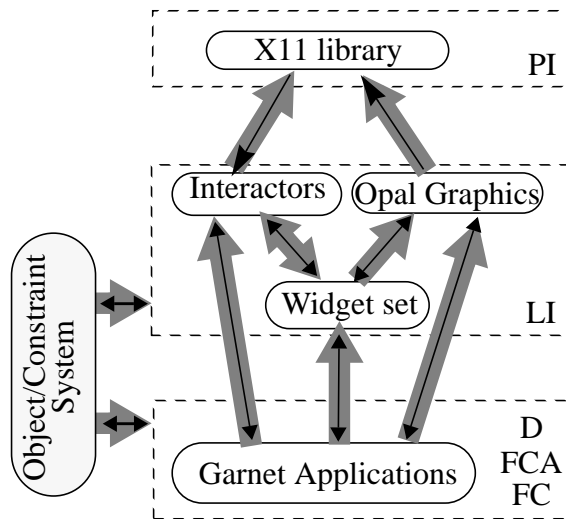Given this analysis, we can re-draw Garnet's architecture as follows:

Figure 7: The Garnet architecture (annotated)

The X11 window system fills the physical interaction function. The Widgets, Interactors and Opal graphics collectively provide the logical interaction function, making use of the object and constraint services. Finally, applications in Garnet fulfill the functions of dialogue, functional core adaptor and functional core, again utilizing object and constraint services.

This characterization of Garnet illustrates several points. Garnet applications subdivide logical interaction into three distinct parts. Dialogue, functional core adaptor and functional core are separated from the rest of the system but are not further subdivided within a Garnet application.

How an architecture partitions its functionality indicates the emphases of the architecture. The partitioning is the architectural manifestation of the stated goals of Garnet: control of the runtime behavior of interaction objects and the visual aspects of the interface (physical interaction and logical interaction components) are paramount.

4   Analyzing architectural qualities

One of our key arguments is that software architectures are neither intrinsically good nor intrinsically bad; they can only be evaluated with respect to the needs and goals of the organizations which use them. It is important to understand the life cycle concerns for a particular organization, providing a list of quality factors which can be used to discriminate between good and bad architectural alternatives for that organization.

In the Evolutionary Development Life Cycle project at Carnegie Mellon University, we are concerned with the architecture of systems which have long projected lifetimes—20-30 years. This is an increasingly important segment of the software market. According to a study by Green and Selby, the average age of all software is increasing [9]. In systems such as these, modifiability is of paramount importance. In the domain of user interface development, modifiability is frequently cited as a motivating factor for a particular architecture, since some changes are very common. For this reason, we will evaluate user interface architectures with respect to the property of modifiability.

Modifiability by itself is too abstract to be useful for analysis. In order to understand how to design for modifiability, it is necessary to better understand the ramifications of this attribute. We look at what sorts of modifications to a software system in our domain are possible. When that is accomplished, we then decide what sorts of modifications are *likely*, or representative of the domain. We evaluate each candidate architecture according to how well it supports the set of likely modifications, or benchmark tasks. If the application domain is well understood, this set of benchmark modifications can often be given a sample distribution, for the purposes of ranking the individual evaluations.

This process is akin to choosing a set of benchmark tasks to evaluate the performance characteristics of a piece of software or hardware. These modifications are intended to approximately model the type and distribution of tasks which are typical of an organization's software development life cycle. This is why a set of example modifications is required.

Oskarsson gives an informal characterization of classes of modifications in [16]. Drawing upon his work, we have enumerated the following classes of modifications:

- *Extension of capabilities*: adding new functionality, enhancing existing functionality;

- Deletion of unwanted capabilities: e.g. to streamline or simplify the functionality of an existing application;

- *Adaptation to new operating environments*: e.g. processor hardware, I/O devices, logical devices

- *Restructuring*: e.g. rationalizing system services, modularizing, optimizing, creating reusable components.

In the user interface domain, two classes of modifications prove most common—adaptation to new operating environments and extensions of capabilities. The other classes of modifications—restructuring and deletion of unwanted capabilities—do not constitute a significant percentage of the modifications for user interfaces. We will not include them in our set of modifications.

Adaptation to new operating environments is a common modification activity which user interface architectures must support. For example, this paper is being composed using a desktop publishing system which is available on Unix-based workstations running the X window system, Macintoshes running the Macintosh toolkit, and IBM-compatible PCs, running MS-Windows. In each of these cases, the underlying functionality of the system is identical. What does change are the devices, both logical and physical, used to interact with the program.

The first of our benchmark modification tasks, therefore, is to change the toolkit. For example, a move from using Motif to Fresco would be typical of this category of modification.

Because user interface development is highly iterative, *extensions of capabilities* (adding new features, reorganizing the appearance of the interface, reorganizing the human-computer dialogue) occur frequently. This is particularly so during the initial development of a user interface, but such modifications are also common later in the life cycle.

The user interface life cycle is distinguished from typical software engineering life cycles because it is highly iterative, relying more on prototyping and empirical testing and validation. Requirements for human-computer interaction are often not well understood in advance, and so iteration and prototyping are often the only ways in which to evolve a system's design. Thus this class of change is the most common (and most costly overall) in the user interface life cycle.

We choose as our second benchmark task a modification to the human-computer dialogue. More specifically, the second benchmark modification is to add a single option to a menu, reflecting some piece of application functionality which must be made available to a user.

5   Architectural analysis for UI tools

Now that we have enumerated our set of benchmark modifications—changing the window system/toolkit and adding a single option to a menu—we are in a position to evaluate the degree to which each of our candidate systems provides *architectural* support for these modifications.

5.1   Serpent

5.1.1   Changing the toolkit

Changing the toolkit in Serpent assumes that we have an application running under one toolkit and we want to change to another toolkit which is not currently supported within the Serpent run-time environment. For example, this situation would occur if we had Motif running in a Serpent presentation process and we want to replace it with OpenLook. What would have to change is both the presentation process and the dialogue.

There are two ways this change can be accommodated in Serpent. If the intended use of the new toolkit is simply to reproduce the look and feel of the old toolkit, then the only modification necessary is to change Serpent's glue code in the presentation process. As discussed in Section 3.2, the glue code provides the logical interaction function for Serpent, but is not isolated in its architectural description, so we conclude that Serpent provides no architectural support for this kind of modification. If the intended use of the new toolkit is to provide new interaction capability, then modifications to the dialogue manager would be necessary, in addition to new glue code.

We can see from this analysis the purpose of the logical interaction function in the Arch/Slinky canonical partitioning. Logical interaction provides a buffer between the dialogue and the physical interaction in order to isolate the effect of changes between them. Architectures that isolate the logical interaction function provide support for this benchmark modification. Serpent, therefore, does not provide architectural support for this modification.

5.1.2   Adding a menu option

Serpent has isolated the dialogue controller, so it is easy to say that the second modification type—adding an option to a menu—will occur somewhere in that process. The dialogue controller process contains two parts, the dialogue manager, which governs the behavior of the dialogue and the active database, which maintains the state of the dialogue. The benchmark modification concerns a change in the behavior of the dialogue, so it is isolated in the dialogue manager. View controllers further subdivide the dialogue manager so that it easier to isolate a specific piece of dialogue behavior, such as a menu. Therefore, we conclude that Serpent provides adequate architectural support for this modification.

5.2   Chiron

5.2.1   Changing the toolkit

Chiron goes one step beyond Serpent in providing architectural support for this modification. Chiron has isolated the logical and physical interaction functions in separate structures within a Chiron server. Since the Virtual Window System only communicates with the Instruction/ Event Interpreter and its associated Abstract Depiction Library, it is isolated from the rest of the architecture. This architectural isolation means that the Virtual Window System and Instruction/Even Interpreter communicate via a well-defined interface. The existence of such an interface means that different toolkits could be inserted into the

architecture as long as they comply with the interface conventions.

This strict separation of concerns aids modifiability, by localizing the effects of a change. Thus, we can conclude that the Chiron architecture provides significant support for this benchmark modification.

5.2.2 Adding a menu option

Chiron fares slightly less well with respect to the second modification. As stated in section 3.3, the division of dialogue responsibilities between Artists and ADTs is not precisely specified. Hence, a modification to the dialogue may manifest itself as a change to an Artist or an ADT, or both. If the ADTs are well-structured, changes to them should be minimal, in which case a change to the dialogue would be isolated to the Artists.

For this particular modification, it is reasonable to assume that no change to an ADT would be necessary since we are assuming that we want to make available some pre-existing application function which would already be defined in one ADT. The modification in this case is restricted to an Artist which is associated to the ADT containing the desired function. We conclude that Chiron provides architectural support for this benchmark modification.

5.3 Garnet

5.3.1 Changing the toolkit

Garnet has isolated the toolkit to three components—interactors, Opal graphics and widgets—and has hidden any X11 dependencies behind them. Garnet has strictly separated physical interaction, and so we conclude that this modification is supported by the architecture.

5.3.2 Adding a menu option

A dialogue in Garnet is monolithic, and can involve any of the language services which Garnet provides. Thus, changing a menu in Garnet involves locating and modifying the affected Lisp code, which may be a difficult task in a complex interface. The second modification is not architecturally supported by Garnet. Garnet does provide tool and language support for this class of change, but as we stated earlier, that is not the concern of this paper.

6 Conclusions and future work

In this paper, we have provided a architectural analysis method (SAAM) and used it to evaluate user interfaces architectures with respect to modifiability. This method is based upon a common understanding and representation for architectures and an analysis of an organization's life-cycle requirements. SAAM permits the comparison of architectures within the context of an organization's particular needs. This sort of comparison has been hitherto quite difficult.

The SAAM places strong demands on an organization to articulate those quality attributes of primary importance. It also require a selection of benchmark tasks with which to test those attributes. The purpose of the SAAM is not to criticize or commend particular architectures, but to provide a method for determining which architecture supports an organization's needs.

We applied the method to evaluate user interface development environments to determine their architectural support for two benchmark modifiability tasks. We saw that the benchmark tasks could be cast in terms of the Arch/ Slinky metamodel. Therefore, systems with a clear allocation of that functional partitioning to their structure could provide architectural support for the modifications. We emphasize that architectural support for a quality attribute is only one possible type of support. We may also have language or tool support. We recognize that an analysis of these other types of support is an important topic for future research.

Quality attributes is a recognized driver for software architectural design, outside of the user interface domain (e.g. [1]). Consequently, we believe SAAM will work for other attributes. This method of evaluation shows the degree to which an architecture was *designed* to support selected quality attributes. We see a strong link between quality attributes and canonical partitionings. More than one canonical partitioning is possible for a given domain, each accommodating a different set of quality attributes [5]. For example, the multiagent PAC model [7] is a candidate canonical partitioning which emphasizes scalability and construction efficiency.

Finally, what is really desirable is to have metrics for more precisely evaluating attributes in terms of architectures, but our understanding of this topic is not yet sufficient to define any measurements. For now, we simply have ways to analyze, compare and rank architectures. We believe that the work of Henry and Kafura [10] on information flow points the way to an analysis technique for architectures, but it must be augmented by techniques for measuring the understandability and consistency of architectures.

7 Acknowledgments

8 References

[1] Abowd, G., Bass, L., Howard, L., Northrop, L. "Structural Modeling: an Application Framework and Development Process for Flight Simulators". Software Engineering Institute, Carnegie Mellon University Technical Report CMU-SEI-TR-93-14. Pittsburgh, PA, 1993.

[2] Abowd. G., Allen, R., Garlan, G. "Using Style to Understand Descriptions of Software Architectures". *Software Engineering Notes*, 18(5):9-20, 1993. Proceedings of SIGSOFT '93: Symposium on the Foundations of Software Engineering.

[3] Bass, L., Clapper, B., Hardy, E., Kazman, R., Seacord, R. "Serpent: A User Interface Management System". *Proceedings of the Winter 1990 USENIX Conference*, Berkeley, CA, January 1990, 245-258.

[4] Bass, L. and Coutaz, J. *Developing Software for the User Interface*, Addison-Wesley, 1991.

[5] Bass, L. Kazman, R. and Abowd, G. "Issues in the Evaluation of User Interface Tools". *Proceedings of the ICSE Workshop on Software Engineering and Human-Computer Interaction*, Sorrento, Italy, May, 1993. Forthcoming

[6] Briand, L.C., Morasca, S. and Basili, V.R. "Measuring and Assessing Maintainability at the End of High Level Design", *Proceedings of the IEEE conference on Software Maintenance*, Montreal, Canada, 1993.

[7] Coutaz, J. "PAC, An Implementation Model for Dialog Design", *Proceedings of Interact '87*, Stuttgart, September, 1987, 431-436.

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

[9] Green, J., Selby, B. "Dynamic Planning and Software Maintenance: A Fiscal Approach", Naval Postgraduate School, Monterey, CA, NTIS Report AD-A112 801/6, 1981.

[10] Henry, S., Kafura, D. "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering, SE-7(5), Sept. 1981.

[11] Hix, D. "Generations of User-Interface Management Systems", *IEEE Software*, 7(5):77-87, 1990.

[12] Kazman, R., Abowd, G., Bass, L., Webb, M., "Analyzing the Properties of User Interface Software Architectures," Carnegie Mellon University, School of Computer Science Technical Report CMU-CS-93-201, 1993.

[13] *Military Standard, Defense System Software Development* (DOD-STD-2167A). Washington, D.C.: United States Department of Defense, 1988.

[14] Myers, B., Giuse, D., Dannenberg, R., *et al.*, "The Garnet Reference Manuals", School of Computer Science, Carnegie Mellon University Technical Report CMU-CS-90-117-R3, 1992.

[15] Myers, B., Giuse, D., Dannenberg, R., *et al*. "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces", IEEE Computer, 23(11): 71-85.

[16] Oskarsson, Ö. "Mechanisms of Modifiability in Large Software Systems", Linköping Studies in Science and Technology Dissertations No. 77, 1982.

[17] Parnas, D.L. "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15(12):1053-1058, 1972.

[18] Perry, D., Wolf, A. "Foundations for the study of software architecture", *SIGSOFT Software Engineering Notes*, 17(4), October 1992, 40-52.

[19] Pfaff, G. (ed.). *User Interface Management Systems*. New York: Springer-Verlag, 1985.

[20] Pittman, J., Kitrick, C. "VUIMS: A Visual User Interface Management System", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology,* Snowbird, UT, October 1990, 36-46.

[21] Pressman, R. *Software Engineering: a Practitioner's Approach*, 3rd edition. New York: McGraw-Hill, 1992.

[22] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenson, W. *Object-Oriented Modeling and Design*, Englewood Cliffs, N.J.: Prentice-Hall, 1991.

[23] SEI, "Serpent Overview," SEI Technical Report CMU/SEI-89-UG-2, Carnegie Mellon University Software Engineering Institute, August 1989.

[24] Sommerville, I. *Software Engineering*, 4th edition. Reading, MA: Addison-Wesley, 1992.

[25] Stevens, W.P., Myers, G.J. and Constantine, L.L. "Structured design" *IBM Systems Journal*, 13(2):115-139, 1974.

[26] Szekely, P. "Standardizing the Interface Between Applications and UIMSs", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology,* Williamsburg, VA, November 1989, 34-42.

[27] Taylor, R., Johnson, G. "Separations of Concerns in the Chiron-1 User Interface Development and Management System," *Proceedings of InterCHI '93,* Amsterdam, May 1993, 367-374.

[28] Taylor, R. personal communication, July 1993.

[29] UIMS Tool Developers Workshop, "A Metamodel for the Runtime Architecture of an Interactive System", SIGCHI Bulletin, 24(1), 32-37.

[30] Yourdon, E. *Modern Structured Analysis*, Englewood Cliffs, N.J.: Prentice-Hall, 1989.