Analyzing the Properties of User Interface Software

Rick Kazman, Len Bass, Gregory Abowd, Mike Webb

October 1993
CMU-CS-93-201

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Software architecture is an increasingly important research topic and in this report we investigate the potential role of architecture in evaluating the properties of a system built to a particular architecture. Currently such architectural analysis is complicated for two main reasons: authors of new architectures describe their creations in idiosyncratic terms; and there is no clear way of understanding an architecture with respect to an organization's life cycle concerns—efficiency, maintainability, modifiability, and so forth. This report addresses these shortcomings by proposing a domain-based method for analyzing software architectures called SAAM (Software Architecture Analysis Method). This method contains several steps. A canonical functional partitioning for the domain is adopted. Next, some candidate architectures in this domain are described in a common and simple structural language, providing a neutral context in which to understand their similarities and differences. Next, life cycle concerns for the quality of the resultant software are determined and a set of benchmark tasks are created which embody these concerns. Finally, the architectures are evaluated and compared with respect to how well they support the benchmark tasks. This report illustrates the method by analyzing user interface architectures with respect to the quality of modifiability.

Rick Kazman is affiliated with the University of Waterloo and Carnegie Mellon University; Len Bass and Gregory Abowd are affiliated with Carnegie Mellon University; Mike Webb is affiliated with Carnegie Mellon University and Texas Instruments Inc.

This work was sponsored in part by the U.S. Department of Defense and monitored under ARPA grants F19628-90-C-003 and N00014-93-1-1119.

Keywords: Software architecture , Method of software analysis , User interface software

1 Introduction

Software architecture is a topic of growing concern within both research and industrial circles. This report investigates what is needed in an architectural description of a system in order to evaluate it with respect to some organizational or life cycle qualities.We restrict the architectural analysis to systems within a common domain and within that domain we only focus on one of many qualities that might be satisfied. In this introduction we will first examine why architectural analysis is difficult and how our method of evaluation can address these difficulties. We will then define three perspectives on software architecture as the basis for describing different tools in a common architectural language. Given this language for describing architectures, we formulate representative tasks as benchmarks to judge an instance of an architecture with respect to the software quality of concern.

Throughout this paper we will use the domain of user interface software as an example to demonstrate the architectural perspectives and the method of using benchmarks to assess the quality of an architecture. Architectural considerations in this domain are important as much of the progress over the past decade in the development of user interfaces can be characterized as changes to the software architecture for the user interface portion of systems.

User interface architectures are typical of architectures in other areas since they are primarily motivated by organizational or life cycle qualities. Examples of such qualities are maintainability, portability, modularity, reusability, and so forth. However, it is often difficult to assess the various claims made by the developers of these architectures; claims such as:

> We have developed … user interface components that can be reconfigured with minimal effort. [22]

> This [model] allows the UIMS to be simple and independent of the graphics software and hardware as well as the data representation used by the application program. [32]

> 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. [25]

> This Nephew UIMS/Application interface is better that [sic] traditional UIMS/Application interfaces from the modularity and code reusability point of views. [28]

The difficulty in assessing the validity of these claims arises for a number of reasons. When people develop new architectures, they typically develop new terms to describe them, or use old terms in a new way. As a result, comparison with existing architectures becomes difficult—there is no level playing field. Furthermore, linking architectural abstractions with life cycle concerns is problematic. Developers tend to concentrate on the functional features of their architectures, and seldom address the ways in which their architectures support concerns within the system development life cycle. Yet another problem is the level of indirection that can arise because a description for a tool is provided, but what we really want to assess is the quality of instances of systems designed with that tool. It is sometimes difficult from the literature to extract what architectural assumptions the developers of a tool have placed on the systems which that tool will construct. In proposing a method to evaluate a tool, we actually mean in this report that we want to evaluate the things the tool helps a designer to build.

The main goal of this paper is to establish a method for describing and analyzing software architectures, Software Architecture Analysis Method (SAAM). The main objectives of the SAAM are to:

1. provide a clear description, in a common language, of the proposed software architecture that exposes its main features and functionality;

2. within a particular organizational context, understand the life cycle concerns surrounding the use of any systems developed within the proposed software architecture, revealing which non-functional qualities of the software are most important;

3. introduce benchmark tasks as concrete instances to test the life cycle activities which the software architecture must support;

4. evaluate the software architecture by examining its performance with respect to the benchmark tasks.

In Section 3, we define three perspectives of a software architectural description: the functional partitioning, the structural composition, and the allocation of function to structure. One of our main concerns for describing architectures is the adoption of a common language that can be used to define all architectural instances. Defining these three perspectives is a step in the direction of defining such a common architectural language.

A secondary goal of this paper is to demonstrate the utility of the SAAM for the evaluation of user interface architectures. In Section 4, we examine several candidate functional partitionings for this domain (the Seeheim model [21], the PAC model [4] and the Arch/Slinky model [31]) and choose the latter as the reference architecture to examine the structural composition and allocation for various influential user interface architectures in the research community (Serpent [1], Chiron [26], and Garnet [12]).

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. Understanding the life cycle concerns for a particular organization provides a list of quality factors that can be used to discriminate between good and bad architectural alternatives for that organization. For example, 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 [8]. In systems such as these, modifiability is of paramount importance. For this reason, we evaluate user interface architectures with respect to the property of modifiability. In Section 5 we discuss quality attributes further and examine modifiability more closely, developing two benchmark tasks in the user interface domain which shape our analysis. The use of modifiability is an example, albeit an important one in the domain of user interfaces, chosen to demonstrate the analysis method. Section 6 completes the demonstration of our analysis method as we compare the candidate user interface architectures with respect to their support for the benchmark tasks.

2 Related Work

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. Perry and Wolf argue that an architectural description provides multiple perspectives of different information, though their perspectives would rightly be considered as perspectives on structure alone in our work [20]. They promote the use of a small lexicon of elements to portray structure as do others [6], [2]. In developing such a lexicon in this report, our concern was for adequacy and not completeness.

Garlan and Shaw have demonstrated how the examination of significant architectural case studies can help to define the field of software architecture [6]. 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. We offer this paper as a case study of software architecture analysis. Those with different architectural concepts, tools and methods might use the example of user interface software architectures to demonstrate the benefits of their approach. In this way, the problem of evaluating user interface architectures could be seen as a touchstone similar to the dining philosophers problem in models of concurrency, or the specification of elevators in formal specification methods. While software architecture description already has a problem domain that is being adopted as a reference task (compiler design), the user interface architecture domain fulfills a different need—that of a reference task for software architecture evaluation methods.

Though we present the analysis of user interface software as only an auxiliary result of this paper, we do feel it is necessary to relate our work to prior analysis work in user interface architectural analysis. Lane provided a design space for describing various user interface architectural solutions and rules for choosing which architectural alternatives best suit a given set of design criteria [13]. His objectives were similar to ours in that he wanted to be able to select the right architecture based on desired quality attributes. However, his method was very different from ours, as his results were based on an expert system rule base which codified empirical evidence of heuristic design decisions made by experienced user interface developers. The results we want to provide will be based on reasoned argumentation, and on demonstrated performance on a set of benchmark tasks. One could argue that our results do not extend Lane's work in the domain of user interface architectures. That would, however, be missing the main point of our paper: how to carry out an architectural analysis in any domain.

3 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:

> [The software architecture level of design addresses] structural issues such as gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; composition of components; scaling and performance; and selection among design alternatives. [6]

> Architectural design is the activity of partitioning the requirements to software subsystems. [26]

> Software architecture alludes to two important characteristics of a computer program: (1) the hierarchical structure of procedural components and (2) the structure of data. Software architecture is derived through a partitioning process that relates elements of a software solution to parts of a real-world problem implicitly defined during requirements analysis. [23]

> [System architecture] shall describe the internal structure of the system. The segments … shall be identified and their purpose summarized. The relationships among the segments … shall be described. [14]

We will discuss each of these perspectives in the next three subsections.

3.1 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 logically separated functions that together comprise the overall system function but individually are simple to describe or otherwise conceptualize.

Typically, a single system's functionality is decomposed through techniques such as structured analysis [33] or object oriented analysis [23], but this is not always the case. When discussing architectures for a broader domain, such as we are doing here, the functional partitioning is often the result of a domain analysis for a class of similar systems in that domain. One sign of a mature domain, such as databases, user interfaces, flight simulators, ATMs, and VLSI design, is that over time some function partitionings emerge that a community of developers adopt. That is, the partitioning of functionality has been exhaustively studied and is typically well understood, widely agreed upon, and canonized in implementation-independent terms. In Section 7, we discuss some assumptions about domain analysis and functional partitioning which would lead us to suggest that more than one partitioning might be needed to evaluate different qualities.

3.2 Structure

We divide a system's software structure into the following parts:

1. A collection of components which represent computational entities, e.g., modules, procedures, or processes, or persistent data repositories;

2. A representation of the connections between the computational entities, i.e., the communication and control relationships among the components;

Figure 1 shows the conventions we use in this report, adapted from [1], to represent the structure of a piece of software. 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 thick grey arrows represent control flow (also uni- or bi-directional).

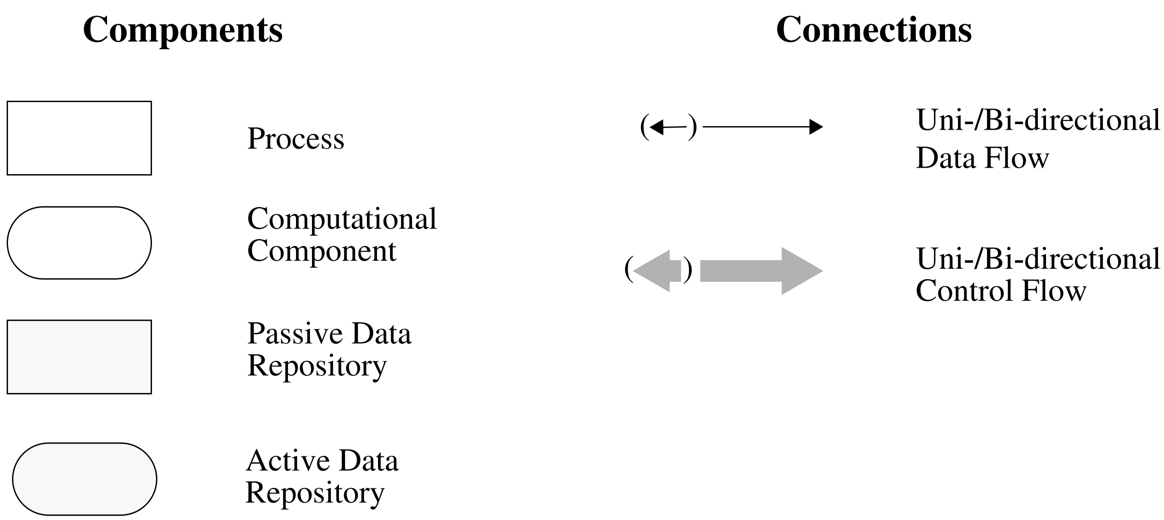

Figure 1: Architectural notations

To understand the overall behavior of a system, we 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 and described separately [1]. This is not our concern in this paper, but has been addressed elsewhere [2].

3.3 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.

4 Description of Architectures for User Interfaces

With this background, we can now describe a number of software architectures for user interfaces in terms of a functional partitioning, structure and allocation of function to structure. Examining the literature on user interface architectures reveals that such architectures fall into two classes, depending on whether or not the architecture describes an actual implementation of a development system. Those architectures that do not describe an actual implementation usually serve as reference architectures for the entire domain of user interfaces; that is, they are sufficiently abstract so as to apply to a wide variety of implementations. These reference architectures (Seeheim, PAC and Arch/Slinky) concentrate on the essential features of any architecture in the domain of user interfaces. In our language, these reference architectures serve as canonical functional partitionings for this domain. Though they do imply certain structural constraints on a software architecture (for instance, see the remarks in Section 6.4), that is not their main contribution. We adopt one of them (Arch/Slinky) as the canonical functional partitioning for this domain (though we consider the ramifications of selecting PAC in Section 6.4).

The second category of architectures address actual implementations of user interface development systems, so we can describe their structure and allocation (with respect to the Arch/Slinky functional partitioning). The complete architectures we describe here are Serpent, Chiron and Garnet. We expect that, over time, many other alternative architectures will be added to our discussion, including more academic systems, such as UIDE [27] and Rendezvous [11], and more commercially recognizable systems, such as Hypercard [7] and Visual Basic [5].

4.1 The Seeheim model

The term UIMS (User Interface Management System) was given wide exposure in a workshop in Seeheim, Germany. Consequently, the model upon which most UIMSs are based is known as the *Seeheim model* [21]. This model shown in Figure 2.

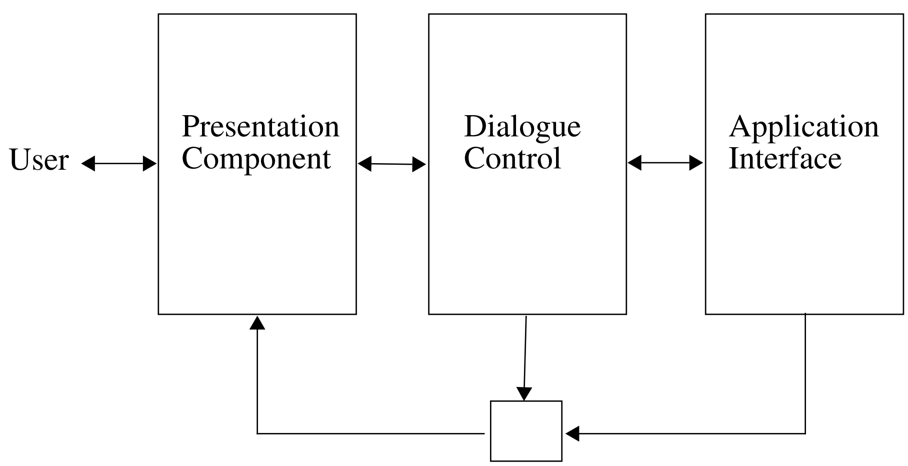

Figure 2: The Seeheim Model

The three main components in this model are: presentation, dialogue and application interface. The dialogue component is the least familiar. It is essentially an organizer of traffic between the presentation and application-specific components. It is responsible for:

- maintaining a correspondence between objects from the application domain and interface objects from the presentation domain;

- maintaining relationships among the interface objects;
- maintaining a dynamic representation of the state of the dialogue between the other two components; and
- defining two communication protocols, one suitable for the functional core, and the second for the presentation.

Dialogue components are sometimes modelled with special purpose languages [3], or make use of special functions for coordination and control which are layered on top of an existing language ([15], [26]).

The application interface component performs those functions that are inherent to the application, irrespective of the presentation of those functions. This is typically written in a high-level language such as C, Fortran or Ada.

The presentation layer controls the end user interactions and generates device-level feedback for the user. Typically, a UIMS will support some high-level user interface toolkit, such as the X toolkit, Sun's OpenLook toolkit or the Macintosh toolkit.

The Seeheim model does not provide a complete description of a user interface architecture. The model was presented as a *logical model* and so fits our description of a functional partitioning. It has withstood the test of time and may therefore be deemed a canonical partitioning for this domain. Its definition is not complete enough to uncover structural decisions, though there certainly are some structural relationships that we can infer. It is confusing, therefore, that we have chosen to portray it graphically in a language that is similar to the structural language, that is, with boxes and arrows. The reason for this is mainly historical.

4.2 The Arch/Slinky Metamodel

The Arch/Slinky metamodel of user interface architectures [31] is an extension of the basic Seeheim model and identifies the following five basic components 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. The FCA is similar to the application interface in the Seeheim model.
- *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.

The Arch/Slinky model is also a functional partitioning, but is both more explicit about its status as such, and more fine-grained than the Seeheim model. The Arch/Slinky functional partitioning will be adopted throughout the remainder of this paper as the canonical functional partitioning through which other architectures are examined.

4.3 PAC

The PAC conceptual model for interactive systems is understood as follows: an interactive system is represented as a collection of Presentation-Abstraction-Control (PAC) agents which are related in a somewhat (though not necessarily strict) top-down hierarchical structure. Each PAC agent triple represents three tightly connected components. The abstraction component contains some application relevant piece of the system, the presentation component some interaction relevant piece and the control component maintains a presentation-abstraction correspondence within the agent. Control components are also responsible for maintaining compatibility between various PAC agents related in the PAC hierarchy. Figure 3 shows a typical PAC structure.

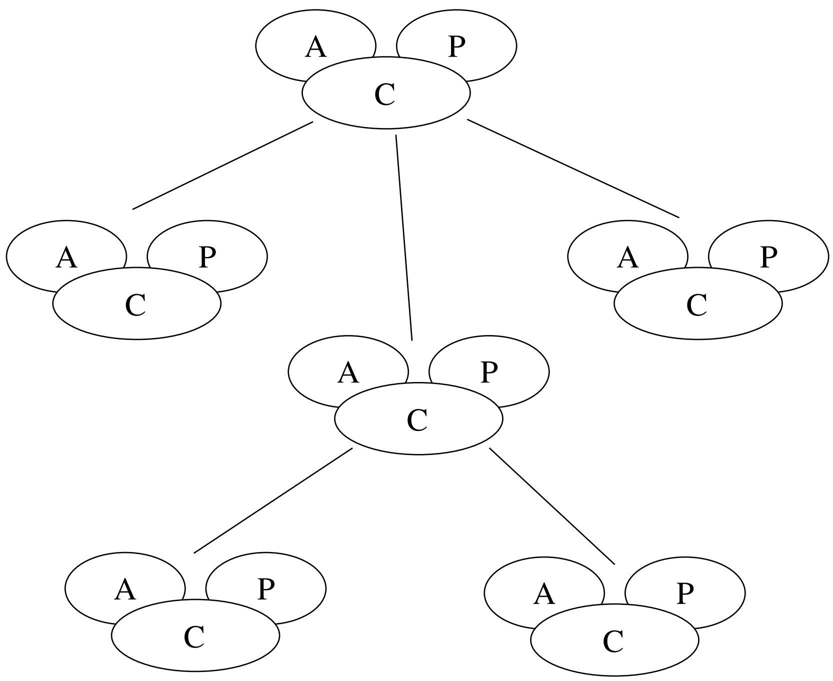

Figure 3: A typical PAC structure

PAC does not represent a tool for constructing user interfaces; it is still just a conceptual model. Because PAC is closer to being a true software architecture than either Seeheim or Arch/Slinky, we can make an attempt at describing its structure and allocation (with respect to the Arch/Slinky partitioning). In our analysis, we must assume that the PAC hierarchy is the complete description of the run-time system, so it must contain all of the roles of the interactive system from functional core to physical interaction. Later models of PAC (e.g., the PAC-AMODEUS model [17]) have relaxed this constraint, viewing the PAC model as a way of decomposing the dialogue component, which sits within the larger context of the Arch/Slinky metamodel.

Thus, in order to evaluate PAC as a complete software architecture, we must postulate a software structure and a mapping of functionality to this structure. We have done this in consultation with an expert in the use of PAC [18]. Our interpretation of PAC as a complete software architecture is given in Figure 4. In this figure, we have indicated the functional roles of each part of the PAC hierarchy, according to the Arch/Slinky functional partitioning. In this figure we have redrawn the components, and indicate the data and control relationships between the various components using the notation defined in Figure 1.

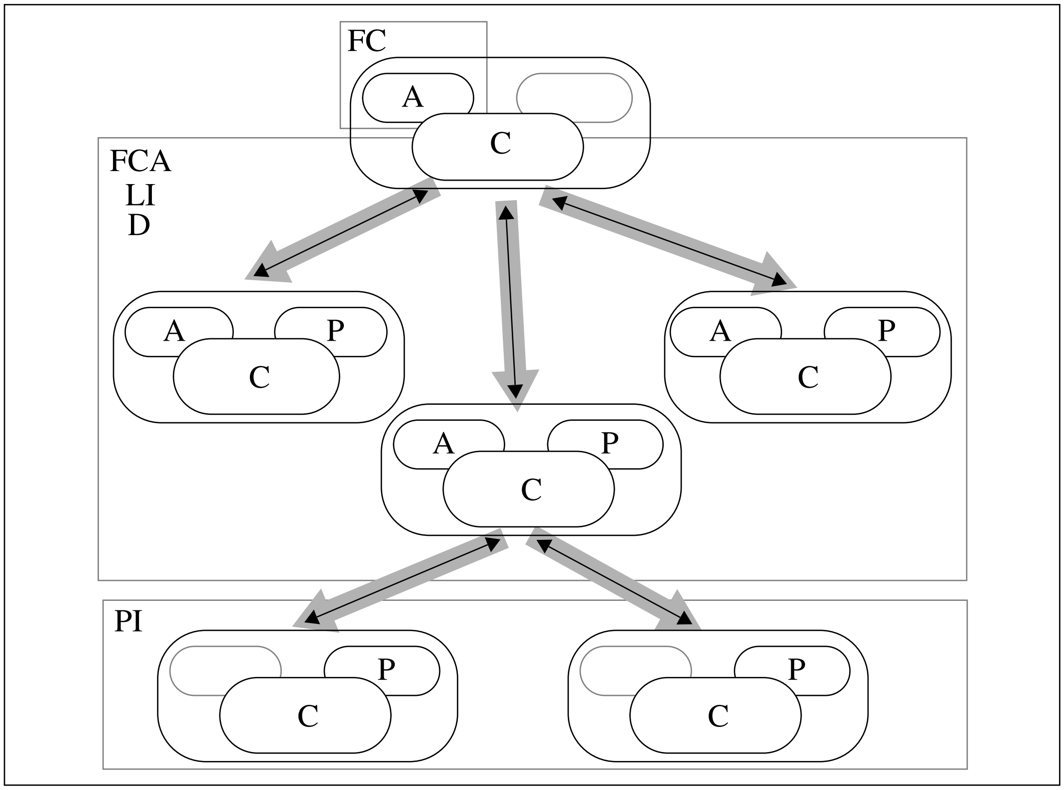

Figure 4: PAC architecture with functional roles annotated

The application component of the topmost (root) PAC agent is the functional core of the system. There is no presentation component in this agent, because the presentation is decomposed in the PAC hierarchy. This is, in fact, a key motivation for PAC—to be able to decompose a presentation and its accompanying dialogue, and to relate pieces of dialogue to pieces of presentation in a 1:1 fashion. Separation of concerns was not a motivation in PAC, which is why PAC agents are tightly coupled (as shown by their being enclosed within a common computational component). This is also why more than one functional role may exist in a single PAC agent, or a number of functional roles may be spread among a number of PAC agents.

Another way of looking at this partitioning is that for all non-root PAC agents in the system, their main purpose is to partition the remaining functional roles. Presentation components at the leaves of the hierarchy fulfill the role of physical interaction. These leaf PAC agents would not necessarily require any application component to be present, as is depicted. Responsibility for dialogue control is distributed among the control components throughout the PAC hierarchy. The functional core adaptor and the logical interaction components are not explicitly represented in PAC, although they do exist explicitly in the PAC-AMODEUS model.

4.4 Serpent

The three types of components which are identified in the Seeheim model of a UIMS and which have been realized as distinct processes in the Serpent architecture [3],are shown in Figure 5: a dialogue controller, a presentation and an application.

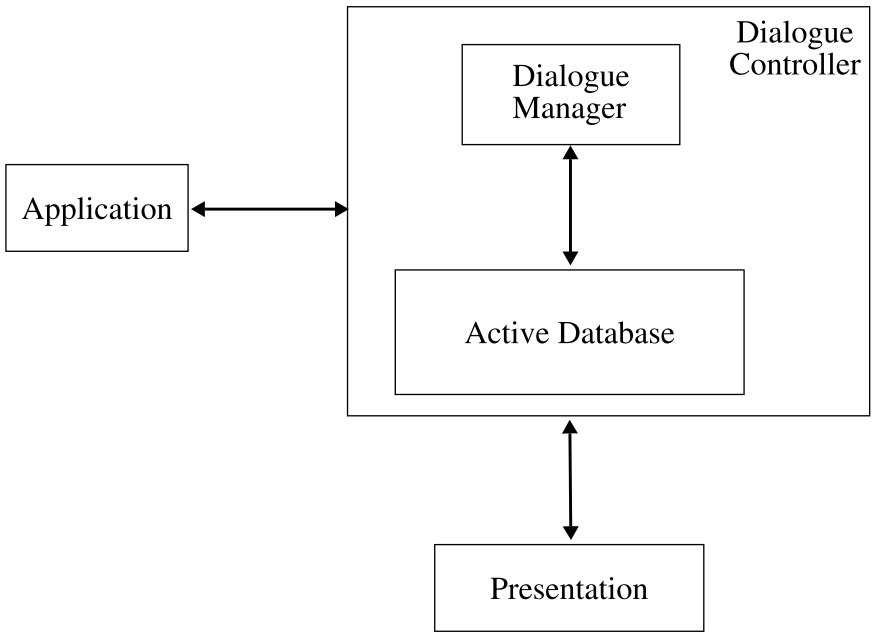

Figure 5: The Serpent architecture (adapted from [3])

Application modules contain the computational semantics required for some domain application. Although there can theoretically be many different applications contained within a given run-time instance of Serpent, there is typically only a single application. Presentation modules represent independent 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 5. 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.

Figure 6 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.[1]

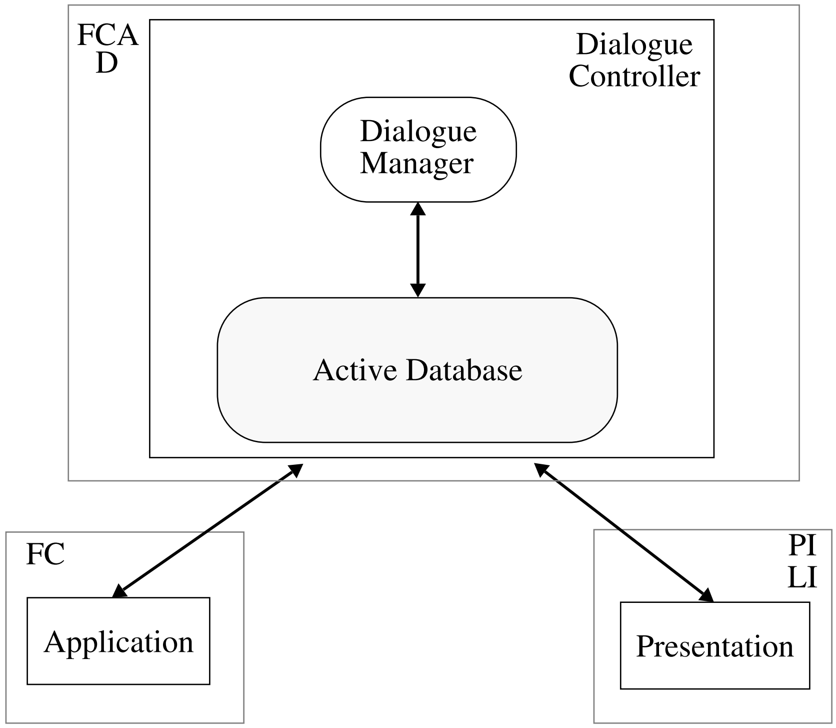

Figure 6: Serpent architecture with functional roles annotated

There are several points of interest to note in this re-characterization of Serpent's structure:

- there is no structural separation between the PI and LI roles in the presentation component;
- there is no structural separation between the FCA and D components in the dialogue controller component;
- all of Serpent's components are monolithic—that is, they provide no *architectural* support for subdivision of functionality within a component.

It is important to note that architectural support is only one type of support that might be provided for subdivision of functionality. A system may provide language, design or tool support for this purpose, for example. We do not deal with these topics in any detail here, but recognize them as important outstanding research areas.

4.5 Chiron

The Chiron architecture [29] was built with the expressed goals of addressing life cycle concerns of maintainability and sensitivity to environmental changes. A Chiron architecture consists of a

1. 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 further significance.

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, which translates from abstract interface depictions into concrete ones; and an instruction/event interpreter. 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.

A typical Chiron run-time architecture is shown in Figure 7.

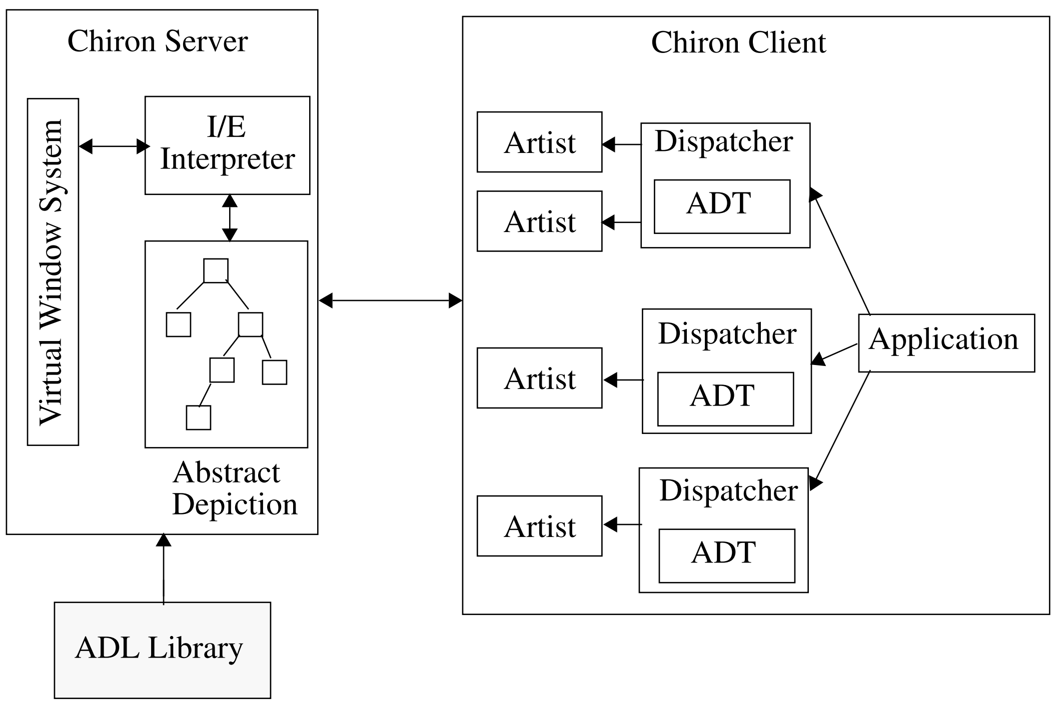

Figure 7: A typical Chiron architecture (adapted from [29])

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 changes. The functional core adapter could live in the ADTs, or in the Artists. 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, in the ADTs or even in the Abstract Depiction Libraries [30].

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 8.

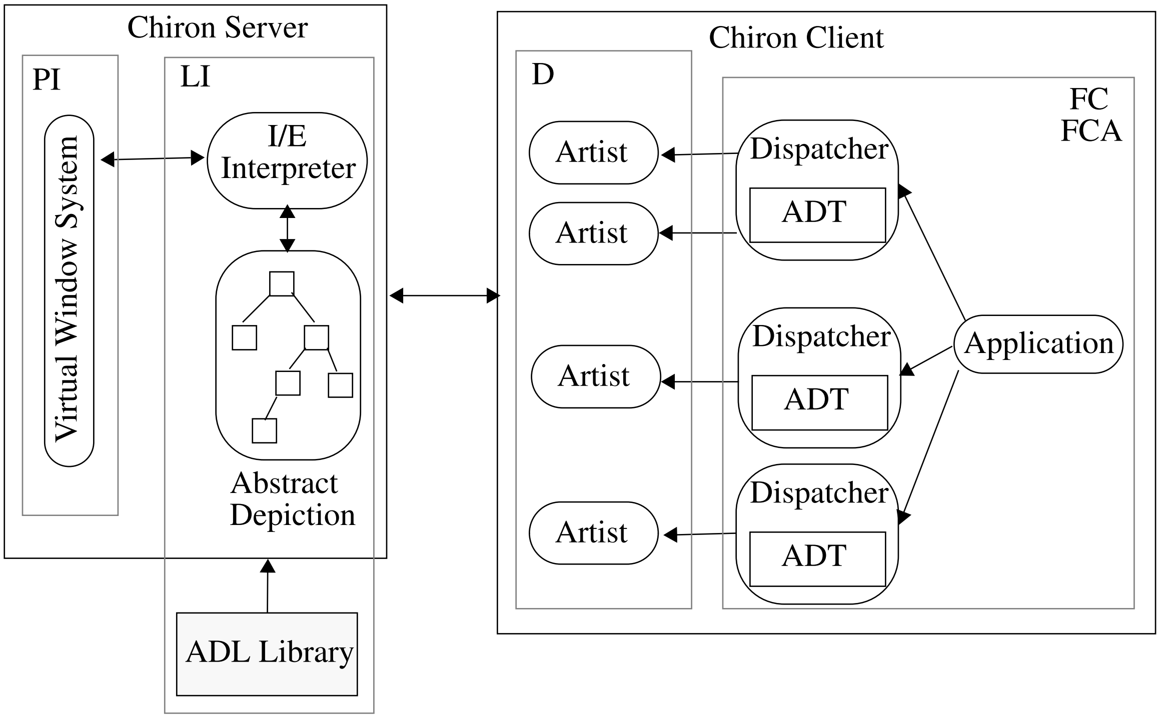

Figure 8: Chiron architecture with functional roles annotated

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.

4.6 Garnet

The emphasis of Garnet's architecture is on control of the runtime behavior of interaction objects and the visual aspects of the interface [15]. This is a different emphasis from the two previous architectures, Serpent and Chiron, which were expressly interested in maintainability and separation of concerns.

Garnet's architecture consists of a single process, with Common Lisp and the X11 window system as its functional foundations. On top of this base the Garnet toolkit has been constructed, and on top of the toolkit are high-level Garnet tools (which can be thought of as applications). Garnet is thus commonly presented as a layered system, as shown in Figure 10.

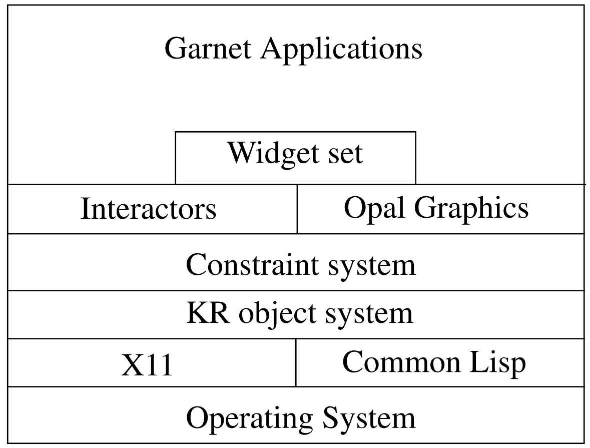

Figure 9: The Garnet architecture (adapted from [15])

However, Garnet is not a strictly layered system. In a strictly layered system, the *n*th layer hides all details of the layers *n-1* and lower, and provides services to layer *n+1*. In Garnet, Common Lisp, the KR object system, and the constraint system are used directly by all layers. The Interactors and Opal Graphics together strictly hide all of the X11 calls. Thus, the widgets and applications do not have any operating system or window manager calls in them, only calls to the interactors, object-oriented graphics, constraints and object system [16].

Given this analysis, we can redraw Garnet's architecture as follows:

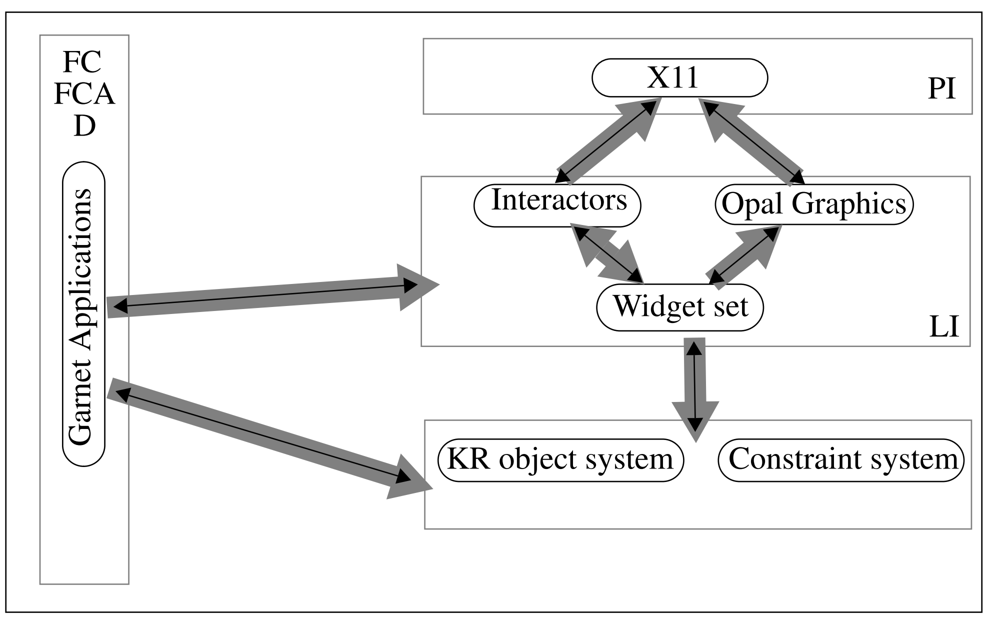

Figure 10: The Garnet architecture with functional roles annotated

The X11 window system is Garnet's physical interaction component. The widget set, interactors and Opal graphics collectively comprise Garnet's logical interaction component, and make use of the object and constraint systems as services. Finally, applications in Garnet are built on top of the logical interaction, object and constraint component "layers", potentially utilizing the services of all of those layers. The arrows in Figure 10 have been deliberately drawn as though they connected the entire layers together, rather than showing separate arrows between each of the elements. This is because Garnet provides the object and constraint systems and logical interaction components as services which are potentially available to *all* layers, rather than as encapsulated structures. By way of comparison, the physical interaction component of Garnet is strictly hidden from the other layers—it can only be accessed through the widget set, interactors and Opal graphics.

Note that we have removed any reference to Common Lisp in our re-characterization of Garnet. This is because Common Lisp is the language of implementation which all components use, rather than a distinct structural entity. Similarly, we have removed reference to the underlying operating system.

This characterization of Garnet illustrates several points: Garnet subdivides dialogue functionality into three distinct parts and logical interaction functionality into three distinct parts. Application functionality, however, is not subdivided, and is not separated, in specification or at runtime, from the rest of the system. Noting where an architecture subdivides its functionality beyond what the domain analysis does indicates the emphases of the architecture. The subdivision is the architectural manifestation of the stated goals of Garnet: control of the runtime behavior of interaction objects (dialogue) and the visual aspects of the interface (physical interaction and logical interaction components).

Also, it is interesting to note that Figure 10 has actually been simplified somewhat: the connections between the FC/FCA/D component and the LI component is shown as a single link. In fact, each of the subcomponents could be linked with any of the other subcomponents: interactors may make use of, for example, the KR object system and the constraint system; Garnet applications may make use of the KR object system, the constraint system, the interactors, widgets or Opal graphics.

5 Analyzing Architectural Qualities

Architectures are not created for abstract purposes. They are typically created in response to a perceived need, or shortcoming with existing software approaches to a problem. With this in mind, it is clear that a method for evaluating architectures must take, as its starting point, a particular quality or set of qualities. Architectures can then be evaluated with respect to how well they support these qualities.

The SAAM (Software Architecture Analysis Method) thus consists of five distinct steps:

1. Characterize the 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.

The first two steps of this architecture were demonstrated in Section 4. We will demonstrate the rest of the SAAM in the coming sections.

5.1 Choosing a Set of Quality Attributes

In order to analyze architectures for a particular quality attribute, it is necessary to understand the implications of that attribute. Once this is done, a generic set of tests, or benchmarks, which characterize that attribute can be designed, and the architecture can be evaluated in terms of this benchmark suite. We will exemplify this process with respect to the attribute of modifiability. This benchmark suite could have been created with respect to other attributes—portability, efficiency, security, and so forth. We have chosen modifiability both because of the previously mentioned emphasis on evolutionary systems and because it is an oft-cited motivation for user interface architectures.

“Modifiability” by itself is an abstract concept. In order to understand how to design for modifiability, it is necessary to better understand the ramifications of this attribute. In order to accomplish this, 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. Then a sample set of modifications can be posited, and each candidate architecture can be evaluated according to how well they support each sample modification. This set of posited modifications becomes a benchmark for all architectures within the application domain. If the application domain is well understood, the benchmark set of modifications can often be given a sample distribution, for the purposes of ranking the individual evaluations within the posited set.

Oskarsson gives an informal characterization of classes of modifications in [19]. 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.

5.2 Choosing a Set of Concrete Tasks

The next step in an analysis of an architecture’s suitability for modifiability is to characterize the particular types of modifications which are likely within the domain. In the user interface domain, two classes of modifications prove most common:

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

The iterative nature of the user interface life cycle is distinguished from typical software engineering life cycles in that it is more based upon empirical testing and validation, and more based upon prototyping. Requirements for the human-computer interaction portion of a system 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. For this reason, the user interface life cycle has often been characterized as a "star" shaped life cycle [12], as shown in Figure 11, in contrast to the typical software engineering life cycle models, which are represented as "waterfalls" or "spirals".

- *Adaptation to new operating environments* is also 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 is the devices, both logical and physical, which the user uses to interact with the program.

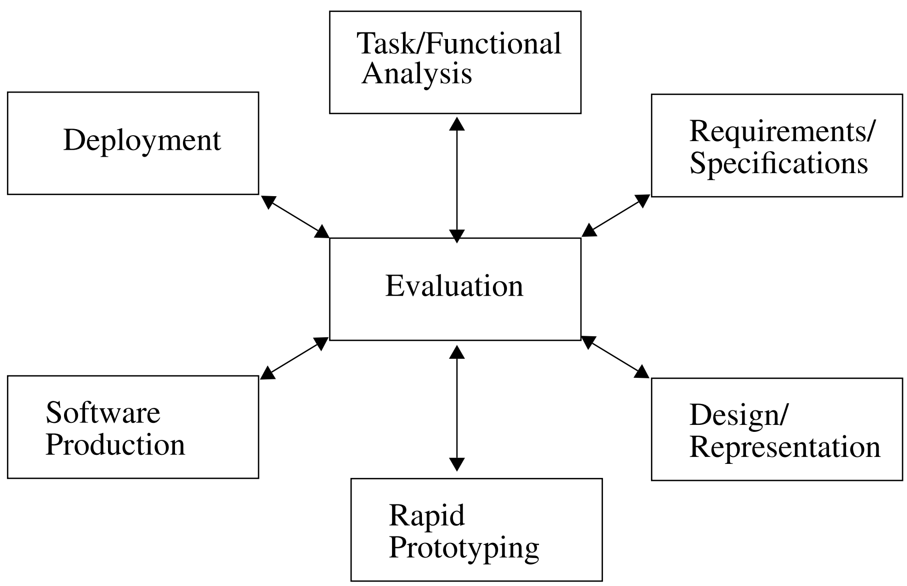

Figure 11: The star life cycle

The other classes of modifications identified in the previous section—restructuring and deletion of unwanted capabilities—while not unknown, do not constitute a significant percentage of the modifications which user interfaces undergo. We will not include them in our set of modifications.

Consequently, with this set of modifications in mind, we can now analyze the ramifications of each type of modification on each architecture. We do this by choosing a set of concrete tasks which test the desired quality attributes.

This process is akin to choosing a set of benchmark tasks through which a piece of software or hardware may be evaluated. These modifications are intended to approximately model the type and distribution of tasks which are typical of ones own software development life cycle. This is why a *set* of example modifications is required.

5.2.1 Changing the Physical Interaction

One type of modification which is common to user interfaces is changing the physical interaction component—typically the toolkit and/or windowing system used. Thus, the first modification exemplar which we examine is a change to the physical interaction component. For example, a move from using Motif to OpenLook or the Macintosh toolkit would be typical of this category of modification. This modification manifests itself as a complete replacement of the physical interaction component. Note that one assumption underlying this sample modification is that all interaction behavior required by the logical interaction component can be met by any of the chosen physical interaction components.

5.2.2 Changing the Dialogue

The next exemplar modification is an extension of capabilities. This class of change is, we assume, the most common (and hence most costly overall) change in the user interface life cycle.

This modification manifests itself as a change within a single functional role, namely the dialogue component. Our example modification is to add a single option to a menu, reflecting some piece of application functionality which must be made available to a user.

6 Analysis of Architectures for User Interfaces

Now that we have enumerated our set of benchmark tasks, we are in a position to evaluate the degree to which each of our candidates provides *architectural* support for these modifications.

6.1 Serpent

6.1.1 Changing the Physical Interaction

Changing the physical interaction component in Serpent assumes that we have an application running under one toolkit and we want to change to another which is supported within the Serpent run-time environment. For example, this situation would occur if we had Motif running in one Serpent PI component and OpenLook in another. What would have to change in this instance is the binding between an application and its physical interaction toolkit. It is not clear from the architecture of Serpent how this binding is achieved. However, it is reasonable to assume that this link is achieved by the dialogue controller component which accesses data in the global shared data structure that was placed there by application and presentation modules. Therefore, changes to the dialogue component would be the only necessary modification to accommodate the switch from one toolkit to another. If the dialogue was written utilizing a sufficiently abstract set of virtual objects, no changes to the dialogue would be necessitated. However, Serpent makes no architectural provision for this: the LI and PI components are combined into a single run-time entity. The architecture *does* ensure that the PI component is isolated from the rest of the system, which minimizes PI dependencies.

6.1.2 Changing the dialogue

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 module of the system. But exactly where the change resides may be more difficult to determine as Serpent dictates no further structure to the dialogue component. Our conclusion is that Serpent's architecture provides no architectural support for the change beyond isolating it to the monolithic structure of its dialogue controller module.[1]

6.2 Chiron

6.2.1 Changing the Physical Interaction

Chiron goes one step beyond Serpent in providing architectural support for changing the PI component. Chiron has isolated both the PI component and the LI component 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 LI and PI components are logically and physically separate, and therefore must communicate via a well-defined interface. The existence of such an interface means that other PI components 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 changing the PI component.

6.2.2 Changing the dialogue

Chiron fares slightly less well with respect to the second modification. As stated in Section 4.5, 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 a change to 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. Chiron does provide support for such a modification in that Artists are associated with individual ADTs, and so there is no monolithic dialogue to modify. However, since it is not clear whether a change will affect an Artist, an ADT, or both, we cannot claim that a strict separation of concerns for this modification exists, and thus we cannot claim that Chiron provides strong architectural support for changing the dialogue.

6.3 Garnet

6.3.1 Changing the Physical Interaction

As can be seen from Figure 10, Garnet provides some architectural support for separating dia-

1. Note that Serpent does provide some language support for this modification, but that is not the topic being considered here.

logue concerns from application and presentation concerns, but that is not its main focus. Garnet is much more highly language-based than any of the other architectures described in this report. As such, many of the functions available in Garnet, such as constraints and the KR object system, are available as integrated language services. In fact, Garnet's main emphasis is in providing integrated dialogue services, rather than on physically isolating components.

Still, Garnet has successfully isolated the PI component (the X11 toolkit in Figure 10) to a single component, and has hidden any X11 dependencies behind the LI component (Opal graphics and Interactors). Garnet has strictly separated the PI component, and so we can claim that changing this component is supported by the architecture and should be relatively easy. In fact, a Macintosh release of Garnet is planned.

6.3.2 Changing the dialogue

The second modification is not supported by Garnet's architecture. A dialogue in Garnet is monolithic, and can involve any of the language features 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. 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.

6.4 PAC

Though we introduced the PAC model as a functional partitioning only, it is wrong for us to ignore some of its structural implications for user interface architectures. PAC is not realized as toolkit for implementing user interfaces (as are Serpent, Chiron and Garnet), but it is worth considering how a notional architecture based on PAC would fare on the benchmark tasks. Though it is clear that PAC did not directly influence such mainstay commercial interface builders such as HyperCard, Visual Basic and NextStep, their similarities with regard to distributed dialogue make a tempting architectural comparison. Also, Hill's Abstraction-Link-View paradigm [10], which underlies the Rendezvous [11] architecture for user interface development, is strikingly similar to PAC.

6.4.1 Changing the Physical Interaction

When we look at our sample modifications from PAC's perspective, we can see that changing the presentation (the physical interaction component) is not a simple process. The physical interaction component is distributed among the leaf agents in the PAC hierarchy. Substituting for the entire physical interaction component would mean that all of these individual agents would have to be modified. Assuming that the implementation of the system respects the modularization of the PAC design, none of the individual changes would be difficult, but it would require many individual changes, which is undesirable.

6.4.2 Changing the dialogue

Changing the dialogue in a PAC model is very different from previous architectures. The PAC hierarchy is composed of a potentially large number of dialogue components. Interaction widgets like a menu would certainly be considered independent PAC agents and so the location of the modification would be much easier to isolate and fix.

Furthermore, the decomposition of the dialogue into a hierarchy of cooperating agents means that individual agents have a regular structure, a well-specified relationship with the rest of the PAC hierarchy, and are self-contained. These are just the qualities that support modifiability: ease of locating the affected area, ease of understanding the proposed change, and separation of concerns (so that a modification needs to be made in only one location, and affects no other location).

Facilitating the decomposition of the dialogue is, in fact, the main motivation for PAC. Thus, we can confidently claim that PAC's architecture strongly supports changing the dialogue.

7 Conclusions and Future Work

We have provided a method for evaluating architectures, based upon two main ideas: a common understanding and representation for architectures, and an analysis of an organization's life cycle requirements. This method permits the comparison and ranking 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 both its needs in terms of the attributes being evaluated and then to choose a representative selection of tasks to demonstrate those needs. This is in keeping with the purpose of the SAAM which is not to criticize or commend particular architectures, but to provide a method for determining which architecture support's an individual's or organization's life cycle needs. This methodology will work for other attributes, and will most likely provide different rankings of architectures for these attributes. The rankings with respect to an attribute can be seen as the degree to which an architecture was *designed* to support that attribute.

As our understanding of this topic is, as yet, insufficient to allow for metrics by which we might more precisely evaluate attributes in term of architectures, we simply provide ways to analyze, compare and rank the architectures themselves. We believe that the work of Henry and Kafura [9] 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.

We must admit at this point that the analysis of user interface software is not completely satisfactory. Our analysis hinged solely on the allocation of function as described by the Arch/Slinky model to the structure of the various tools. Hence, our judgement from an architectural perspective depended on whether a tool's structure localized some logical function in the domain. But localization is not the only concern when considering modifiability. As we already mentioned, a tool may provide language support for further structuring within a logical component (e.g., the use of Slang in Serpent to specify dialogue behavior in the dialogue controller). Furthermore, we have not taken into the knowledge that structural components have of each other, even though the structural diagrams show data and control relationships. Clearly, our analysis of modifiability is at best incomplete.

We must also note an apparent tautology in our arguments. We presented the canonical domain partitioning for user interface software and then expressed benchmark modifications in terms of that partitioning. It is not surprising that tools whose structure mirrored the canonical partitioning fare better with respect to modifiability. This suggests that domain partitioning is dependent on software quality as well. If we were to extend our analysis to other qualities, we might very need a different canonical partitioning. In fact, this might explain the difference between the Arch/Slinky model and PAC—they were suggested with different software qualities in mind. Arch/Slinky is clearly motivated by separation between its five logical components. PAC suggests that separation between application and presentation is secondary to the separation of user-specific conceptual objects which comprise the individual PAC agents.

8 Acknowledgments

This research was conducted as part of the Evolutionary Development Life Cycle (EDLC) project led by Dick Martin of the School of Computer Science at CMU. We would very much like to thank Brian Boesch and Steve Cross of ARPA for their support of our research. We would also like to acknowledge the intellectual input of various colleagues at CMU from the Software Engineering Institute and the Computer Science Department, especially Larry Howard and Reed Little whose thoughts on software architecture have greatly influenced our own thinking and writing. Finally, we thank Brad Myers, Laurence Nigay and Dick Taylor for their help in understanding Garnet, PAC and Chiron, respectively.

9 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”, in *SIGSOFT ‘93 Symposium on the Foundations of Software Engineering*, ACM Press, December, 1993. Forthcoming.

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

[5] Euler, L., Maffei, E. Rauch, A. “Create Real Windows Applications in a Graphical Environment Using Microsoft Visual Basic”, *Microsoft Systems Journal*, July 1991, 57-70, 116.

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

[7] Goodman, D. *The Complete HyperCard Handbook*. New York: Bantam Books, 1987.

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

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

[10] Hill, R. “The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications”, *Proceedings of CHI ‘92,* Monterey, CA, May 1992, 335-342.

[11] Hill, R., Brinck, T., Patterson, J., Rohall, S., Wilner, W. “The Rendezvous language and architecture”, *Communications of the ACM*, 36(1):62-7, January, 1993.

[12] Hix, D., Hartson, H. “Formative Evaluation: Ensuring Usability in User Interfaces”, in Bass, L., Dewan, P. (eds.) *User Interface Software*, Chichester: John Wiley & Sons, 1993.

[13] Lane, T. “Studying software architecture through design spaces and rules”, Software Engineering Institute, Carnegie Mellon University Technical Report CMU/SEI-90-TR-18, Pittsburgh, PA, 1990.

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

[15] Myers, B., Giuse, D., Dannenberg, R., Vander Zanden, B., Kosbie, D., Pervin, E., Mickish, A., Marchal, P. “Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces”, IEEE Computer, 23(11): 71-85.

[16] Myers, B. personal communication, August 1993.

[17] Nigay, L., Coutaz, J. “Building user interfaces: Organizing software agents”, ESPRIT ‘91 Conference, Brussels, November 1991.

[18] Nigay, L. personal communication, June 1993.

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

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

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

[22] 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.

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

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

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

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

[27] Sukaviriya, P., Foley, J., Griffith, T. “A Second Generation User Interface Design Environment: The Model and The Runtime Architecture,” *Proceedings of InterCHI ‘93,* Amsterdam, May 1993, 375-382.

[28] 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.

[29] 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.

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

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

[32] Wielemaker, J., Anjewierden, A. "Separating User Interface and Functionality Using a Frame Based Data Model", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology,* Williamsburg, VA, November 1989, 25-33.

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