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

Research Paper

Richard N. Taylor Nenad Medvidovic Kenneth M. Anderson
E. James Whitehead Jr. Jason E. Robbins

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717-3425
{taylor,neno,kanderso,ejw,jrobbins}@ics.uci.edu

Abstract

While a large fraction of application system code is devoted to user interface (UI) functions, support for reuse in this domain has largely been confined to creation of UI toolkits ("widgets"). We present a novel architectural style directed at supporting larger grain reuse and flexible system composition. Moreover, the style supports design of distributed, concurrent, applications. A key aspect of the style is that components are not built with any dependencies on what typically would be considered lower-level components, such as user interface toolkits. Indeed, all components are oblivious to the existence of any components to which notification messages are sent. Asynchronous notification messages and asynchronous request messages are the sole basis for inter-component communication. While our focus has been on applications involving graphical user interfaces, the style has the potential for broader applicability. Several trial applications using the style are described[1].

1.0 Introduction

Software architectural styles are key design idioms [PW92][GS93]. Unix's pipe-and-filter style is more than twenty years old; blackboard architectures have long been common in AI applications. User interface software has typically made use of two primary run-time architectures: the client-server style (as exemplified by X windows) and the call-back model, a control model in which application functions are invoked under the control of the user interface. Also well known is the model-view-controller (MVC) style [KP88], which is commonly exploited in Smalltalk applications. The Arch style is more recent, and has an associated meta-model [Wor92].

This paper presents a new architectural style. It is designed to support the particular needs of applications that have a graphical user interface aspect, but the style clearly has the potential for supporting other types of applications. This style draws its key ideas from many sources, including the styles mentioned above, as well as specific experience with the Chiron-1 user interface system [TJ93]. In the following exposition, the style is referred to as the Chiron-2, or C2, style.

A key motivating factor behind development of the C2 style is the emerging need, in the user interface world, for a more component-based development economy. User interface software frequently accounts for a very large fraction of application software, yet reuse in the UI domain is typically limited to toolkit (widget) code. The architectural style presented supports a paradigm in which UI components, such as dialogs, structured graphics models (of various levels of abstraction), and constraint managers, can more readily be reused. A variety of other goals are potentially supported as well. These goals include the ability to compose systems in which: components may be written in different programming languages, components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple users may be interacting with the system, multiple toolkits may be employed, multiple dialogs may be active (and described in different formalisms), multiple media types may be involved, and multiple user tasks ("pro-

1. This material is based upon work sponsored by the Air Force Materiel Command, Rome Laboratory and the Advanced Research Projects Agency under contract number F30602-94-C-0218. The content of the information does not necessarily reflect the position or policy of the Government and no official endorsement should be inferred.

cesses”) supported. We have not yet demonstrated that all these goals are achievable or especially supported by this style. We have examined several key properties and built several diverse experimental systems, however. The focus of this paper, therefore, is to present the style and describe the evidence we have to date. We believe our preliminary findings are encouraging and that the style has substantial utility “as is.” Further studies will examine the degree to which the style supports attainment of the various goals.

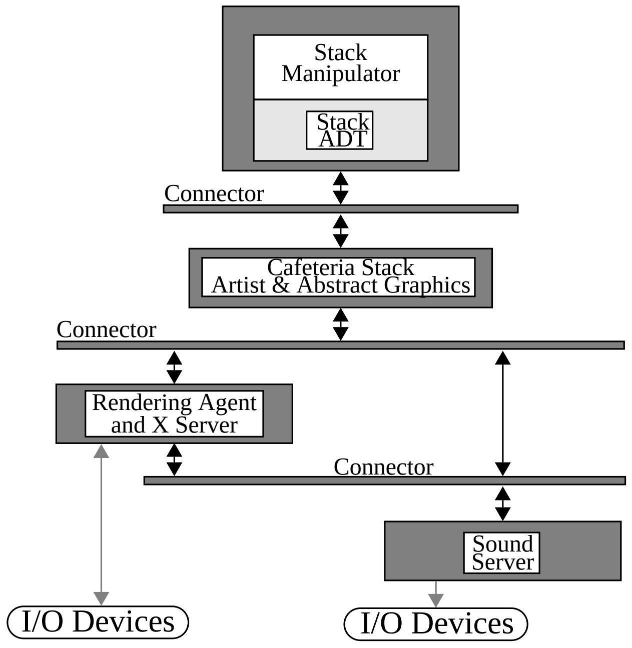
The new style can be informally summarized as a network of concurrent components hooked together by message routing devices. Central to the architectural style is a principle of limited visibility: a component within the hierarchy can only be aware of components “above” it, i.e., components typically closer to the “application,” and thus further from, e.g., the windowing system[2]. Components are totally unaware of the components—including toolkits—which reside “beneath” them. All components have their own thread(s) of control and there is no assumption of a shared address space. It is also important to recognize that this conceptual architecture is distinct from the implementation architecture. There are many ways of realizing a given conceptual architecture, and this topic will be briefly discussed later.

A small example serves to illustrate several of these points. In Figure 1, we diagram a system in which a program alternately pushes and pops items from a stack; the system also displays the stack graphically, using the visual metaphor of a stack of plates in a cafeteria. The human user can “directly” manipulate the stack by dragging elements to and from it, using a mouse. As the user drags elements around on the display, a scraping sound is played. Whenever the stack is pushed, a sound of a spring being compressed is played; whenever the stack is popped, the sound of a plate breaking is played.

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

Notions of above and below are used in this paper to support an intuitive understanding of the architectural style. In this discussion the application code is (arbitrarily) regarded as being at the top while user interface toolkits, windowing systems, and physical devices are at the bottom. (The human user is thus at the very bottom, interacting with the physical devices of speaker, keyboard, mouse, microphone, and so forth.) While this vertical orientation may be helpful in developing understanding, it should be noted that the precise uses of top and bottom, provided below, do not rest on assumption of this particular vertical orientation.

FIGURE 1. An audio-visual stack manipulation system.



Visual depiction of the stack is performed by the “artist” that receives notification of operations on the stack and creates an internal abstract graphics model of the depiction. The rendering agent monitors manipulation of this model and ultimately creates the pictures on the workstation screen. To produce the audio effects, the sound server at the bottom of the hierarchy monitors the notifications sent from the artist and the graphics server; depending on the events detected, the various sounds are played. Performance is such that playing of the sound is very closely associated with mouse movement; there is no perceptible lag. The artist and rendering agent are completely unaware of the activities of the sound server; similarly, the stack manipulator is completely unaware that its stack object is being visualized.

The paper is organized as follows. Section 2.0 presents the new architectural style. Section 3.0 presents a set of sample applications that have been built to investigate various aspects of the style. Section 4.0 discusses related work. Discussion of a large list of open issues and a conclusion round out the paper.

2.0 A UI architectural style supporting heterogeneity, concurrency, and composition

In this section we present the *architectural style* and its rationale. Key elements of the C2 architectural style are components and connectors. A configuration of a system of connectors and components is an *architecture*. There is also a set of principles governing how the components and connectors may be legally composed. We attempt to provide a rationale for the desired properties of the components and connectors, as well as the choice of principles. Due to space constraints, our presentation is largely informal.

The architectural style consists of *components* and *connectors*. Components and connectors both have a defined top and bottom. The top of a component may be connected to the bottom of a single connector. The bottom of a component may be connected to the top of a single connector. There is no bound on the number of components or connectors that may be attached to a single connector. Components can only communicate via connectors; direct communication is disallowed. When two connectors are attached to each other, it must be from the bottom of one to the top of the other. Both components and connectors have semantically rich interfaces.

Components communicate by passing *messages*: *notifications* travel down an architecture and *requests* up. Connectors are responsible for the routing and potential multicast of the messages.

2.1 Components

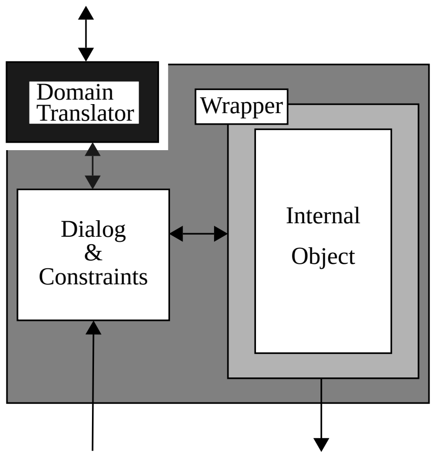
Components have state, their own thread(s) of control, and a top and bottom domain. The top domain specifies the set of notifications to which this component responds, and the set of requests that this component emits up an architecture. The bottom domain specifies the set of notifications that this component emits down an architecture and the set of requests to which it responds. (The elements of a bottom domain's sets are closely related, as will be discussed later. The two sets comprising the top domain do not necessarily have any relation.)

For purposes of exposition below, a specific internal architecture, targeted at the user interface software domain, is assumed. (It will be clear from the ensuing discussion that issues concerning composition of an architecture are independent of a component's internal structure.) Components contain an object with a defined interface, a wrapper around the object, and a dialog and constraint maintenance program, as shown in Figure 2. The object can be arbitrarily complex. (For example, one component's object might be a complete structured graphics model of the contents of a window.) An object's wrapper provides the following service: whenever one of the access routines of the object's interface is invoked, the wrapper reifies that invocation and its return values as a notification (in the component's bottom domain) and sends the notification to the connector below the component[3]. Thus the types of notifications emitted from a component are determined by the interface to its internal object.

A domain translator subcomponent may also be present, to assist in mapping between the component's internal semantic domain and that of the connector above it.

3. Components can alternatively be formulated such that the wrapper sends the connector the state, or part of the state, of the internal object. This variation is discussed briefly in Section 5.0.

FIGURE 2. The Internal Architecture of a C2 Component.



The access routines of the object may be invoked (only) by the dialog portion of a component. This code, which has its own thread of control, may act upon the object for any reason, but the intended style includes three situations: a) in reaction to a notification that it receives from the connector above it, b) to execute a request received from the connector below it, and c) to maintain some constraint, as defined in the dialog.

For case (a), the dialog receives a notification in its top domain and determines what, if anything, to do as a result of receiving the notification.

In case (b), the component receives a request in its bottom domain and determines what, if anything, to do with the request. For instance, it could choose to delay processing of the request, ignore it, perform it without any additional processing, or perhaps perform some other action.

Case (c) is best understood by considering its user interface purpose: constraint managers are commonly employed in GUI applications to resize fields, planarize graphs, or otherwise keep parts of objects in some defined juxtaposition. The constraint portion of a component can play this role either as part of case (a) or (b), or the constraint manager may autonomously manipulate the component's object.

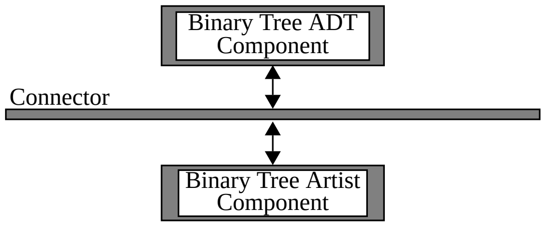
The dialog portion of a component may, in addition, choose to send a request to the connector above it. These requests should be phrased in terms of a function of the connector's top_in domain, a property which derives from the definitions in Section 2.4.

2.2 Notifications and Requests

Components in an architecture communicate asynchronously via messages. There are two types of messages: notifications and requests. A notification is sent downward through a C2 architecture while a request is sent up. Notifications are announcements of state changes of the internal object of a component. As noted above, the types of notifications that a component can emit are fully determined by the interface to the component's internal object.

For instance, consider a small system consisting of two components connected by one connector, as shown in Figure 3. One component manages a binary tree abstract data type (ADT) while the other component manages a depiction of that binary tree[4]. An example notification from the ADT component is "new key has been inserted". This notification is generated automatically by the wrapper that monitors the usage of the component's internal object. The Artist component receives the notification and makes calls to its internal object to update the depiction.

FIGURE 3. A partial C2 architecture.



Requests, on the other hand, are directives from components below, generated by their dialog, requesting that an action be performed by some set of components above. The requests that a component can receive are determined by the interface to the component's internal object, similar to the way that notifications are determined. The difference is that a notification is a statement of what interface routine was invoked and what its parameters and return values were, whereas a request is a statement of a desired invocation of one of the object's access functions.

To continue the example, the user may select a node of the binary tree depiction, managed by the Artist, indicating that the node should be removed from the tree. A request to remove the key associated with the selected node is generated by the Artist and sent by the connector to the ADT. The Binary Tree Component removes the key from its internal object to satisfy the request. This, in turn, generates a notification down the architecture, stating that the key has been deleted, causing the Artist to update its depiction.

Note that many potential C2 components, such as commercial user interface toolkits, have interface conventions that do not match up with C2's notifications and requests. Typically these systems will generate events of the form "this window has been selected" or "the user has typed the letter 'a'" and send them *up* an architecture. These toolkit events will need to be caught by C2 bindings to the toolkits (i.e., adaptors) and converted into C2 request messages. Conversely, notifications from a C2 architecture will have to be converted to the type of invocations that a toolkit expects. In order for these translations to occur and be meaningful, careful thought has to go into the design of the internal objects of the bindings to the toolkits such that they contain the required functionality and are reusable across architectures. (This is not an unreasonable task: we have already accomplished this for both Motif and OpenLook.)

2.3 Connectors

Connectors bind components together into a C2 architecture. They may be connected to any number of components as well as other connectors. A connector's primary responsibility is the routing and broadcast of messages. A secondary responsibility is message filtering.

2.3.1 Broadcast Policies

Connectors may provide a number of filtering and broadcast policies for messages, such as the following.

- No Filtering: Each message is sent to all connected components on the relevant side of the connector (bottom for notifications, top for requests).
- Notification Filtering: Each notification is sent to only those components that registered for it.
- Conditional: The connector defines an ordering (i.e. priority ranking) over its connected components and sends a notification to each component in order until some condition has been met. This is useful for cases in which components connected to one side of the connector all perform the same function (with possibly different implementations) and a destination is computed based on current conditions.

A connector has an upper and lower domain, defined by the components and connectors[5] attached to it. These are described in the following section.

2.4 Architecture Composition and Properties

An architecture consists of a specific configuration of components and connectors. The meaningfulness of an architecture is a function of the connections made. This section formalizes several key relationships. In addition to aiding precise exposition, the formalizations are the basis for automated analyses of candidate architectures by a design environment [RWMT95].

Let *bottom_in* be the set of requests received at the bottom side of a component or connector. Let *bottom_out* be the set of notifications that a component or connector emits from its bottom side. Furthermore, let *top_in* be the set of notifications received on the top side of a component or connector, and let *top_out* be the set of requests that they send from their top sides.

4. For purposes of this discussion, the external applications using the binary tree, as well as the other components and connectors needed to actually display the depiction are elided.

5. For the purposes of the discussion below, we do not make a distinction between components attached to a connector and a connector attached to a connector.

Figure 4 represents the external view of a component C_i. $C_i.top\_out$ and $C_i.top\_in$ are defined by the component's dialog: they are the requests it will be submitting and notifications it will be handling. $C_i.bottom\_out$ are the notifications the component will be making, reflecting changes to its internal object. $C_i.bottom\_in$ are the requests the component accepts. In keeping with the discussion of Section 2.2, the types of those requests can be defined as a function, *N_to_R*, of the notifications

$$C_i.bottom\_in = N\_to\_R(C_i.bottom\_out)$$

This function is 1-to-1 and onto; it has an inverse function, *R_to_N*, that will uniquely map the requests to notifications.

FIGURE 4. C2 Component Domains

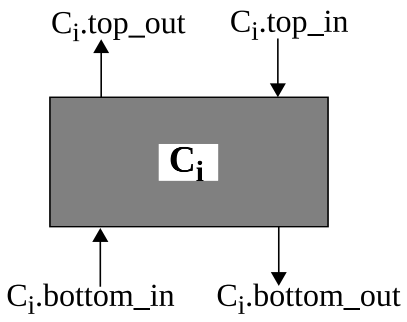
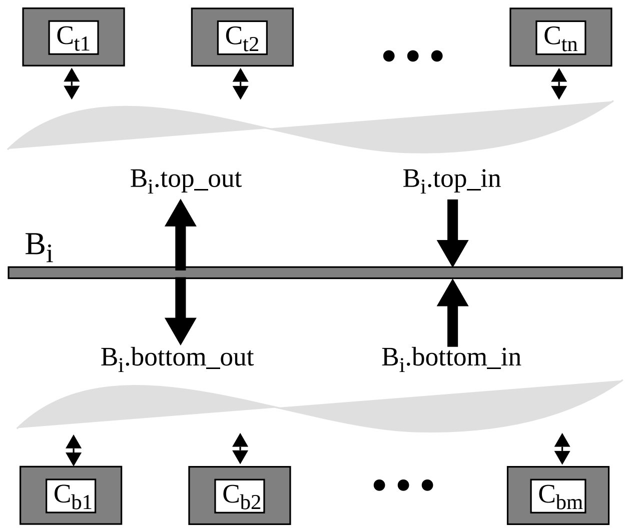


Figure 5 represents the external view of a connector B_i, with the components C_{tj} and C_{bk} attached to its top and bottom respectively. A connector's upper and lower domains are completely specified in terms of these components.

FIGURE 5. C2 Connector Domains



Consider the notifications that come in from the components C_{ti} above the connector:

$$B_i.top\_in = \bigcup_j C_{tj}.bottom\_out$$

Then, since connectors may have the ability to filter messages, as discussed in Section 2.3.1, the notifications that come out of the bottom of a connector are a subset of the notifications that come in from above. Thus, for each connector B_i, it is possible to identify the function *Filter_TB*, such that

$$B_i.bottom\_out = Filter\_TB(B_i.top\_in)$$

Similarly, consider the requests that come in from the components C_{bk} below the connector:

$$B_i.bottom\_in = \bigcup_j C_{bk}.top\_out$$

Finally, if it is also possible for a connector to filter requests, the requests that come out of the top of a connector are a subset of those that come in from below, so that the function *Filter_BT* is defined as follows

$$B_i.top\_out = Filter\_BT(B_i.bottom\_in)$$

In summary, a connector's domain is defined by the unions of the domains of the components above and below it, along with any filtering that the connector does to those domains.

Pairwise relationships can be specified between the domains of any connector and the component attached to it. These relationships are expressed in terms of the potential for communication between them.

A connector B_i and the j-th component above it, C_{tj}, are considered *fully communicating* if every request the connector sends up to the component is "understood."

$$Full\text{-}Comm(B_i, C_{tj}) \equiv$$
$$B_i.top\_out_j \subseteq C_{tj}.bottom\_in$$

B_i and C_{tj} are *partially communicating* if the component understands some, but not all of the requests the connector sends.

$$Partial\text{-}Comm(B_i, C_{tj}) \equiv$$
$$(B_i.top\_out_j \cap C_{tj}.bottom\_in \neq \varnothing) \wedge$$
$$(B_i.top\_out_j \cap C_{tj}.bottom\_in \subset B_i.top\_out_j)$$

Finally, they are *not communicating* as follows:

$$No\text{-}Comm(B_i, C_{tj}) \equiv$$
$$B_i.top\_out_j \cap C_{tj}.bottom\_in = \varnothing$$

The relationship between a connector B_i and a component C_{bk} below it can be defined in a similar manner, by substituting 'bottom_out' for 'top_out' and 'top_in' for 'bottom_in' in the above equations.

The degree of utilization of a component's services, i.e., the relationship between a component and a connector from the perspective of the requests and notifications the component *receives* from the connector can be defined through a simple substitution of terms in the three equations above. For instance, if B_i.top_out is a non-empty proper subset of C_{tj}.bottom_in, then C_{tj} is being *partially utilized*.

Finally, by utilizing the specified functions and relationships, it is possible to express a number of other relationships in a given configuration (e.g., B_i*.bottom_out* can be expressed as a function of C_{tj}*.bottom_in*).

These definitions enable us to answer such questions as whether a component can be added to an existing architecture without modifications, whether its requests will be handled, if it will be able to process the notifications it will be receiving, etc. Conceivably such analyses could be performed either statically or dynamically. We are currently focusing only on analyses performed statically, on a model of an architecture, by a system development environment.

The C2 design environment is also intended to provide support for domain translation. Substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. One issue that must be addressed, however, is the potential dependence of a given component on its "superstrate," i.e., the components above it. If each component is built so that its top domain closely corresponds to the bottom domains of those components with which it is specifically intended to interact in a given architecture, its reusability value is greatly diminished. For that reason, the C2 style introduces the notion of request translation. For each component, a mapping from generic requests to the specific interfaces of those the connector above it can be produced and encapsulated in the domain translator sub-component. Producing this mapping will take place in the development environment by the system architect.

2.5 Design and Development Environment

The specific architecture formed when components are connected plays as large a role in determining the behavior of the overall system as the internal logic of the components. For that reason, development tools that operate on architectural specifications are as important as tools that work on individual components. This section describes the goals of a design environment [FGNR92] for building C2-style architectures. A prototype of such an environment has been built (See Section 3.3), and work is continuing [RWMT95].

The C2 design environment will be an editor in which designers can construct a model of a software system, have that model checked for syntactic and semantic correctness, receive some domain-specific feedback about various design qualities, keep track of unfinished steps in the design process, and view example architectures. Because diagrams are so effective for describing software architectures, the C2 design environment will use a graphical front end to a precise internal representation.

Given additional implementation details as annotations on the internal abstract architectural description, a design environment also holds the promise of automatically generating some parts of the final system. Examples of implementation details include grouping of components into operating system processes and location of source files that implement individual components. Examples of generated parts include makefiles, process invocation scripts, and routing tables in cases where connectors are implemented with static routing. When implementation details are incomplete, the design environment may be able to generate some parts by use of defaults, hints, or rules.

2.6 Principles of the C2 Architectural Style

The architectural style is characterized by several principles, the collection of which distinguish it from other UI architectures (subsets of the principles, of course, characterize a variety of other systems). A few of these are described below.

- *Substrate Independence* - a component is not aware of the components below it. In particular, the notification of a change in a component's internal object is entirely transparent to its dialog. Instead, the wrapper does this automatically when the dialog accesses the internal object. However, even the wrapper only generates a message, not knowing whether anyone will receive it and respond. Substrate independence fosters substitutability and reusability of components across architectures.
- *Message-based Communication* - all communication between components is solely achieved by exchanging messages. This requirement is suggested by the asynchronous nature of applications that have a GUI aspect, where both users and the application perform actions concurrently and at arbitrary times and where various components in the architecture must be notified of those actions. Message-based communication is extensively used in distributed environments for which this architectural style is suited.
- *Multi-thread* - this property is also suggested by the asynchronous nature of tasks in the GUI domain. It simplifies modeling and programming of multi-user and concurrent applications and enables exploitation of distributed platforms.
- *No assumption of shared address space* - any premise of a shared address space in an architectural style that allows composition of heterogeneous components, developed in different languages, with their own threads of control, internal objects, and domains of discourse, would be unreasonable.
- *Implementation separate from Architecture* - many potential performance issues can be remedied by separating the design architecture from actual implementation techniques. For example, while the C2 style

disallows any assumptions of shared threads of control and address spaces in a design architecture, substantial performance gains may be made in the actual implementation by placing multiple components in a single process and a single address space where appropriate. Furthermore, modelling the exchange of messages among components by procedure calls where appropriate could yield performance gains.

3.0 Examples and Trial Applications

To formulate a viable new architectural style is a major undertaking. A variety of experiments and proof-of-concept exercises are needed just to assess initial plausibility of the key ideas. As we are especially concerned with user interface applications, and since performance is a critical factor in assessing the viability of any technology in this domain, we have conducted a variety of research prototypes. The trial applications described below, as well as the example of Section 1.0, are results of projects completed by students in a graduate-level course on user interfaces at the University of California, Irvine (UCI). These trial applications were designed as small-scale experiments to examine one or more aspects of the C2 style. We present a selection of these prototypes here, focusing on those which examined the style's visibility rules and multi-component nature. Space constraints prohibit discussion of a multi-user distributed appointment system. With the exception of the modeling workbench (Section 3.3), these applications were implemented using the Chiron-1 user interface system [TJ93].

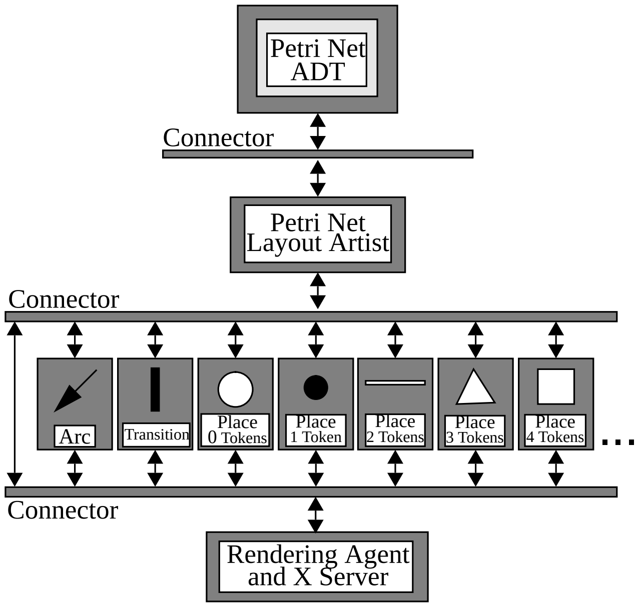
3.1 A Petri Net Tool with Multiple Place Symbols

This example consisted of building a Petri Net editor and simulator such that places in the net are depicted by polygons whose number of sides equals the number of tokens inside each place. Clearly, places with zero, one, or two tokens cannot be represented by polygons. For the purpose of this exercise, they were depicted by an empty circle, a point (dot), and a line respectively. Every time a transition is fired, the shapes of all the places connected to that transition will change.

In order to achieve this, an existing Chiron-1 Petri Net artist was redesigned to fit the C2 architectural style by separating the layout of the Petri Net from its presentation. In addition, the presentation of places with different numbers of tokens was entrusted to separate components. The resulting architecture is shown in Figure 6. The Petri Net Layout Artist maintains the coordinates of places, transitions, and arcs, addresses issues of adjacency, and maintains logical associations with ADT objects. At the same time, it has no knowledge of the artists in the presentation layer or the actual look of the Petri Net.

The project illustrates the substrate independence principle, as well as the multi-level and multi-component nature of the style. The separation of the presentation from the layout enabled the designers to easily change the presentation of Petri Net places from the standard circle-with-dots-as-tokens to polygons. The components in the presentation layer are simple and entirely independent of each other. They can be added, interchanged, or substituted with new ones, without affecting the rest of the system.

FIGURE 6. **Petri Net places are polygons whose number of sides equals the number of tokens**



3.2 Graph Editor with Constraints

This exercise focused on a simple boxes-and-arrows editor, where arrows are constrained to begin and end on edges of certain boxes. As boxes are moved, the arrows are updated accordingly.

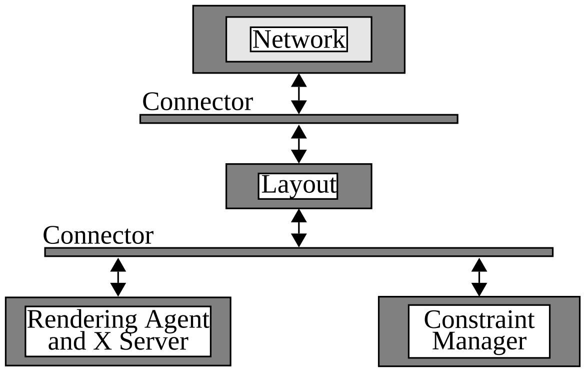
The architecture is shown in Figure 7. The Network component maintains a graph of nodes with their incoming and outgoing links. The Layout block defines the geometry of various types of nodes and maintains their display coordinates and associations with Network objects. The Constraint Manager generates and maintains constraints: for each link between two nodes in a graph, the Manager builds a set of linear constraints based on the geometry of the nodes. The Manager receives the same notifications as the Graphics server. These notifications are processed, constraints applied, and requests sent back to the Layout artist, so that positions may be updated[6].

This example highlights one goal of our work: to be able to include a complex constraint manager in a UI architecture, where its inclusion had not been previously

6. Note that other topologies are probably preferable for this application, namely placing the constraint manager above the rendering agent/window manager. The purpose of this exercise was to examine feasibility issues and was done by evolving a legacy system.

planned. Most constraint managers described in the UI literature are large and often intertwined with the rest of the system [Mye89] This exercise demonstrated C2's ability to incorporate such a manager in a very clean way. While performance may become an issue in such a configuration, no slowdown was noticed in this simple trial.

FIGURE 7. A constraint manager is added as a server to a drawing editor system



3.3 A Workbench for Experimenting with Multi-Level Software Architectures

This exercise focused on building a modeling workbench in which various issues related to C2 could be explored. A model was built of a simple stack application. That model was embellished with various design features in order to explore trade-offs such as the choice between broadcasting (an abstraction of) the state of a component when it changes or broadcasting notifications as described above. Also explored was the usefulness of allowing the domain of a component to vary over time, as a function of its current state. The concept of a domain translator was explored, both for its own usefulness and in combination with run-time domain representation.

The modeling was done programmatically in Self [US91]. Self allowed convenient what-if analysis via both programmatic changes and direct manipulation of the model in Self's graphical inspector. Components were modeled as Self objects. Paths between connectors and components were modeled as pointers. Messages were modeled as Self message sends. Connectors were modeled as Self objects that responded to messages by resending them to all appropriate components. Domains were modeled as Self objects containing a list of available operations. In addition to various insights which resulted and which are reflected in this paper, the need for a modeling and design environment to aid the system designer to visualize and manipulate C2 style architectures was clearly highlighted.

4.0 Related Work

The C2 work draws from the work of many other researchers and systems. We highlight a few of them here, discussing them in a framework of fundamental concepts which influenced the C2 architectural style.

4.1 Implicit Invocation

In the C2 style, implicit invocation occurs when a component reacts to a notification sent down an architecture by invoking some code. The invocation is implicit, because the component which initially issued the notification did not know if the notification would cause any reaction, and the notification certainly did not explicitly name an entry point into a component below it. An excellent discussion of the benefits of implicit invocation can be found in [SN92], as embodied in their mediators concept. The Chiron-1 system [TJ93], through its transmission of abstract data type modification events to potentially reactive artists, also supports implicit invocation. This is similar to VisualWorks [Inc94], a Smalltalk GUI library based on the Model-View-Controller paradigm [KP88], where the model broadcasts change of state notifications to views and controllers. While many systems employ implicit invocation for its benefits in separating modules, the C2 style extends this by providing a discipline for ordering components which use implicit invocation, yielding substrate independence.

4.2 Messages and Message Mechanisms

Message mechanisms in existing systems transmit either service requests, events (notifications), objects, or a combination. Existing systems are distinguished by any discipline imposed on message use and by consequential usage styles. Both Chiron-1 and X windows [SG86] use service request and notification messages. However, their use of notification messages varies: Chiron-1 notifications come from either the application or the graphics server, yielding a separation of concerns between application and depiction, a feat X cannot duplicate. The Field [Rei90] and SoftBench [Cag90] systems also use service request and notification messages. However, unlike Chiron-1 and X, messages in these two systems have no discipline on their use; the two message types are indistinguishable.

In the Weaves system [GR91], concurrently executing tool fragments communicate by passing (pointers to) objects. This passing of objects causes Weaves to be used in a data flow manner. Weave systems do not, to our knowledge, involve data moving both forwards and backwards in a weave. Additionally, there is no notion of reifying as messages (or objects) service invocations upon an internal abstract object.

Experience from the Chiron-1 system indicates that if message traffic occurs across a process boundary in a non-shared address space, then inter-process communications (IPC) becomes a key performance determinant. Experience with the Avoca system [BO94] provides confirmation. These observations motivate a key goal of the C2 style: to provide a discipline for using service request and notification messages which can be mapped to either inter- or intra-process message mechanisms as needed.

4.3 Layered Systems

Concentrating solely on the layering in their architecture, existing approaches span a wide range. Both Field and SoftBench have only a single layer, while the client/server spilt of X supports two. The Chiron-1 system has three layers, the application, artists, and graphics server. The Arch Model (an extension of the Seeheim [Pfa83] model) and the Slinky User Interface MetaModel [Wor92] partition the work of supporting user interfaces into five layers, known as the domain-specific (i.e., “application”) component, domain adaptor, dialogue, presentation, and interaction toolkit components. (The dialog component may be further subdivided to arbitrary levels [Cou91].)

In contrast to these existing systems, the C2 architectural style does not assume that a certain number of layers is “magic” and allows layering to vary naturally with application domain. In this, the C2 style is similar to the composable, parameterized components of the GenAvoca style [BO94], which may also be layered naturally to handle each specific domain. Furthermore, C2 provides a layering mechanism based on implicit invocation, rather than the explicit calls of the GenAvoca style. This allows the C2 style to provide greater flexibility in achieving substrate independence in an environment of dynamic, multi-lingual components. In particular component recompilation and relinking can be avoided and on-the-fly component replacement enabled through use of the message mechanisms.

4.4 Language and Process Support

Many existing systems can support multiple languages, though they are often skewed heavily towards a single language and process subdivision. For example, while there are now many different language bindings for the X system, it still remains the case that C (and C++) is the preferred language for X development. In the extreme, a particular system is tied to a given language, as VisualWorks is to Smalltalk. In contrast, C2 embodies no language assumptions; components may be written in any convenient language. To support this, C2 employs technology and embodies wisdom from previous multi-lingual systems [Kad92] for mapping parameters from one type system to another and avoiding conflicts in runtime language support, heap memory allocation, and use of operating system resources.

Existing systems tend to be rigid in terms of their process mappings. At one extreme, X applications contain exactly two processes, a client and a server. While there is greater process flexibility in VisualWorks and Weaves, both of these systems assume a shared address space. It is only with systems such as GenAvoca, Field/SoftBench, and C2 that simultaneous satisfaction of arbitrary numbers of processes in a non-shared address space is achieved.

While individual systems share key features with the C2 architectural style, the goal of simultaneous satisfaction of implicit invocation via notifications, inter- and intra-process message mechanisms, domain-specific architectural layering, and multi-language and multi-process support differentiates C2 from existing work, and motivates our future work.

5.0 Open Issues

Many issues crucial to the C2 style have been explored in detail and several applications completed. Nonetheless, assessing any new architectural style takes many years. When pipe and filter or client/server paradigms were introduced, it was unlikely that all of the ramifications and required improvements could have been forecast in the beginning. Similarly, we have not yet answered, or even asked, all of the questions about the style. However, we do recognize that certain areas warrant further study.

- As specified in this paper, notifications are reifications of operations that occurred within a component. As such, they are equivalent to messages encapsulating deltas to the state of the component. An alternative is to send out the full state of the component. We chose the state delta approach because of our successful experience with it in the Chiron-1 system. There are circumstances when full state broadcast is more beneficial, such as when the recipient would react to a notification of an event by issuing a series of queries. Choice of the nature of the notifications is orthogonal to other aspects of the architecture. We will attempt to develop characterizations of the situations favoring each choice.
- All of the applications built thus far have been relatively small. The style, on the other hand, is also intended for large-scale systems that reuse components and build extensive multi-level hierarchies. In order to fully support compositionality, is a mechanism needed to support “recursive” application of the style *within* a component?
- Connectors in the architectural style have properties similar to software buses. There are numerous existing bus technologies that may be suitable in the implementation of an architecture. Examples include Chiron 1.4 dispatchers, Tooltalk, Softbench, and CORBA. We need to determine under what circumstances these could or should be used.
- One trade-off that is likely to occur is between scalability and performance. All the trial applications completed thus far have shown excellent performance, but they have also been smaller-scale systems. What will happen when the applications start to grow? Will the threshold of scalability with respect to performance be reached, and when?
- The new architectural style admits fine-grain distribution, where each component and connector may be contained in its own process, may have its own address

space, and may be running on different machines, and even different platforms. When is this appropriate? What operating systems, programming languages, and interprocess communication mechanisms will support the performance required?

6.0 Conclusion

User interfaces of emerging systems are rich and complex. Future systems will be increasingly distributed, complex, multi-media, heterogeneous, and multi-user. Supporting such interfaces in a cost-effective manner demands the use of open architectures, architectures that enable a marketplace of components to flourish. C2 is being developed in an attempt to create the basis for such architectures. The C2 style exploits and generalizes key techniques from a variety of previous systems to achieve this. One notable characteristic is the inability for a component to have dependencies on the technologies upon which it rests. Rather a component is "hopeful" that the components below it will create useful visualizations based on notification of actions that it performs.

A variety of small scale experiments have been conducted to provide initial assessment of the feasibility of the approach. The experiments have been successful: the strong separations enforced by C2 enable radical changes in system structure without significant work. Moreover, the performance of the systems has been very good.

Current and future work encompasses a wide range of activities, including assessing key scalability factors, construction of a development environment, and exploration of how current commercial offerings may be adapted to serve as reusable C2 components.

7.0 Acknowledgments

We would like to acknowledge the students of UCI's graduate course in user interfaces for providing proof-of-concept examples of C2. In particular, we wish to thank N. Medvidovic and J. Shaw for the Petri-Net example, D. Tonne for the exploration of constraints, L. Palen, E. Charne, and K. Anderson for their exploration of adding sound to existing systems, and J. Robbins and P. Oreizy for their prototype of a C2 design environment. Other members of the Chiron-1 and 2 teams have also made key contributions, including C. MacFarlane, G. Johnson, and G. Bolcer. Special thanks to K. Nies for careful review of an earlier draft of this paper. We also thank the ICSE reviewers for their helpful and insightful comments.

8.0 References

[BO94] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1994.

[Cag90] Martin R. Cagan. The HP SoftBench environment: An architecture for a new generation of software tools. *Hewlett-Packard Journal*, 41(3):36–47, June 1990.

[Cou91] Joëlle Coutaz. Architectural Design for User Interfaces. In *Proceedings of the 3rd European Software Engineering Conference, ESEC'91*, pages 7–22, Milan, Italy, October 1991.

[FGNR92] Gerhard Fischer, Andreas Girgensohn, Kumiyo Nakakoji, and David Redmiles. Supporting software designers with integrated domain-oriented design environments. *IEEE Transactions on Software Engineering*, 18(6):511–522, June 1992.

[GS93] David Garlan and Mary Shaw. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing, 1993.

[GR91] Michael M. Gorlick and Rami R. Razouk. Using Weaves for software construction and analysis. In *Proc. Thirteenth Int'l Conf. on Software Engineering*, pp. 23–34, Austin, TX, May 1991.

[Inc94] ParcPlace Systems Inc. Visualworks 2.0. Sunnyvale, California, 1994.

[Kad92] R. Kadia. Issues encountered in building a flexible software development environment: Lessons learned from the Arcadia project. In *Proc. ACM SIGSOFT'92: Fifth Symposium on Software Development Environments*, December 1992.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, Aug./Sept. 1988.

[Mye89] Brad A. Myers. Encapsulating interactive behaviors. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 319–324, Austin, May 1989.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[Pfa83] Günther E. Pfaff, editor. *User Interface Management Systems*, Seeheim, Nov. 1983. Eurographics, Springer-Verlag.

[SN92] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Trans. on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–66, July 1990.

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

[SG86] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, April 1986.

[TJ93] Richard N. Taylor and Gregory F. Johnson. Separations of concerns in the Chiron-1 user interface development and management system. In *Proceedings of CHI '93*, pages 367–374, Amsterdam, April 1993.

[US91] D. Ungar and R.B. Smith. SELF: The Power of Simplicity. *LISP and Symbolic Computation*, 4(3):187–205, July 1991.

[Wor92] The UIMS Tool Developers Workshop. A metamodel for the runtime architecture of an interactive system. *SIGCHI Bulletin*, 24(1):32–37, January 1992.