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

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

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

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

I. Introduction

Software architectural styles are key design idioms [8, 24]. 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 runtime architectures: the client-server style (as exemplified by X windows) and the call-back model, a control model in which application functions are invoked under the control of the user interface. Also well known is the model-view-controller (MVC) style [15], which is commonly exploited in Smalltalk applications. The Arch style is more recent, and has an associated metamodel [38].

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

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

The new style can be informally summarized as a network of concurrent components hooked together by message routing devices. Central to the architectural style is a principle of limited visibility: a component within the hierarchy can only be aware of components "above" it and completely unaware of components which reside "beneath" it. Notions of above and below are used in this paper to support an intuitive understanding of the architectural style. As is typical with virtual machine diagrams found in operating systems textbooks, in this discussion the application code[2] 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 keyboard, mouse, microphone, and so forth.[3]

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.

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

All components have their own thread(s) of control and there is no assumption of a shared address space. At minimum, this means that components may not assume that they can directly invoke other component's operations or have direct access to other components' data. It is 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 further discussed in Section III.E.

A small example serves to illustrate several of these points. In Fig. 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 appropriate for a spring being compressed is played; whenever the stack is popped, the sound of a plate breaking is played.

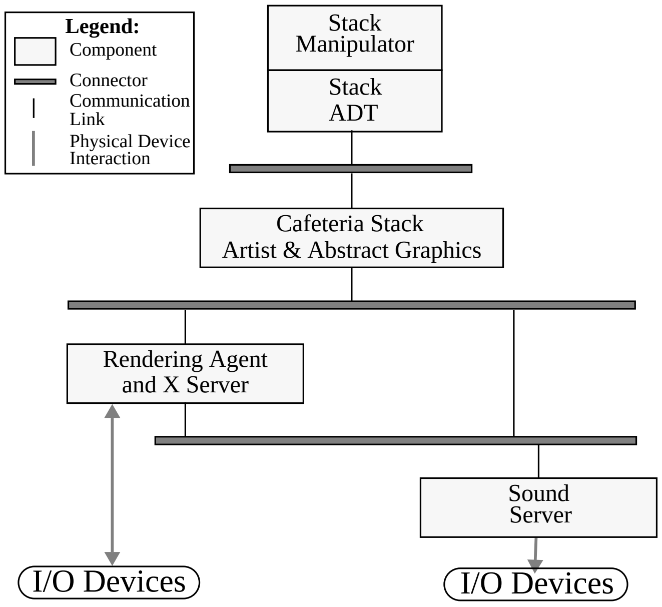


Fig. 1. An audio-visual stack manipulation system.

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 II presents the new architectural style. Section III presents a set of sample applications that have been built to investigate various aspects of the style. Section IV discusses the C2 design environment. Section V discusses related work and Section VI provides practical hints on how to create an architecture in the C2 style. Discussion of open issues and a conclusion round out the paper.

II. 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 components and connectors 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 for the choice of principles.

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 multi-cast of the messages.

A. Components

Components may have state, their own thread(s) of control, and must have a top and bottom domain. The top domain specifies the set of notifications to which the component responds, and the set of requests that the component emits up an architecture. The bottom domain specifies the set of notifications that the 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 of a component, targeted at the user interface software domain, is assumed.[4] Components contain an object with a defined interface, a wrapper around the object, a dialog and constraint manager, and a domain translator, as shown in Fig. 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. The 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 any return values as a notification in the component's bottom domain and sends the notification to the connector below the component.[5]

3. 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.

4. It will be clear from the ensuing discussion that issues concerning composition of an architecture are independent of a component's internal structure, so this assumption is not at all restrictive.

Thus the types of notifications emitted from a component are determined by the interface to its internal object.

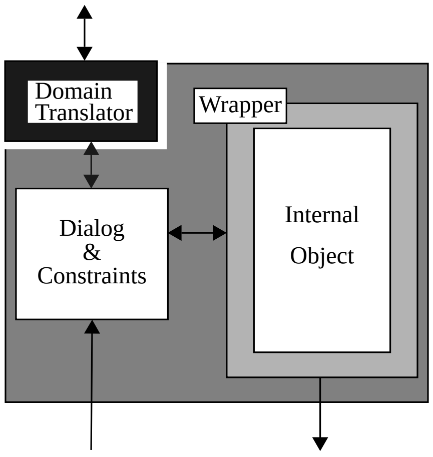


Fig. 2. The Internal Architecture of a C2 Component.

The access routines of the object may only be invoked by the dialog portion of a component. This code, which may have 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.

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. Section II.E presents a detailed discussion of domain translation.

B. Notifications and Requests

Components in an architecture communicate asynchronously via messages. Messages consist of a name and an associated set of typed parameters. 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 Fig. 3. One component manages a binary tree abstract data type (ADT) while the other component manages a depiction of that binary tree.[6] An example notification from the *binary tree 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 *binary tree Artist* component receives the notification and makes calls to its internal object to update the depiction.

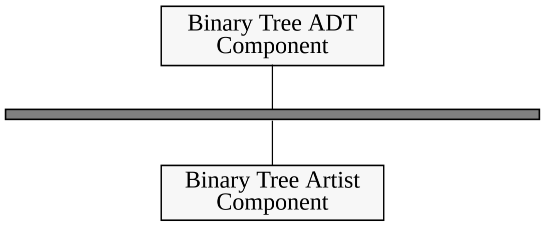


Fig. 3. A partial C2 architecture.

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 *binary tree 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 *binary tree Artist* and sent by the connector to the *binary tree ADT*. The *binary tree ADT* 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 *binary tree Artist* to update its depiction.

1) Interfacing with Legacy Systems: 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 'x' key" and send them *up* an architecture. These toolkit events will need to be converted by C2 bindings to the toolkits 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 and applications. This is not an unreasonable task: we have

5. 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 VII.

6. 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.

already accomplished this for both Motif and OpenLook in Chiron-1 [34].

C. 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.

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.
- Prioritized: The connector defines a priority ranking over its connected components, based on a set of evaluation criteria specified by the software designer during the construction of the architecture. This connector then sends a notification to each component in order of priority until a termination condition has been met. Prioritized connectors are useful for cases in which several components connected to one side of the connector perform the same function, e.g., spell-checking a document, with possibly different implementations. In such a case, some computation is needed to select the appropriate destination for a notification. For example, an HTML document needs a spell checker that is able to ignore the HTML markup commands contained therein.
- Message Sink: The connector ignores each message sent to it. This is useful for isolating subsystems of an architecture as well as incrementally adding components to an existing architecture. A developer can connect a new component to the architecture and then "turn on" its connector, by changing its filtering policy, when the component is ready to be tested.

A connector has an upper and lower domain, defined by the components and connectors attached to it.[7] These are described in the following section.

D. 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 [28].

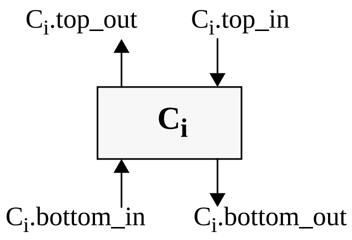


Fig. 4. C2 Component Domains.

7. 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.

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 sent from its top side.

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

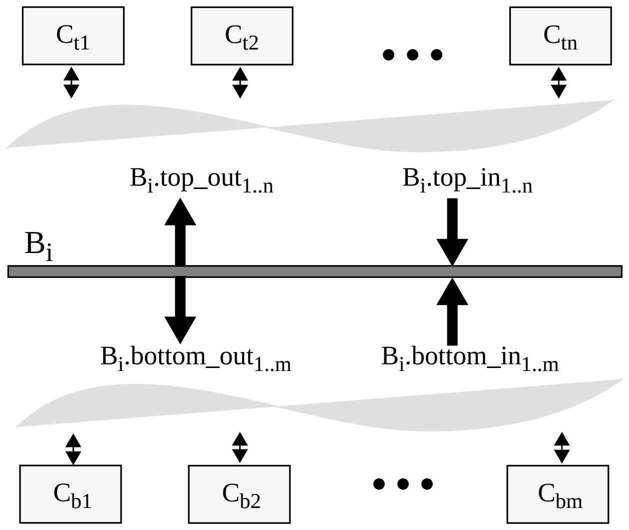


Fig. 5. C2 Connector Domains.

Fig. 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.

Consider the notifications that come in from the components C_{tj} 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 the previous section, 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:

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

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 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 any 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:

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

Finally, they are *not communicating* as follows:

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

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*.

Clearly, the ideal scenario in an architecture would be one where (1) components are fully communicating with the connectors to which they are attached and (2) components' services are fully utilized. However, such a constraint would limit reusability of components across architectures, as discussed in the following section. Therefore, in general, there is no guarantee that a component, C_{bk}, will receive notifications in reply to a request that it issues. In addition to the potential inability of the intended recipient, C_{tj}, to understand the request, this can happen for several other reasons:

- both the request and the resulting notification(s) may be lost across the network and/or delayed due to network failure;
- the nature of the request may be such that C_{tj} is able to respond to it only after receiving other requests. If those requests are not issued, C_{bk} will not get a response;
- C_{tj} may itself need to issue requests to components above it in order to be able to respond to the current request. If it does not receive the required information for any of the above reasons, it will not be able to issue notifications in response to the original request.

The asynchronous nature of components will allow C_{bk} to still perform its function meaningfully in the above cases. C_{bk} may choose to block on other messages for a certain amount of time and/or preserve the part of its context relevant to properly handling the expected notifications. After the specified time, the component may unblock, assuming either that the request was lost or that the intended recipient is unable to respond to the request. The appropriate action in such a case will depend on the component and the situation.

Finally, by utilizing the functions and relationships specified above, 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 $C_{tj}.bottom\_in$). All such relationships can be deduced from the complete formal definition of the C2 style [18], which uses Z [31] as its formal notation.

The formal definition enables 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 design environment.

E. Domain Translation

Since a component has no knowledge of the interfaces of components below it and does not directly issue requests to those components, a component is independent of its substrate layers. This 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 domain translation. Domain translation is a transformation of the requests issued by a component into the specific form understood by the recipient of the request, as well as the transformation of notifications received by a component into a form understood by that component. This transformation process is encapsulated in the domain translator part of a component, as shown in Fig. 2.

Domain translation of a single request or notification consists of at least two steps, described below. While this discussion applies equally to requests and notifications, for simplicity, examples will mainly discuss requests.

- Message Name Matching: a mismatch may occur because a message name is different than expected. For example, a component may issue a "stack_pop" request to a component which has a "pop_stack" entry point. In this case, domain translation involves a simple name replacement.
- Parameter Matching: a mismatch may occur in the number, ordering, type, and units of parameters. As an example of parameter matching difficulties, suppose component A issues a "make_alarm" request giving a time delay in seconds before component B issues an "alarm" notification. A parameter mismatch occurs if component B only understands compound time

values of seconds and milliseconds, or only understands time values if they are given in milliseconds.

Other factors may potentially affect domain translation. For example, if a component issues a notification containing a complete state, and the receiving component expects a state change instead, the domain translator might have to store the state and manually generate the expected state delta. Factors external to a component's interface, such as time performance or memory usage, might also affect domain translation.

Simple domain translations, such as name replacement and parameter order swapping could be specified by the system architect using the facilities of the development environment. We are hopeful that these simple translations will frequently be automatable, particularly in cases where there exists an approximate one-to-one correspondence between the messages received by a component and those it actually understands. More commonly, however, this task will at least partly be guided by the system architect. A human agent is needed to provide semantic interpretation for both the component's top domain and the interface presented by the connector above it, especially in cases of partial communication and/or service utilization.

More difficult domain translations such as the generation of missing parameters and unit conversions will require manual generation of domain translators using either a scripting language or a programming language.

Domain translation unavoidably adds overhead to the message passing process. Except in rare cases, this is less than the cost of passing the message itself, and is not a major source of inefficiency, however. Domain translation can be viewed as a tradeoff between slightly diminished message passing efficiency and the ability to reuse components as-is.

The need for domain translation can be considerably reduced by the adoption of standard interfaces for similar components. Exemplifying this approach are domain-specific architectures [35], where similar components are characterized by similar interfaces, certain component configurations are common, and usual patterns of component usage are known to both the architect and the design environment.

F. 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 these principles, of course, characterize a variety of other systems.

- *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 notification, not knowing whether any component 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-threaded* - 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 would be unreasonable 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.
- *Implementation separate from architecture* - many potential performance issues can be remedied by separating the conceptual architecture from actual implementation techniques. For example, while the C2 style disallows any assumptions of shared threads of control and address spaces in a conceptual architecture, substantial performance gains may be made in a particular implementation of that architecture 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.

III. 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 to assess 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 constructed a variety of research prototypes. 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.

With two exceptions, the examples in this section make use of Chiron-1 [34] as a testbed. This allowed the rapid exploration of several key C2 concepts without having to devote a large development effort on support infrastructure. The first exception is the modeling workbench which used a rapid prototyping environment, described in Section III.D. The second exception, the KLAX example, given Section III.E, came after these initial experiments and validated their findings by reproducing their results in an environment completely separate from Chiron-1, in addition to exploring other C2 concepts. Note that with the exception of the KLAX example, performance descriptions are anecdotal and should be interpreted as subjective descriptions of a user's experience with the application. In the KLAX example, we provide performance metrics to reinforce these subjective descriptions.

A. Petri Net Tool with Multiple Presentation Components

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 are depicted by an empty circle, a point (filled circle), and a line respectively. Every time a transition is fired, the shapes of all the places connected to that transition potentially 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 Fig. 6. The *Petri net layout artist* maintains the coordinates of places, transitions, and arcs, addresses issues of adjacency, and maintains logical associations with *Petri net 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. Therefore, when a place is added, deleted, or repositioned, or its number of tokens changes due to a transition firing, the *layout artist* broadcasts the appropriate notifications and only the artist maintaining the presentation of places with the specified number of tokens responds to them.

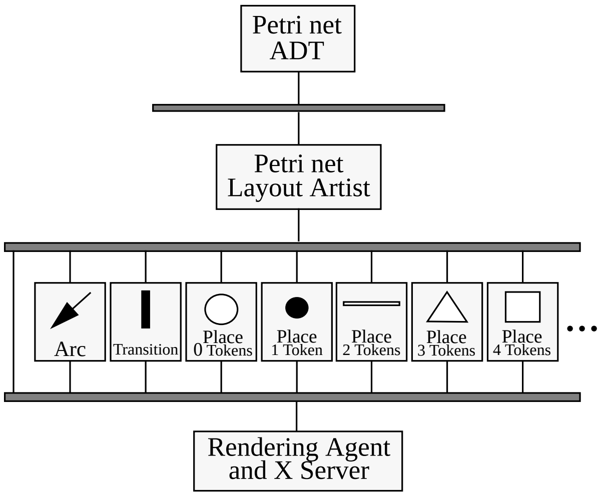


Fig. 6. Petri net places are polygons whose number of sides equals the number of tokens.

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.

B. 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 Fig. 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 *rendering agent*. These notifications are processed, constraints applied, and requests sent back to the *layout* artist, so that positions may be updated.[8]

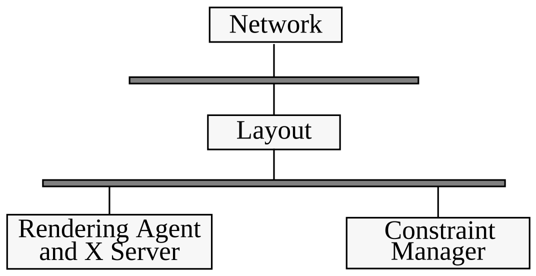


Fig. 7. A constraint manager is added to a drawing editor system.

This example highlights one goal of our work: to be able to include a constraint manager in a UI architecture, where its inclusion had not been previously planned. Most constraint managers described in the UI literature are large and often intertwined with the rest of the system [19]. 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. A more significant example of the use of an externally produced constraint manager is discussed in Section III.E.

C. Distributed, Multi-user Meeting Scheduler

A meeting scheduling application was built to explore how use of the C2 architectural style affects development of distributed applications supporting multiple users. The application allows multiple users working on different machines to schedule meetings with each other. Each user’s display consists of three windows. A “meeting proposal” window allows scheduling of a new meeting. A “schedule” window displays a list of the meetings the user had agreed to attend. An “invitation” window appears each time the user is invited to a meeting and allows the user to either accept or decline the invitation.

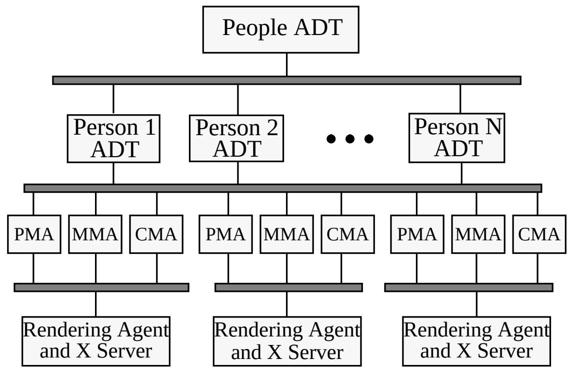

People ADT
Person 1 ADT
Person 2 ADT
Person N ADT
PMA
MMA
CMA
PMA
MMA
CMA
PMA
MMA
CMA
Rendering Agent and X Server
Rendering Agent and X Server
Rendering Agent and X Server

Fig. 8. The C2 architecture for the distributed meeting scheduler.

The C2 architecture for this experiment is shown in Fig. 8. A centralized *people ADT* keeps track of all the people using the

8. 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.

meeting scheduler. A *person ADT*, which manages an individual user's schedule, is created for each user in the system. These *person ADTs* request a list of people using the system from the *people ADT*. Each user also has a set of artists which allow interaction with the system. The personal meeting artist (*PMA*), the make meeting artist (*MMA*), and the create meeting artist (*CMA*) manage the schedule window, the meeting proposal window, and the invitation window respectively.

A unique aspect of this experiment is that connectors, which are used to broadcast notifications about new meetings, invitation acceptances and declinations, span both multiple users and multiple machines. For instance, the middle connector routes meeting invitation messages from the *MMA* to the *person ADTs* of the users who have been invited. Additionally, each person has an individual bus which routes their rendering information to rendering components. In contrast to other meeting scheduling applications such as Schedule+[9] and Calendar Manager[10], there is no centralized meeting schedule server or ADT. Instead, each user has an individual meeting schedule ADT, namely the *person ADT*. This ADT only stores information about the user's scheduled meetings.

This application demonstrates the feasibility of using the C2 style to develop applications which support multiple users. It also demonstrates that C2 messages could be sent to multiple machines across a network without a noticeable degradation in performance.

D. Workbench for Experimenting with Multi-Level Software Architectures

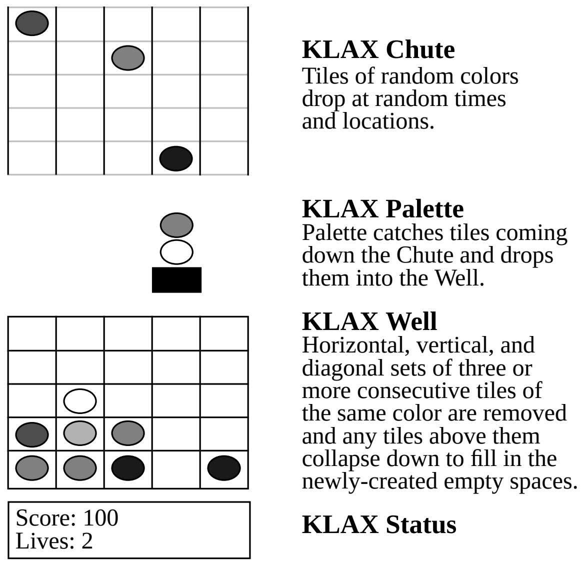
This exercise focused on building a modeling workbench to explore issues related to C2. A model was built of a simple stack application. That model was embellished with various design features to explore trade-offs such as the choice between broadcasting an abstraction of the state of a component when it changes or broadcasting an abstraction of the event that caused the modification. This experiment also explored 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 in combination with runtime domain representation.

The modeling was done programmatically in Self [36]. Self allowed convenient what-if analysis via both programmatic changes and direct manipulation. Components were modeled as Self objects. Paths between connectors and components were modeled as pointers. Messages were modeled as Self messages. 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 insights which resulted and which are reflected in this paper, the need for a modeling and design environment to visualize and manipulate C2 style architectures was clearly highlighted.

9. SCHEDULE+ is a registered trademark of Microsoft Corporation.
10. Calendar Manager is a registered trademark of Sun Microsystems.

E. KLAX^{TM} Example

This exercise involved implementing a version of the video game KLAX.[11] A description of the game is given in Fig. 9. This particular application was chosen as a useful test of the C2 style concepts in that the game is based on common computer science data structures and the game layout maps naturally to modular artists. Also, the game play imposes some real-time constraints on the application, bringing performance issues to the forefront.



KLAX Chute
Tiles of random colors drop at random times and locations.
KLAX Palette
Palette catches tiles coming down the Chute and drops them into the Well.
KLAX Well
Horizontal, vertical, and diagonal sets of three or more consecutive tiles of the same color are removed and any tiles above them collapse down to fill in the newly-created empty spaces.
Score: 100
Lives: 2
KLAX Status

Fig. 9. A snapshot and description of our implementation of the KLAX^{TM} video game.

The design of the system is given in Fig. 10. The components that make up the KLAX game can be divided into three logical groups. At the top of the architecture are the components which encapsulate the game's state. These data structure components are placed at the top since game state is vital for the functioning of the other two groups of components. These ADT components receive no notifications, but respond to requests and emit notifications of internal state changes. ADT notifications are directed to the next level where they are received by both the game logic components and the artists components.

The game logic components request changes of ADT state in accordance with game rules and interpret ADT state change notifications to determine the state of the game in progress. For example, if a tile is dropped from the well, the *relative positioning logic* determines if the *palette* is in a position to catch the tile. If so, a request is sent to the palette to catch the tile. Otherwise, a notification is sent that a tile has been dropped. This notification is detected by the *status logic* causing the number of lives to be decremented.

The artist components also receive notifications of ADT state changes, causing them to update their depictions. Each artist maintains the state of a set of abstract graphical objects which, when modified, send state change notifications in hope that a lower-level graphics component will render them. The *tile artist* provides a flexible presentation level for tiles. Artists maintain information about the placement of abstract tile objects. The *tile artist* intercepts any notifications about tile objects and recasts

11. KLAX is trademarked 1991 by Atari Games.

them to notifications about more concrete drawable objects. For example, a “tile-created” notification might be translated into a “rectangle-created” notification. The *layout manager* component receives all notifications from the artists and offsets any coordinates to ensure that the game elements are drawn in the correct juxtaposition.

The *graphics binding* component receives all notifications about the state of the artists’ graphical objects and translates them into calls to a window system. User events, such as a key press, are translated into requests to the artist components.

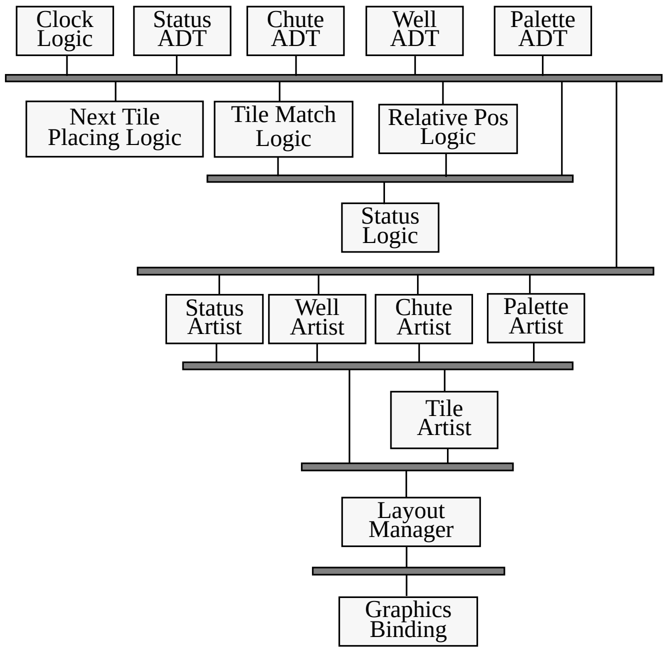


Fig. 10. Conceptual C2 architecture for KLAX. Note that the Logic and Artist layers do not communicate directly and are in fact siblings. The Artist layer is shown below the Logic layer since the components in the Artist layer perform functions closer to the user (See Footnote 2).

The KLAX architecture is intended to support a family of “falling-tile” games. The components were designed as reusable building blocks to support different game variations. One such variation on KLAX is discussed below, following the discussion of implementation issues.

To support the implementation of the KLAX architecture, a C++ framework consisting of classes for C2 concepts such as components, connectors, and messages was developed. The size of this reusable framework is approximately 3100 commented lines of C++ code and it supports a variety of implementations, discussed below, for a single conceptual architecture. This framework is also significant since it allowed us to cut our ties with Chiron-1 as a testbed for C2 concepts and to integrate an external component, the Xlib toolkit, by wrapping it with a C2 component, the *graphics binding*. The KLAX implementation built using the framework consists of approximately 8100 additional lines of commented C++ code.

The framework allowed mappings of the KLAX conceptual architecture to several implementations, including single-process, multi-threaded, and multi-process implementations. The first implementation placed all components in a single UNIX process with a scheduler that distributed a time-slice to each component. The second implementation mapped the components into four separate threads within a single UNIX process. The third implementation split the architecture into three UNIX processes. One process contained all of the ADTs, game logic, and artist components, except for the *tile artist*, which was placed in its own process. The last process contained the *graphics binding* and the *layout manager* components. It is important to note that in these three implementations the components were never changed; instead, the end result was obtained by attaching the components to the appropriate framework classes at architecture construction time. For instance, the multi-process implementation was obtained by substituting interprocess connectors for single-process connectors.

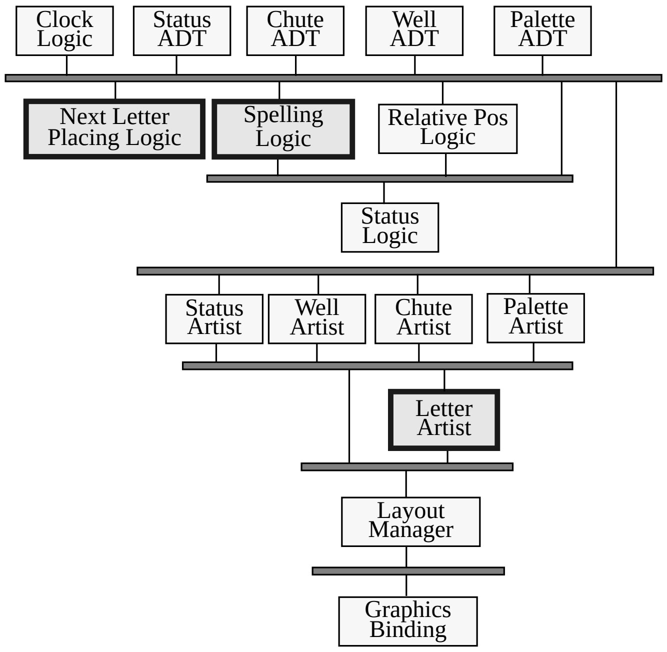


Fig. 11. A variation on KLAX. By replacing three components from the original architecture, the game turns into one whose object is to spell words horizontally, vertically, or diagonally.

The multi-process implementation allowed the exploration of a multi-lingual implementation. The original C++ tile process was reimplemented using the Ada programming language.[12] The only functional difference between the Ada tile process and the C++ tile process was that the presentation of tiles was changed from ovals to rectangles. The Ada tile process can be swapped with the C++ tile process dynamically. The sole effect of this swap on a running KLAX application was that the representation of the tiles on the screen would change from ovals to rectangles or vice-versa.

A variation of the original architecture, shown in Fig. 11, involved replacing the original tile matching, tile placing, and tile artist components with components which instead matched, placed, and displayed letters. This transformed the objective from matching the colors of tiles to spelling words. Each time a word was spelled correctly, it would be removed from the well. The *spelling logic* component wrapped an existing spell-checker, written in C.

Another variation of the original architecture involved incor-

12. A small portion of the C++ framework (multi-process connectors and the C2 message class) was reimplemented in Ada as well.

porating a research-off-the-shelf constraint management system, SkyBlue [29], into the application. Constraint-management code, such as specifying the left and right boundaries of palette's movement and ensuring that the tiles on the palette move along with the palette, was dispersed throughout the original application. This variation replaced that code with SkyBlue constraints. Furthermore, SkyBlue needed to be adapted to communicate with other KLAX components via C2 messages. This was accomplished by placing SkyBlue inside the *layout manager*, as shown in Fig. 12, thus creating a constraint management component in the C2 style

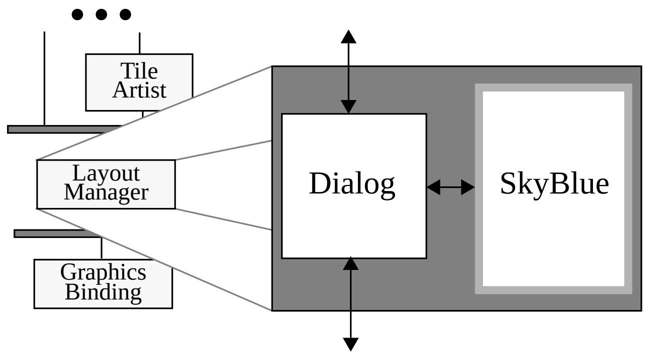


Fig. 12. The SkyBlue constraint management system is incorporated into KLAX by placing it inside the Layout Manager component. Layout Manager's dialog handles all the C2 message traffic.

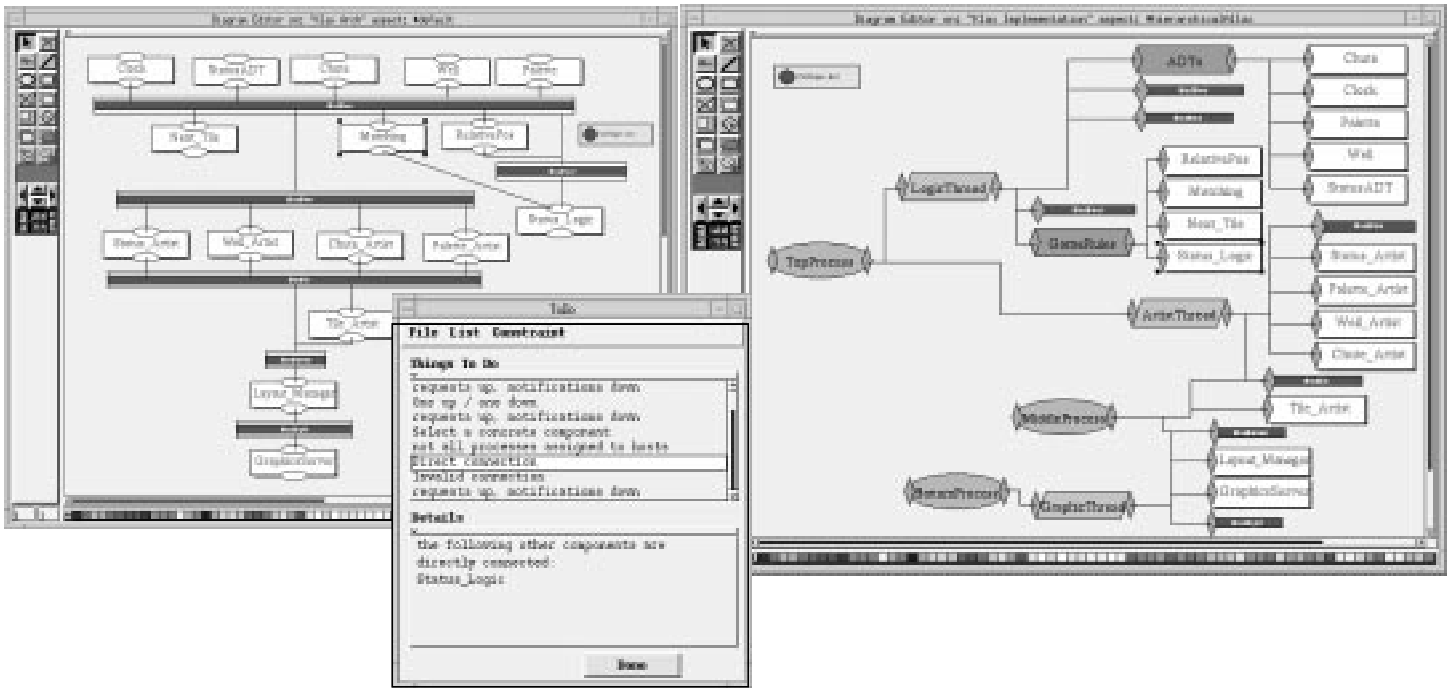
Performance of the implementations was good on a Sun Sparc2 workstation, easily exceeding human reaction time if the clock component used short time intervals. Although we have not yet tried to optimize performance, benchmarks indicated our current framework can send 1200 simple messages per second when sending and receiving components are in the same process. In the KLAX system a keystroke typically caused 10 to 30 message sends, and a tick of the clock component typically caused 3 to 20.

Several important aspects of the C2 style were explored in this effort. Three external components, Xlib, SkyBlue, and the spelling checker, were integrated. Dynamic substitution of components was demonstrated. Component substitution was used not only to provide alternative presentations of tiles but also to transform the application into spelling KLAX, another game in the same application family. This example also demonstrated support for multi-lingual components in C, C++, and Ada. A reusable C2 framework that supports multiple implementations of a given C2 architecture was also explored.

IV. Argo: The C2 Design 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 individual components. For that reason, development tools that operate on architectural specifications are as important as tools that work on individual components. This section describes Argo, a design environment for building C2-style architectures.

The C2 design environment is an editor in which designers construct an architectural 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 generate running programs for that system. Because diagrams are so effective for describing software architectures, the C2 design environment uses a graphical front end to a precise internal representation. The notation used to describe C2 style architectures is a connected graph of software components and connectors. Argo allows for direct manipulation of connected graphs and various annotations on them. The left side of Fig. 13 shows the KLAX conceptual architecture represented in Argo: white rectangles represent software components, black bars represent



Things To Do
requests up, notifications down
One up / one down
requests up, notifications down
Select a concrete component
not all processes assigned to hosts
Direct connection
Invalid connection
requests up, notifications down

Details
the following other components are
directly connected:
Status_Logic

Fig. 1. Fig.13. Screenshots from Argo. Left: Conceptual architecture. Center: To-do list with item indicating a problem with Status_Logic. Right: Implementation Architecture. Small shapes attached to components are user interface affordances for making connections.

connectors, arcs represent communication pathways, small ovals on the components represent the communication ports of each component. The right side of Fig. 13 shows the mapping from the conceptual architecture to an implementation architecture. The conceptual components and connectors are the leaves of an is-composed-of tree. Internal nodes in that tree represent operating system threads and processes which contain components and connectors.

Dialog boxes provide access to the annotations on the components, connectors, and processes. Argo represents the interface of each component as a collection of message signatures, one for each notification and request in its domain. Environmental constraints on components are represented as a set of attributes: required memory size, supported operating systems, etc. Argo associates information used to generate working applications with each component, including the C++ class that implements the component, the names of source files used to build the component, and the required arguments to the component constructors.

A. Critical Feedback

Critics are active agents in the design environment that help the designer make better design decisions by delivering advice which is relevant and timely. The design environment itself does not prevent the designer from entering invalid designs. Rather, critics continually advise the designer about potential syntactic and semantic errors or oversights. Argo allows designers to move from one valid design state to another, even if invalid design states must be traversed along the way. The current version of Argo has roughly twenty critics; more critics are planned for future versions.

Because critical feedback is continuous and simultaneous with design manipulation, feedback mechanisms must not disrupt the designer's train of thought. Several mechanisms are used in Argo to limit the execution of critics and manage their feedback. Critical feedback is presented to the designer via a to-do list in its own window. Items in the list are prioritized, and selecting an item gives a brief explanation and selects the offending design element. At any given time, the designer will have many items on his list and is free to act on them on his own initiative.

Our implementation of critics is more object-oriented than rule-based: the design environment does not critique the design so much as the objects in the design representation critique themselves. Each component representation object in Argo carries its own critics, and the introduction of a new component representation object potentially introduces new critics. When the critique command is issued to a set of components, each of them asks their critics to check their conditions. When a condition is violated, the critic posts its item on the user's to-do list. [27] describes critics in more detail.

B. Code Generation

In the conceptual architecture of a software system, each component has its own thread of control and executes in its own memory space. When architectures are implemented, those assumptions may introduce too much overhead if fine-grained components are used. Thus, Argo and our C2 class framework allow the designer to specify the mapping from conceptual to efficient implementation architectures. The diagram shown in Fig. 13 represents most of that mapping, Argo finishes the mapping via default rules.

Argo can generate some of the code needed to build an executable application. Currently, Argo assumes that the components are implemented as C++ classes which fit into a C2 class framework, but multi-lingual support is a goal of future work. In that framework, running applications are implemented as instance graphs where each component and connector is an instance, and each bidirectional message pathway is a pair of pointers. Argo finishes the mapping between conceptual and implementation architectures by replacing each conceptual component with an instance of the class for that component, and each conceptual connector with one of two kinds of implementation connectors. Connectors which provide communication between components in a single operating system process pass messages in memory as simple C++ data structures. Connectors which span multiple processes are implemented with the Q interprocess communication mechanism [17].

Argo generates one main procedure for each process in the implementation architecture. Each main procedure creates, connects, and schedules the implementation components for its process. Those processes can be run to form the executing system. The main procedure of the KLAX example and most of its variants can be generated by Argo, the exception being the Ada main program for the Ada *tile artist*. The technical details of Argo and a number of other issues are discussed in [28].

C. Architecture Analysis

The design environment currently supports several types of static analysis on the architecture using pro-active design critics. The analysis performed by these critics include: checking for conformance to the C2 style rules, checking for completeness by matching requests with component services, and checking consistency between different views of the architecture.

Supporting broader classes of analyses requires more information about the architecture. Incorporating more semantic information for components [9] and the overall architecture enables more specific analyses, like predicting deadlock or starvation. Dynamic analysis techniques may also be applied independently or in conjunction with static techniques. These include instrumenting the architecture to check timing and resource constraints, performing runtime consistency checks on components, and checking assertions at the architecture level.

V. 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.

A. 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 [32, 33], as embodied in their mediators concept. Chiron-1 [34], through its transmission of abstract data type modification events to potentially reactive artists, also supports implicit invocation. This is similar to VisualWorks [11], a Smalltalk GUI library based on the Model-View-Controller paradigm [15], 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.

B. Messages and Message Mechanisms

Message mechanisms in existing systems transmit either service requests, events (notifications), objects, or a combination. Existing systems are distinguished by the discipline, if any, they impose on message use, and usage style, which is a result of the system's message use discipline. Both Chiron-1 and X windows [30] 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 [26] and SoftBench [3] systems also use service request and notification messages. Messages in these two systems, however, have no discipline on their use; the two message types are indistinguishable.

In the Weaves system [10], 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 service invocations upon an internal abstract object as messages (or objects).

Experience from the Chiron-1 system indicates that if message traffic occurs across a process boundary in a non-shared address space, then interprocess communications (IPC) becomes a key performance determinant. Experience with the Avoca system [1] 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.

C. Layered Systems

Concentrating solely on the layering in an 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. Chiron-1 has three layers: the application, artists, and graphics server. The Arch Model, which is an extension of the Seeheim [25] model, and the Slinky User Interface MetaModel [38] partition the work of supporting user interfaces into five layers, known as the domain-specific (i.e., "application") component, domain adaptor, dialog, presentation, and interaction toolkit components. The dialog component may be further subdivided to arbitrary levels [4].

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 the application domain. In this, the C2 style is similar to the composable, parameterized components of the GenVoca style [1], 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 GenVoca 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.

D. 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 [13] 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 GenVoca, Field or SoftBench, and C2 that simultaneous satisfaction of arbitrary numbers of processes in a non-shared address space is achieved.

E. Component Interoperability Models

Existing component interoperability models, such as OLE [2] and OpenDoc [20], provide standard communication mechanisms for components. Typically, the model provides a standard format for describing services offered by a component and runtime facilities to locate, load, and execute services of other components. Since these models are concerned with low-level implementation issues and provide little or no guidance in building a system out of components, their use is neither subsumed by or restricted by C2. In fact, these models may be used to realize an architecture in the C2 style.

F. Design Environments

In the terminology of [5], Argo can be summarized as a domain-oriented design environment for the domain of C2 style software architectures. Systems developed at the University of Colorado, such as Janus [6] and Framer [16], use critics to give designers domain specific feedback. Argo's interaction paradigm is similar to that of Janus, although Argo has more ways to control critics and manage their feedback.

Aesop [7] is a generation tool for software architecture design environments which focuses on architectural styles. Aesop interoperates with external analysis and code generation tools and a component repository. Aesop's choice of formalism and external tools allow it to provide more analysis and code generation ability than Argo currently provides. However, Aesop is not organized around critics, and has little support for the designer's task beyond graphical support for the design notations and invocation of the analysis tools.

G. Relating C2 Concepts to Object-Oriented Types

Much of the discussion of component compositionality, reusability, and substitutability can be linked to the terminology of object-oriented (OO) types [21, 22, 23]. Doing so serves a dual purpose: (1) it enables readers whose primary expertise is in the area of OO type theory to relate the concepts and terminology of C2 to those with which they are familiar, and (2) it helps clarify the composition properties of components and connectors. For example, conditions specifying when one component may be substituted for another is akin to subtyping in OO programming languages (OOPL). At the same time, the differences between the presented architectural concepts and typing in OOPL can help identify the limits of applicability of methods and techniques developed in one to the other.

A C2 component may be viewed as an OOPL class at the conceptual level, but they are not identical. The services a component provides are equivalent to a class specification, and the requests it sends correspond to OO messages. However, no OOPL concept corresponds to C2 notifications, whereby state changes of C2 components are reified as messages and no assumptions are made about the existence or the number of recipients of those messages. This results in the possibility of messages being ignored in a C2 architecture, whereas a similar situation would result in a runtime error in an OOPL. The distinction between notifications and requests and the topology imposed by the C2 style on a set of components in an architecture are the major differentiators between C2 and OOPL.

Nevertheless, the similarities between C2 components and OO classes allow us to explore the ramifications of OO subtyping on reusability and substitutability of C2 components. For that reason, we assume that a component is a class in the OO sense, exporting two interfaces, one corresponding to the top and the other to the bottom domain. [23] provides a spectrum of types, from arbitrary subclassing, where methods can be added, deleted, or redefined, to strictly monotone subclassing, where methods can only be added, requiring that even a particular implementation be preserved. An orthogonal distinction is between specification types, e.g., requiring interface conformance, and implementation types, e.g., requiring behavior conformance.

Programming languages generally adopt a single type checking method. On the other hand, C2 allows several subtyping mechanisms. For example, while *interface* conformance is usually not restrictive enough, since it does not preserve the behavior, we found it useful in modifying the original KLAX architecture to create spelling KLAX, as described in Section III.E. *Strictly monotone subclassing* is often too restrictive, as it disallows different versions of the same functionality. However, it can facilitate incremental development of applications and expansion of legacy system functionality in a C2 architecture. Finally, *behavior* as a subtyping method ensures both interface and behavioral conformance, while also allowing particular implementations to be changed to, e.g., optimize a component's performance. This type checking method can be used to select from sets of components during automatic implementation generation.

H. Summary

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. Furthermore, as the central facility for architectural design in C2, Argo builds upon the state of the art in the area of design environments. Finally, we can leverage existing and future work in the area of OOPL by relating the core C2 ideas to OO types.

VI. CREATING AN ARCHITECTURE IN THE C2 STYLE

For the practitioner, a pertinent question is how one arrives at an architecture in the C2 style. While we do not as yet claim to have a definitive answer, our experience with the style has imparted insights useful to other designers. Designing an architecture involves an iterative process of subdividing functionality into components, determining the external interfaces of components, and positioning components within a C2 architecture. Implementing an architecture requires reusing or implementing a message passing infrastructure for connectors and developing a component template. These design and implementation steps are discussed below.

Subdivision of system functionality into components in the C2 style is essentially the traditional problem of modular decomposition. As such, components which encapsulate ADTs, or which encapsulate functionality likely to change, are appropriate C2 components, and would be appropriate modules in any style. In particular, the C2 style affords good separation of concerns by supporting implicit invocation via notification. This is especially evident in the separation of concerns between an ADT and a component which visualizes it. To date, our modular decompositions have been heavily influenced by previous experience with the Chiron-1 system and its separation between ADTs, artists, and graphics server; these three elements are visible in all of our examples.

Specifying the external interface of a component requires a determination of which notifications and requests the component will process. This involves making a translation between constructs in the component implementation language and requests or notifications. For example, on a component's bottom interface, publicly accessible methods of an object can be translated into understood requests, and modifications to the object's data can be translated into notifications. On the component's top interface, calls to another component are translated into issued requests, and understood notification messages are translated into existing operations within the component. Seemingly simple once the component's functionality is known, this activity is complicated by an architecture's desired topology. In the simplest case, a component at the top of an architecture will not process notifications since it will never receive any, while a component on the bottom will only process notifications; this then affects the design of the external interface of the component. However, a component in the middle of an architecture may process both notifications and requests since it will likely receive both.

When determining the topology of an architecture, there are several principles which can inform the positioning of a component within a hierarchy. The first such principle is the producer-consumer relationship for requests and notifications. If component A emits notifications that component B uses, then B must be placed below A in an architecture, much as artists are always below ADTs. This is a direct consequence of the rule that notifications are always transmitted down an architecture. The second principle is potential for substitution. In all C2 examples, the graphics binding is always at or near the bottom because it is easier to substitute a different graphics component into an architecture if its functionality is accessed solely via implicit invocation. The third principle is the grouping of like components, as exemplified by the grouping of ADTs into a single layer in the KLAX architecture, shown in Fig. 10. This grouping principle supports extensions to the architecture, since a layer of like components provides a location for the addition of new, similar components.

Once the components and architectural topology have been designed, implementation may begin. Infrastructure support needed for small- to moderately-sized C2 applications, such as the ones described in Section III, is minimal. This, coupled with simple style rules, results in a low-entry barrier for building C2 applications. The object-oriented component and message passing framework consists of 12 classes for C2 concepts, such as components, connectors, and messages, as shown in Fig. 14. The size of the framework is approximately 3100 commented lines of C++ code and it was built in 150 programmer hours. As discussed in Section III.E, the framework supports a variety of implementation configurations for a given conceptual architecture. Furthermore, it provides structure for component implementations and eliminates many repetitive programming tasks.

```
C2Object
  ├──C2Message
  ├──C2Port
  └──C2Brick
       ├───C2Connector
       │     ├──C2Connector_SameProcess
       │     └──C2Connector_IPC
       └───C2Component
             ├──C2Architecture
             │    └──C2Architecture_Threads
             ├──C2ADT
             └──C2Artist
```

Fig. 13. The C++ object-oriented framework used in the development of KLAX.

VII. 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 and optimizations 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. There are circumstances when full state broadcast is beneficial, such as when an operation causes a complex state change to occur within the component. Without full state broadcast, this would cause the recipient of the notification to react by either issuing a series of requests to the component to ascertain the new state, or to infer the new state by duplicating much of the semantics of the above component. Thus, for example, the *well* component of the KLAX example broadcasts full state notifications. Simple well operations, such as removing a tile, can cause a complex change in state due to the well's gravity semantics. If operation-based notifications were used, a listening component, such as the well artist, would either have to query the entire state of the well, or duplicate gravity semantics. In this case, full state notifications greatly reduce the complexity of listening components. Choice of the nature of the notifications is orthogonal to other aspects of the architecture.
- 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 [12], SoftBench, and CORBA [20]. We need to determine under what circumstances these could or should be used.
- A connector could potentially make use of transaction information provided by a component to manage the flow of messages to it. Transactions would be one method of handling the complexity of message sequences between components. This would relieve a component's dialog from having to contain code to handle out-of-order messages and the like. We need to determine what transaction scheme is appropriate for C2 architectures and how transaction information is specified by a component.
- 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?
- 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 KLAX example from Section III.E is one data point on this issue. Even with three large legacy components integrated, performance of the application was acceptable.
- Our work on the trial applications and the design environment has highlighted the need for suitable architectural description (ADL) and interface definition (IDL) languages. Early prototypes of both an ADL and an IDL for C2 exist and have been used successfully as the design notation during the development of the KLAX example from Section III.E. However,

much work remains to be done on both the languages and the corresponding code generation and analysis tools.

- 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?

VIII. Conclusion

User interfaces of emerging systems are rich and complex. Future systems will be increasingly distributed, complex, multimedia, 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 perform useful work 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 design environment, and exploration of how current commercial and research offerings may be adapted to serve as reusable C2 components.

IX. Acknowledgements

We would like to acknowledge the students of UCI's graduate course in user interfaces for providing initial proof-of-concept examples of C2. In particular, we wish to thank G. Wong, J. Shaw, D. Tonne, L. Palen, and E. Charne. Other members of the Chiron-1 team also made key contributions, including C. MacFarlane, G. Johnson, and G. Bolcer. Finally, we thank the referees of TSE and ICSE-17 for their helpful reviews.

X. References

[1] D. Batory and S. 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 1992.

[2] K. Brockschmidt. *Inside OLE 2*. Microsoft Press, 1994.

[3] M. R. Cagan. "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools." *Hewlett-Packard Journal*, 41(3):36–47, June 1990.

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

[5] G. Fischer, A. Girgensohn, K. Nakakoji, and D. Redmiles. "Supporting Software Designers with Integrated Domain-Oriented Design Environments." *IEEE Transactions on Software Engineering*, 18(6):511–522, June 1992.

[6] G. Fischer, A. C. Lemke, T. Mastaglio, and A. I. Morch. "Critics: an Emerging Approach to Knowledge-Based Human-Computer Interaction." *International Journal of Man-Machine Studies*, 35(5):695–721, November 1991.

[7] D. Garlan, R. Allen, and J. Ockerbloom. "Exploiting Style in Architectural Design Environments." In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, December 1994.

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

[9] J. A. Goguen. "Reusing and Interconnecting Software Components." *IEEE Computer*, pages 16–28, February 1986.

[10] M. M. Gorlick and R. R. Razouk. "Using Weaves for Software Construction and Analysis." In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 23–34, Austin, TX, May 1991.

[11] ParcPlace Systems Inc. *VisualWorks 2.0 User's Guide*. Sunnyvale, California, 1994.

[12] A. Julienne and B. Holtz. *Tooltalk and Open Protocols: Inter-Application Communication.* SunSoft Press/Prentice Hall, April 1993.

[13] R. Kadia. "Issues Encountered in Building a Flexible Software Development Environment: Lessons Learned From the Arcadia Project." In *Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments*, pages 169–180, Tyson's Corner, Virginia, December 1992.

[14] R. Kazman, L. Bass, G. Abowd, and M. Webb. "SAAM: A Method for Analyzing the Properties of Software Architectures." In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 81–90, Sorrento, Italy, May 1994.

[15] G. E. Krasner and S. T. Pope. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80." *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[16] A. Lemke and G. Fischer. "A Cooperative Problem Solving System for User Interface Design." In *Eighth National Conference on Artificial Intelligence*, pages 479–484, Boston, MA, USA, July 1990. AAAI.

[17] M. J. Maybee, D. H. Heimbigner, and L. J. Osterweil. "Multilanguage Interoperability in Distributed Systems: Experience Report." In *Proceedings of the Eighteenth International Conference on Software Engineering*, Berlin, Germany, March 1996. Also issued as CU Technical Report CU-CS-782-95.

[18] N. Medvidovic. "Formal Definition of the Chiron-2 Software Architectural Style." UCI–ICS Technical Report UCI-ICS-95-24, Department of Information and Computer Science, University of California, Irvine, July 1995.

[19] B. A. Myers. "Encapsulating Interactive Behaviors." In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 319–324, Austin, May 1989. Association for Computing Machinery.

[20] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc., 1996.

[21] J. Palsberg and M. I. Schwartzbach. "Type Substitution for Object-Oriented Programming." In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications/European Conference on Object-Oriented Programming*, pages 151–160, Ottawa, Canada, October 1990.

[22] J. Palsberg and M. I. Schwartzbach. "Object-Oriented Type

Inference.” In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 146–161, Phoenix, AZ, USA, October 1991.

[23] J. Palsberg and M. I. Schwartzbach. “Three Discussions on Object-Oriented Typing.” *ACM SIGPLAN OOPS Messenger*, 3(2):31–38, 1992.

[24] D. E. Perry and A. L. Wolf. “Foundations for the Study of Software Architecture.” *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[25] G. E. Pfaff, editor. *User Interface Management Systems*, Seeheim, FRG, November 1983. Eurographics, Springer-Verlag.

[26] S. P. Reiss. “Connecting Tools Using Message Passing in the Field Environment.” *IEEE Software*, 7(4):57–66, July 1990.

[27] J. E. Robbins and D. F. Redmiles. “Software Architecture Design from the Perspective of Human Cognitive Needs.” In *Proceedings of the California Software Symposium*, Los Angeles, CA, USA, April 1996.

[28] J. E. Robbins, E. J. Whitehead Jr., N. Medvidovic, and R. N. Taylor. “A Software Architecture Design Environment for Chiron-2 Style Architectures.” Arcadia Technical Report UCI-95-01, University of California, Irvine, January 1995.

[29] M. Sannella. “SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction.” In *Proceedings of the Seventh Annual ACM Symposium on User Interface Software and Technology*, pages 137–146, Marina del Rey, California, November 1994.

[30] R. W. Scheifler and J. Gettys. “The X Window System.” *ACM Transactions on Graphics*, 5(2), April 1986. Actually appeared June 1987.

[31] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, New York, 1989.

[32] K. J. Sullivan. *Mediators: Easing the Design and Evolution of Integrated Systems*. Ph.D. thesis, University of Washington, 1994. Available as UW technical report UW-CSE-TR-94-08-01.

[33] K. J. Sullivan and D. Notkin. “Reconciling Environment Integration and Software Evolution.” *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[34] R. N. Taylor, K. A. Nies, G. A. Bolcer, C. A. MacFarlane, K. M. Anderson, and G. F. Johnson. “Chiron-1: A Software Architecture for User Interface Development, Maintenance, and Run-Time Support.” *ACM Transactions on Computer-Human Interaction*, 2(2):105–144, June 1995.

[35] W. Tracz. “DSSA (Domain-Specific Software Architecture) Pedagogical Example.” *Software Engineering Notes*, 20(4), July 1995.

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

[37] E. J. Whitehead Jr., J. E. Robbins, N. Medvidovic, and R. N. Taylor. “Software Architectures: Foundation of a Software Component Marketplace.” In D. Garlan, editor, *Proceedings of the First International Workshop on Architectures for Software Systems*, pages 276–282, April 1995.

[38] The UIMS Tool Developers Workshop. “A Metamodel for the Runtime Architecture of an Interactive System.” *SIGCHI Bulletin*, 24(1):32–37, January 1992.