Exploiting Architectural Style to Develop a Family of Applications

Nenad Medvidovic and Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92697-3425
{neno,taylor}@ics.uci.edu

Abstract -- Reuse of large-grain software components offers the potential for significant savings in application development cost and time. Successful reuse of components and component substitutability depends both on qualities of the components reused as well as the software context in which the reuse is attempted. Disciplined approaches to the structure and design of software applications offers the potential of providing a hospitable setting for such reuse. We present the results of a series of exercises designed to determine how well "off-the-shelf" constraint solvers could be reused in applications designed in accordance with the C2 software architectural style. The exercises involved the reuse of SkyBlue and Amulet's one-way formula constraint solver. We constructed numerous variations of a single application (thus an application family). The paper summarizes the style and presents the results from the exercises. The exercises were successful in a variety of dimensions; one conclusion is that the C2 style offers significant potential for the development of application families and that wider trials are warranted.[1]

Index Terms **-- architectural styles, message-based architectures, application families, graphical user interfaces (GUIs), constraint management, component-based development.**

I. Introduction

Software architecture research is directed at reducing costs of developing applications and increasing the potential for commonality between different members of a closely related product family. One aspect of this research is development of software architectural styles, canonical ways of organizing the components in a product family [10], [27]. Typically, styles reflect and leverage key properties of one or more application domains and recurring patterns of application design within those domains. As such, architectural styles have the potential for providing structure for off-the-shelf (OTS) component reuse.

However, all styles are not equally well equipped to support reuse. If a style is too restrictive, it will exclude the world of legacy components. On the other hand, if the set of style rules is too permissive, developers may be faced with all of the well documented problems of reuse in general [3], [9], [13], [35]. Therefore, achieving a balance, where the rules are strong enough to make reuse tractable but broad enough to enable integration of OTS components, is a key issue in formulating and adopting architectural styles.

Our experience with C2, a component- and message-based style for GUI software [38], [39] indicates that it provides such a balance. In a series of exercises, we were able to integrate several OTS components of various granularities into architectures that adhere to the rules of C2. This paper focuses on a subset of these exercises, in which we successfully integrated two externally developed UI constraint solvers into a C2 architecture: SkyBlue [32] and Amulet's one-way formula solver [17]. In doing so, we were able to create several constraint maintenance components in the C2 style, enabling the construction of a large family of applications. We describe the details of these exercises and the lessons we learned in the process.

The remainder of the paper is organized as follows. Section II describes the rules and intended goals of C2, as well as its relationship to the research that has preceded and influenced it. Part of the material in this section is condensed from a more detailed exposition on the style, given in [39]. Section III discusses our approach to providing implementations for architectures built according to the rules of C2. Section IV presents a detailed overview of the architecture and implementation of KLAX, the application used as the basis for our exercises. Section V motivates the need for a constraint manager in KLAX and describes the particular KLAX constraints we decided to maintain in an external constraint solver. Section VI discusses the design and implementation issues encountered in integrating SkyBlue and Amulet's constraint manager with the architecture. The library of KLAX components created in the process of including SkyBlue and Amulet is described in Section VII. A discussion of several instances of the KLAX application family built with the components from the library is given in Section VIII. A discussion of related work and conclusions round out the paper.

II. Overview of the C2 Architectural Style

C2 is an architectural style designed to support the particular needs of applications that have a graphical user interface aspect. The style supports a paradigm in which UI components, such as dialogs, structured graphics models (of various levels of abstraction), and, as this paper will show, 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 multiple media types may be involved.

1. Effort sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-94-C-0218 and F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

II.A. Style Rules

The C2 style can be informally summarized as a network of concurrent components hooked together by connectors, i.e., message routing devices. 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 and the bottom of a component may be connected to the top of a single connector. No direct component-to-component links are allowed. There is no bound on the number of components or connectors that may be attached to a single connector. When two connectors are attached to each other, it must be from the bottom of one to the top of the other (see Fig. 1).

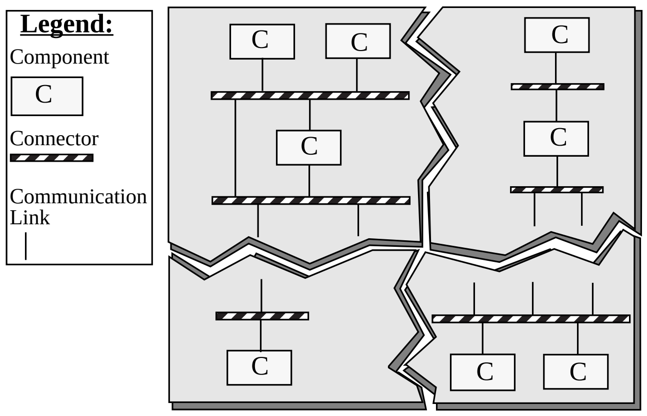


Fig. 1. A sample C2 architecture. Jagged lines represent the parts of the architecture not shown.

Each component has a top and bottom domain. The top domain specifies the set of notifications to which a component responds, and the set of requests that the 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. All communication between components is achieved by exchanging messages. This requirement is suggested by the asynchronous nature of component-based architectures, and, in particular, 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.

Central to the architectural style is a principle of limited visibility or *substrate independence*: a component within the hierarchy can only be aware of components "above" it and is 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 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.

Substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. One issue that must be addressed, however, is the apparent dependence of a given component on its "substrate," 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 the given architecture, its reusability value is greatly diminished and it can only be substituted by components with similarly constrained top domains. For that reason, the C2 style introduces the notion of event 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 it understands [39], [43]. The C2 design and development tools [20], [30] are intended to provide support for accomplishing this task.

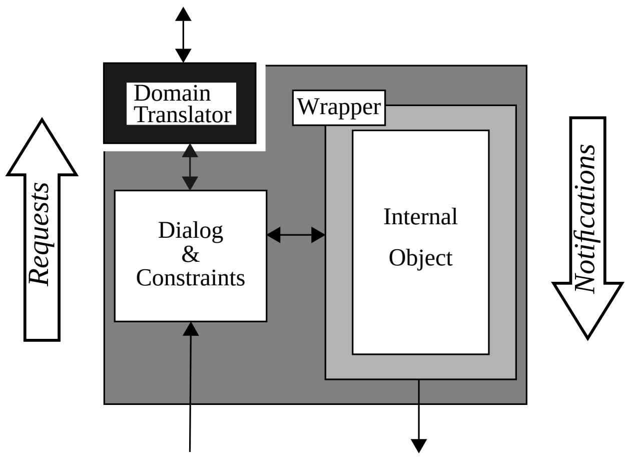


Fig. 2. The Internal Architecture of a C2 Component.

The internal architecture of a component shown in Fig. 2 is targeted to the user interface domain. While issues concerning composition of an architecture are independent of a component's internal structure, for purposes of exposition below, this internal architecture is assumed. C2 also supports compositionality, or hierarchical composition, where an entire architecture becomes a single component in another, larger architecture.

Each component may have its own thread(s) of control, a property also suggested by the asynchronous nature of tasks in the GUI domain. It simplifies modeling and programming of multi-component, multi-user, and concurrent applications and enables exploitation of distributed platforms. A proposed conceptual architecture is distinct from an implementation architecture, so that it is indeed possible for components to share threads of control.

Finally, there is no assumption of a shared address space among components. Any premise of a shared address space would be unreasonable in an architectural style that allows composition of heterogeneous, highly distributed components, developed in different languages, with their own threads of control, internal structures, and domains of discourse.

II.B. Influences

The C2 work has drawn inspiration from many sources:

- layered systems,
- implicit invocation,
- operating system (OS) process support,

- component interoperability models, and
- software architectures.

While C2 has many similarities to existing work in these areas, there are also important differences that distinguish it from them.

In contrast to existing systems, such as Field [29] and SoftBench [6], X Windows [33], Chiron-1 [40], Arch [28], and Slinky [44], which support only a fixed number of *layers* in an architecture, the C2 architectural style allows layering to vary naturally with the application domain. In this, the C2 style is similar to GenVoca [2], whose components may be composed in a number of layers that naturally reflects the characteristics of a particular domain. Unlike GenVoca, which uses explicit invocation, C2 provides a layering mechanism based on implicit invocation. This allows the C2 style to provide greater flexibility in achieving substrate independence in an environment of dynamic, multilingual components: component recompilation and relinking can be avoided and on-the-fly component replacement enabled [24].

In C2, *implicit invocation* occurs when a component invokes its internal object in reaction to a notification. The invocation is implicit because a component issuing notifications does not know if those notifications will cause any reaction, nor does it explicitly name an entry point into a component below it. The benefits of implicit invocation are described in the context of mediators by Sullivan and Notkin [36], [37]. While many systems, such as Chiron-1 and VisualWorks [26][2], employ implicit invocation for its benefits in minimizing module interdependencies, the C2 style also provides a discipline for ordering components which use implicit invocation, yielding substrate independence.

Another example of implicit invocation is the Weaves system [11], in which concurrently executing tool fragments communicate by passing (pointers to) objects. This passing of objects causes Weaves to be used in a data flow manner. Weaves allows data moving between output and input portals of connected tool fragments. Unlike C2, which allows messages to flow in both directions along a communication link, data flow in weaves is unidirectional ("left to right"); flow in the other direction is achieved by explicitly creating communication cycles. Similarly to C2, communication in Weaves occurs via connectors (transport services). Furthermore, the granularity of tool fragments is on the order of a single procedure. This, coupled with unidirectional data flow and communication cycles, can result in Weave architectures consisting of large numbers of tool fragments and transport services with a complicated interconnection structure.

Existing systems tend to be rigid in terms of mapping their components to *OS processes*. 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.

2. VisualWorks is a Smalltalk GUI library based on the Model-View-Controller paradigm [12], where the model broadcasts change of state notifications to views and controllers.

Existing *component interoperability models*, such as OLE [5] and OpenDoc [25], provide standard formats 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.

Perhaps the greatest influence on C2 has been the growing body of research on *software architectures*: abstractions for representing high-level structure of (potentially large) systems, languages for modeling those abstractions, and tools to support specification, analysis, and implementation of architectures. Examples of relevant approaches are Rapide [14], Wright [1], UniCon [34], MetaH [42], and Darwin [15]. C2 in particular shares the basic vocabulary with architecture description languages (ADLs) like Wright and UniCon, which explicitly focus on software connectors as first-class entities in describing architectures. Recognizing that connectors play a primary role in software architectures enables the separation of computation form communication and thus fosters system reconfigurability.

It is this explicit treatment of connectors that, for example, directly distinguishes C2 from more traditional layered systems, such as network systems (e.g., Avoca [2]) and operating systems. Connectors provide a level of indirection that reduces dependencies among computational elements (components). Coupled with implicit invocation and domain translation, this indirection gives developers more flexibility in building systems out of (existing) components whose interfaces do not match perfectly, and enhancing such systems incrementally as additional (needed) functionality becomes available. For example, a C2 connector can decide to route some of the requests that were initially handled (and possibly ignored) by component X to the new component Y, which can process them faster and/or provide higher-precision results. From the stand point of components which are below them in a C2 architecture, X and Y comprise a single component, as illustrated in Fig. 3.

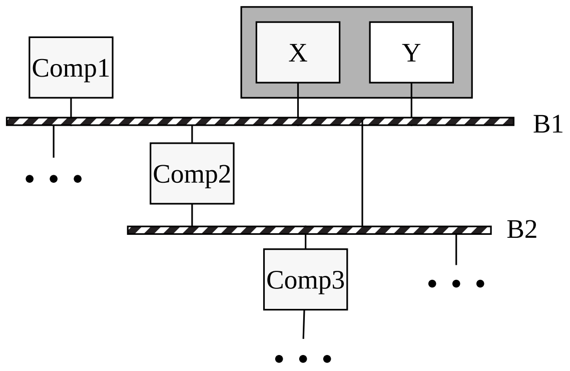


Fig. 3. An example (partial) architecture built according to C2 style rules. The architecture demonstrates C2's support for reconfigurability: component Y has been added to an existing architecture and connector B1 routs to it some of the requests that used to be delivered to component X. None of the components below B1 (e.g., Comp2 and Comp3) need to be updated in any manner, as they are effectively still communicating with a single component.

III. Implementing C2-Style Architectures

The ultimate goal of any software design and modeling endeavor is to produce the executable system. An elegant and effective architectural model is of limited value unless it can be converted into a running application. Doing so manually may result in many problems of consistency and traceability between an architecture and its implementation. For example, it may be difficult to guarantee or demonstrate that a given system correctly implements an architecture. Furthermore, even if this is currently the case, one has no means of ensuring that future changes to the system are appropriately traced back to the architecture and vice versa. It is, therefore, desirable, if not imperative, for architecture-based software development approaches to provide source code generation tools.

```
C2Object
   ├──C2Message
   │     ├──C2Request
   │     └──C2Notification
   ├──C2Port
   │     └──C2Port_FIFO
   └──C2Brick
         ├──C2Connector
         │     ├──C2Connector_SameProcess
         │     ├──C2Connector_Thread
         │     └──C2Connector_IPC
         └──C2Component
               ├──C2Architecture
               └──C2Component_Threads
                     └──C2Architecture_Threads
```

Fig. 4. C2 object-oriented class framework.

To support implementation of C2 architectures, we developed an extensible framework of abstract classes for C2 concepts such as components, connectors, and messages, shown in Fig. 4. This framework is the basis of development and OTS component reuse in C2. It implements component interconnection and message passing protocols. Components and connectors used in C2 applications are subclassed from the appropriate abstract classes in the framework. This guarantees their interoperability, eliminates many repetitive programming tasks, and allows developers of C2 applications to focus on application-level issues. In order to incorporate OTS components into a C2 architecture, they are wrapped inside framework components, as shown in Fig. 5. The framework supports a variety of implementation configurations for a given architecture: the entire resulting system may execute in a single thread of control, or each component may run in its own thread of control or operating system (OS) process.

The framework has been implemented in C++ and Java;[3] a subset is also available in Ada. The size of the framework in both C++ and Java is approximately 3000 commented source lines of code. We have been able to successfully reuse the Q interprocess communication (IPC) library [16] to enable message exchange between C2 components implemented in C++ and Ada. Similar functionality for Java C2 components is under development.

3. The C++ and Java frameworks and several simple applications developed with them are available at http://www.ics.uci.edu/pub/arch/.

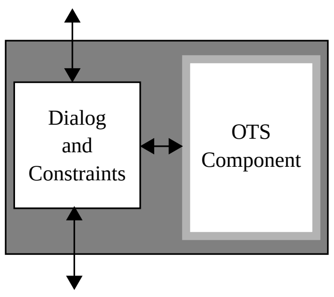


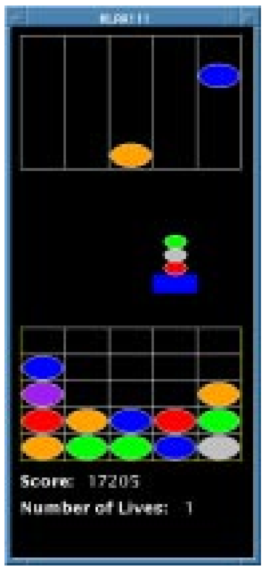
Fig. 5. An OTS component is wrapped inside a C2 component.

We have also used the C++ and Java frameworks to integrate the Xlib [33] and Java AWT [7] user interface toolkits, respectively. By essentially wrapping them in the manner depicted in Fig. 5, they become C2 “graphics binding” components. Such graphics bindings are needed since many potential C2 components, such as these commercial 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 into C2 requests. 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 graphics bindings such that they contain the required functionality and are reusable across architectures and applications. In integrating Xlib and AWT, we drew from our experience in adapting Motif and OpenLook for use in Chiron-1 [40].

Our current approach to implementing C2 architectures is *implementation constraining* [21]: there is a 1-to-1 relationship between the components in the architectural model and those in its corresponding implementation. It is important to note that this is a property of our current toolset, and *not* of C2. C2 allows arbitrary mappings of conceptual to concrete components [19]. At the same time, while limiting in certain regards (e.g., the performance of systems implemented in this manner will not always be adequate), this approach has enabled us to produce implementation prototypes of C2 architectures quickly and reliably. This has, in turn, allowed us to focus our attention on other facets of C2, such as dynamic change of C2 architectures [24], expansion to domains other than GUI software [20], and C2’s suitability and support for OTS component integration and development of application families, discussed below.

IV. Overview of KLAX

The architecture that was used as the basis for our investigation of OTS component integration in C2 is a version of the video game KLAX. A description of the game is given in Fig. 6. 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 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.

KLAX Status

Fig. 6. A screenshot and description of our implementation of the KLAX video game.

The architecture of the system is depicted in Fig. 7. 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 components are placed at the top since game state is vital for the functioning of the other two groups of components. The game state components receive no notifications, but respond to requests and emit notifications of internal state changes. Notifications are directed to the next level, where they are received by both the game logic components and the artist components.[4]

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

The artist components also receive notifications of game 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 the hope that a lower-level graphics component will render them on the screen. *TileArtist* provides a flexible presentation for tiles. Artists maintain information about the placement of abstract tile objects. *TileArtist* intercepts any notifications about tile objects and recasts them to notifications about more concrete drawable objects. For example, a "Tile-Created" notification might be translated into a "Rectangle-Created" notification. The *LayoutManager* component receives all notifications from the artists and offsets any coordinates to ensure that the game elements are drawn in the correct two-dimensional juxtaposition.

The *GraphicsBinding* 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.

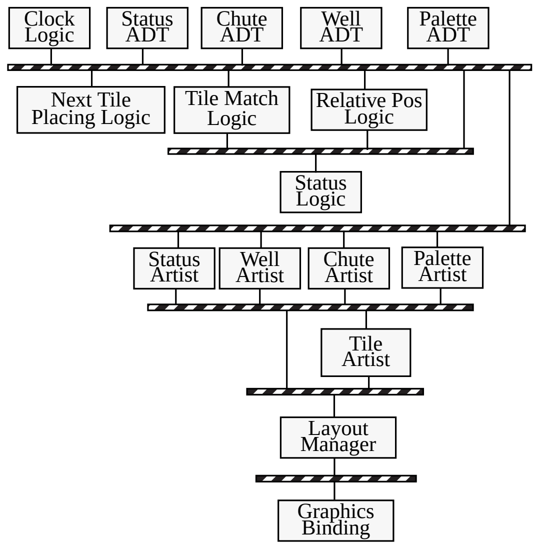


Fig. 7. Conceptual C2 architecture for KLAX.

The KLAX architecture is intended to support a family of "falling-object" games. The components were designed as reusable building blocks to support different game variations. One such variation is described in [39].

The KLAX implementation is built using the C++ framework. The implementation consists of approximately 8100 lines of commented C++ code, in addition to the base framework's 3000 lines of code. Performance of the implementations is good on a Sun Sparc2 workstation, easily exceeding human reaction time if the *ClockLogic* component is set to use short time intervals. Although we have not yet tried to optimize performance, benchmarks indicate that 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 causes 10 to 30 message sends, and a tick of the clock typically causes 3 to 20.

V. KLAX Constraints

In its form as described above, KLAX does not necessarily need a constraint solver. Its constraint management needs would certainly not exploit the full power of a solver such as SkyBlue, e.g., handling constraint hierarchies [32]. On the other hand, we think it should be possible to use a powerful constraint manager for maintaining a small number of simple constraints. Additionally, the main purpose of this effort was to explore the architectural issues in integrating OTS components into a C2 architecture. We therefore opted not to unnecessarily expend resources to artificially create a situation where a number of complex constraints needed to be managed. Instead, we decided to integrate SkyBlue with KLAX to support its extant constraint management needs. If we were unable to do so, there would be at least three possi-

4. An artist is a component that creates an abstract depiction of the data it receives. That abstract depiction can then be transformed into a screen image by a rendering agent (in our case, C2's graphics binding component).

ble sources of problems: (1) the C2 style, (2) the KLAX architecture, and (3) SkyBlue. In any case, we would learn a useful lesson.

We defined the following 4 constraints for management by SkyBlue:

- *Palette Boundary:* The palette cannot move beyond the chute and well's left and right boundaries.
- *Palette Location:* Palette's coordinates are a function of its location and are updated every time the location changes.[5]
- *Tile Location:* The tiles which are on the palette move with the palette. In other words, the x coordinate of the center of the tile always equals the x coordinate of the center of the palette.
- *Resizing:* Each game element (chute, well, palette, and tiles), is maintained in an abstract coordinate system by its artist. This constraint transforms those abstract coordinate systems, resizing the game elements to have the relative dimensions depicted in Fig. 6 before they are rendered on the screen. This constraint would be essential in a case where the application is composed from preexisting components supplied by different vendors. A similar constraint could also be used to accommodate resizing of the game window, and hence of the game elements within it.

VI. Integrating External Constraint Managers with KLAX

VI.A. Integrating SkyBlue

The four constraints were defined based on the needs of the overall application. Further thought was still needed to decide the location of the constraint manager in the KLAX architecture. There clearly were several possibilities. One solution would have been to include SkyBlue within the appropriate components for the *Palette Boundary*, *Palette Location*, and *Tile Location* constraints, since they affect individual game elements (i.e., they are "local"). The *Resizing* constraint pertains to several game elements, and would thus belong in a separate component.

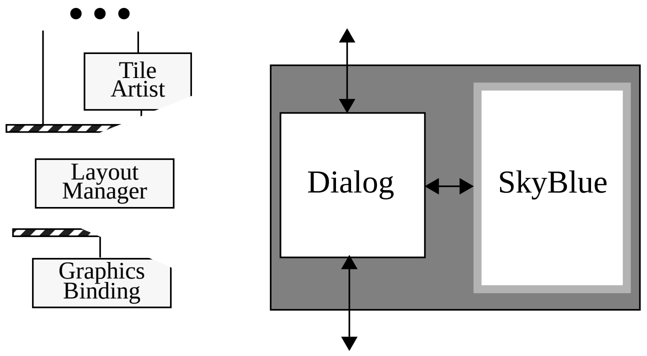


Fig. 8. The SkyBlue constraint management system is incorporated into KLAX by placing it inside the *LayoutManager* component. *LayoutManager*'s dialog handles all the C2 message traffic.

We initially opted for another solution: define all four constraints in a centralized constraint manager component. The *LayoutManager* component was intended to serve as a constraint manager in the original design of KLAX shown in Fig. 7. However, in the initial implementation, the constraints were solved with in-line code locally in *PaletteADT* and *PaletteArtist* and the sole purpose of *LayoutManager* was to properly line up game elements on the screen. The implemented version of *LayoutManager* also placed the burden of ensuring that the game elements have the same relative dimensions on the developers of the *PaletteArtist*, *ChuteArtist*, and *WellArtist* components. Incorporating constraint management functionality into *LayoutManager* therefore rendered an implementation more faithful to its original design.

The constraints were defined in the "dialog and constraints" part of the *LayoutManager* component (see Fig. 2), while SkyBlue became the component's internal object. As such, SkyBlue has no knowledge of the architecture of which it is now a part. It maintains the constraints, while all the request and notification traffic is handled by *LayoutManager*'s dialog, as shown in Fig. 8. *LayoutManager* thus became a constraint management component in the C2 style that can be reused in other applications by only modifying its dialog to include new constraints.[6]

PaletteADT, *PaletteArtist*, *ChuteArtist*, and *WellArtist* also needed to be modified. Their local constraint management code was removed. Furthermore, their dialogs and message interfaces were expanded to notify *LayoutManager* of changes in constraint variables and to handle requests from *LayoutManager* to update them.

It is important to note that it was not necessary to modify these four components in order for the architecture containing the new *LayoutManager* to behave correctly. However, just like the original *LayoutManager* was modified to reflect its intent, these components' implementations were modified to mirror their intended behavior as well. As already discussed in the preceding section, building this new version of *LayoutManager* and inserting it into the architecture was not motivated by the need for functionality that did not already exist in the architecture (the application had already behaved as desired). Rather, the drivers were improved traceability of architectural decisions to the implementation and vice versa, construction of a powerful constraint management component in the C2 style, and investigation of issues in integrating OTS components into C2-style architectures. This matter is further discussed below in Section VIII.B.

11 new messages were added to handle this modification of the original application and there was no perceptible performance degradation. The entire exercise was completed by one developer in approximately 45 hours.

VI.B. Integrating Amulet

C2 supports reuse through component-based development, substrate independence, and domain translation. These features also support component substitutability and localization of change. We claim that, in general, two behaviorally equivalent components can always be substituted for one another and that behavior preserving modifications to a component's implementation have no architecture-wide

5. Location is an integer between 1 and 5.

6. In the remainder of the paper, when we state that a constraint solver is "inside" or "internal to" a component, the internal architecture of the component will resemble that of *LayoutManager* from Fig. 8.

effects [19].

In the example discussed in the previous section, this would mean that SkyBlue may be replaced with another constraint manager by only having to modify the "dialog and constraints" portion of *LayoutManager* to define constraints as required by the new solver. The set of messages in *LayoutManager*'s interface and the rest of the KLAX architecture would remain unchanged.

To demonstrate this claim, we substituted SkyBlue with Amulet's one-way formula constraint solver [17]. This exercise required identifying, extracting, and recompiling the needed portion of Amulet, a task that was accomplished by a single developer in approximately 25 hours.[7] This added effort was necessitated by our inability to locate implementations of any other constraint solvers. It resulted in a situation that is common when attempting software reuse: OTS systems may not contain components that can be clearly identified or easily isolated and extracted [4], [9], [13].

Once the solver was extracted from the rest of Amulet, it was successfully substituted for SkyBlue in the KLAX architecture and tested by one developer in 75 minutes. As anticipated, no architecture-wide changes were needed. Only the interior of the *LayoutManager* component needed to be modified: its internal object was now Amulet instead of SkyBlue; the constraint variables updated by the component's dialog in response to incoming C2 messages were now defined in Amulet. The look-and-feel of the application remained unchanged. There was again no performance degradation.

VII. KLAX Component Library

Integrating SkyBlue and Amulet with KLAX provided an opportunity for building multiple versions of *PaletteADT*, *PaletteArtist*, *ChuteArtist*, *WellArtist*, and *LayoutManager* components. Individual versions of each component would differ based on two criteria:

- constraints maintained — if two versions of a component maintain different constraints internally, their message interfaces will also differ to account for that. Extreme cases are (1) components that enforce all of their local constraints and (2) those that enforce no constraints.
- mechanism used for constraint maintenance — a component can maintain a constraint (1) with in-line code, as in the original implementation, (2) in SkyBlue, (3) in Amulet, or (4) using a combination of the three.

The two integrations described in the previous section resulted in three versions of *LayoutManager*: the original, SkyBlue, and Amulet versions. These are listed as *LayoutManager* versions 1, 2, and 3 in Table 1. Two versions each of *PaletteADT*, *PaletteArtist*, *ChuteArtist*, and *WellArtist* were created as well: original components maintaining local constraints with in-line code (versions 1 of the four components in Table 1) and components whose constraints were managed elsewhere in the architecture (versions 2 of the four components in Table 1).[8]

Table 1: Implemented Versions of *PaletteADT, PaletteArtist, ChuteArtist, WellArtist*, and *LayoutManager* KLAX Components

	Version Number	Constraints Maintained	Constraint Managers
Palette ADT	1	Palette Boundary	In-Line Code
	2	None	None
	3	Palette Boundary	SkyBlue
	4	Palette Boundary	Amulet
Palette Artist	1	Palette Location Tile Location Tile Size	In-Line Code
	2	None	None
	3	Palette Location Tile Location	SkyBlue
	4	Palette Location Tile Location	Amulet
Chute Artist	1	Chute Size	In-Line Code
	2	None	None
Well Artist	1	Well Size	In-Line Code
	2	None	None
Layout Manager	1	None	None
	2	All	SkyBlue
	3	All	Amulet
	4	Resizing	SkyBlue
	5	Resizing	Amulet
	6	All	SkyBlue & Amulet

The two initial integrations also suggested other variations of these components, such as replacing in-line constraint management code with SkyBlue and Amulet constraints in *PaletteADT* and *PaletteArtist* (see Footnote 6). Also, a version of *LayoutManager* was implemented that maintained only the *Resizing* constraint, in anticipation that other components will internally manage their local constraints (this scenario was briefly described at the beginning of Section VI). This resulted in a total of 18 implemented versions of the five components, as depicted in Table 1.

VIII. Building a Program Family

The four versions of *PaletteADT* and *PaletteArtist*, two versions of *ChuteArtist* and *WellArtist*, and six versions of *LayoutManager*, described in Table 1, could potentially be used to build 384 different variations of the KLAX architecture, i.e., members of the KLAX application family. Three such variations were described in Section IV (using versions 1 of all five components), Section VI.A (using versions 2 of the five components), and Section VI.B (replacing *LayoutManager*-2 with *LayoutManager*-3 in the architecture from Section VI). In this section, we discuss several additional implemented variations of the architecture that exhibit interesting properties.

VIII.A. Multiple Instances of a Constraint Manager

In the architecture depicted in Table 2, the *Palette Boundary*, *Palette Location*, and *Tile Location* constraints are defined and maintained in SkyBlue inside *PaletteADT* and

7. For the purpose of brevity, in the remainder of the paper Amulet's one-way formula constraint manager will be referred to simply as "Amulet."

8. In the rest of the paper, a particular component version will be depicted by the component name followed by version number (e.g., *PaletteADT*-2).

PaletteArtist, while the *Resizing* constraints are maintained globally by *LayoutManager*. Therefore, multiple instances of SkyBlue maintain the constraints in different KLAX components. Since C2 separates architecture from implementation, we were able to implement the three components that contain their own logical copies of SkyBlue using a single physical instance of the constraint manager.

Table 2: Multiple Instances of SkyBlue

Component	Version Number	Constraints Maintained	Constraint Managers
PaletteADT	3	Palette Boundary	SkyBlue
PaletteArtist	3	Palette Location Tile Location	SkyBlue
ChuteArtist	2	None	None
WellArtist	2	None	None
LayoutManager	4	Resizing	SkyBlue

VIII.B. Partial Communication and Service Utilization

Particularly interesting are components that are used in an architecture for which they have not been specifically designed, i.e., they can do more or less than they are asked to do. This is an issue of reuse: if we build components a certain way, are their users (designers) always obliged to use them “fully”; furthermore, can meaningful work be done in an architecture if two components communicate only partially, i.e., certain messages are lost? The architectures described below represent a crossection of exercises conducted to better our understanding of partial communication and partial component service utilization.

- A variation of the original architecture was implemented by substituting *LayoutManager*-2 into the original architecture, as shown in Table 3. *LayoutManager*-2’s functionality remains largely unused as no notifications are sent to it to maintain the constraints (see Section VI.A). The application still behaves as expected and there is no performance penalty. Note that this will not always be the case: if *LayoutManager*-2 was substantially larger than *LayoutManager*-1 or had much greater system resource needs (e.g., its own operating system process), the performance would be affected.

Table 3: None of *LayoutManager*’s Constraint Management Functionality is Utilized

Component	Version Number	Constraints Maintained	Constraint Managers
PaletteADT	1	Palette Boundary	In-Line Code
PaletteArtist	1	Palette Location Tile Location Palette Size	In-Line Code
ChuteArtist	1	Chute Size	In-Line Code
WellArtist	1	Well Size	In-Line Code
LayoutManager	2	All	SkyBlue

- Another architecture that was built is shown in Table 4. This exercise was intended to explore heterogeneous approaches to constraint maintenance in a single architecture: some components in the architecture maintain their constraints with in-line code (*WellArtist* and *ChuteArtist*), others maintain them internally using SkyBlue (*PaletteADT*), while *PaletteArtist*’s constraints are maintained by an external constraint manager. *LayoutManager*-2 is still partially utilized, but a larger subset of its services is used than in the preceding architecture.

Table 4: *LayoutManager*’s Constraint Management Functionality is Only Partially Utilized

Component	Version Number	Constraints Maintained	Constraint Managers
PaletteADT	3	Palette Boundary	SkyBlue
PaletteArtist	2	None	None
ChuteArtist	1	Chute Size	In-Line Code
WellArtist	1	Well Size	In-Line Code
LayoutManager	2	All	SkyBlue

- In the architecture shown in Table 5, *PaletteADT* expects that the *Palette Boundary* constraint will be maintained externally by some other component. However, in this case, *LayoutManager*-1 does not understand and therefore ignores the notifications sent by *PaletteADT* (partial communication). Movement of the palette is thereby not constrained and the application behaves erroneously: the palette disappears when moved beyond its right boundary; the execution aborts when the palette moves beyond the left boundary and the *GraphicsBinding* component (see Section IV) attempts to render it at negative screen coordinates.

Table 5: *Palette Boundary* Constraint is not Maintained

Component	Version Number	Constraints Maintained	Constraint Managers
PaletteADT	2	None	None
PaletteArtist	1	Palette Location Tile Location Palette Size	In-Line Code
ChuteArtist	1	Chute Size	In-Line Code
WellArtist	1	Well Size	In-Line Code
LayoutManager	1	None	None

The above examples seem to imply that partial service utilization generally has no ill effects on a system, while partial communication does. This is not always the case. For example, an additional version of each component from the original architecture was built to enable testing of the application. These components would generate notifications that were needed by both components below them in the architecture and the testing harness. If a “testing” component was inserted into the original architecture, all of its testing-related messages would be ignored by components below it, resulting in partial communication, yet the application would still behave as expected. Clearly, the overhead of dispatching messages that ultimately get ignored may be prohibitively expensive in certain situations. In general, a useful metric for determining the potentially negative effects of partial communication would be the ratio of the number of lost messages to the total number of messages in an architecture.

VIII.C. Multiple Constraint Managers in a Single Component

LayoutManager-6 had some of its constraints defined in SkyBlue and others in Amulet. Combining multiple constraint solvers in a single *system* has only recently been identified as a potentially useful approach to constraint management [17], [32]. Integrating multiple constraint solvers in a single C2 *component* is certainly at a different level of granularity. However, this exercise sensitized us to several issues intrinsic to the interaction of heterogeneous constraint managers.

Specifying constraints in different solvers over disjoint sets of variables is a trivial task, since there are no dependencies between the solvers. On the other hand, if the two sets of constraint variables intersect, the problem is more complex. In our case, constraint variables in SkyBlue and Amulet are of different types, so that the same variable cannot be used in constraints specified in both solvers. Therefore, each conceptually common variable is implemented by two actual variables (*var_SkyBlue* and *var_Amulet*). Furthermore, additional functionality is needed to monitor the changes in the variables and programmatically update one when the other is changed due to constraint enforcement (see Fig. 9).

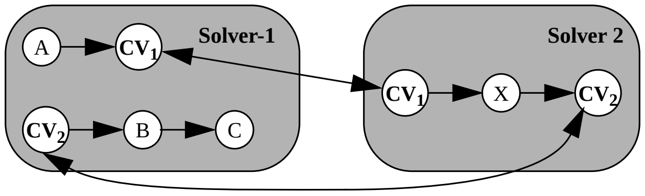


Fig. 9. To provide consistent constraint maintenance across constraint solvers, each conceptually common constraint variable (CV) is implemented with two actual variables. Changes in one are automatically reflected in the other.

For example, in *LayoutManager*-6, *Palette Boundary*, *Tile Location*, and *Resizing* constraints are defined in SkyBlue, while *Palette Location* is specified in Amulet. Every time *location_SkyBlue* changes, its new value is assigned to *location_Amulet* so that Amulet can properly update the *paletteX_Amulet* variable. To propagate its change through the rest of SkyBlue variables, *paletteX_Amulet*'s new value is copied into *paletteX_SkyBlue*.

Our solution to defining SkyBlue and Amulet constraints over overlapping sets of variables, although effective, was not particularly elegant. It had the feel of programming one's own application-specific constraint management functionality. While the purpose of the exercise was to investigate issues pertinent to software architectures and application families, this problem has broader ramifications. A scenario where both a powerful but complex solver and a simple one are needed in an application is likely. Therefore, we consider the problem of multiple interacting constraint managers an open research issue that requires careful examination. We are currently exploring what role an architectural style such as C2, and particularly its support for compositionality, may play in the resolution of this problem.

VIII.D. Multiple Constraint Managers in an Architecture

An issue related to using multiple constraint managers inside a single component is using multiple constraint managers in different components, but in a single architecture. Such an architecture was built using components shown in Table 6. In this architecture, *Palette Boundary* and *Resizing* constraints are maintained by SkyBlue, and *Palette Location* and *Tile Location* by Amulet. Since the sets of constraint variables managed by the two solvers are disjoint, there are no interdependencies of the kind discussed in the previous example between SkyBlue and Amulet.[9] Hence, this modification to the architecture was a simple one.

Table 6: Multiple Constraint Solvers

Component	Version Number	Constraints Maintained	Constraint Managers
PaletteADT	3	Palette Boundary	SkyBlue
PaletteArtist	4	Palette Location Tile Location	Amulet
ChuteArtist	2	None	None
WellArtist	2	None	None
LayoutManager	4	Resizing	SkyBlue

IX. Related Approaches

Explicit focus on software architectures, and architectural styles in particular have a great potential to facilitate both OTS component reuse and development of families of applications. Krueger points out some common problems in basing reuse on software architectures [13]. His criticism mainly applies to those approaches that do not identify higher-level abstractions applicable across applications. C2, on the other hand, is a style that attempts to exploit commonalities across systems, and reuse individual components as well as successful structural and communication patterns.

The two goals of maximizing reuse and building system families do not always go hand in hand. For example, one focus of domain-specific software architectures (DSSAs) is on developing a generic *reference architecture* for all systems in a particular domain of applicability [41]. The reference architecture is then instantiated for every individual system within the domain, as shown in Fig. 10. By making reference architectures explicit, DSSAs employ a systematic approach to developing application families.

In the work we discussed in this paper we do not develop a reference architecture as a basis for building the KLAX application family. However, there is also nothing inherent in the C2 style that prohibits one from doing so. Quite the contrary, as an architectural style, C2 can be applied to multiple domains, each of which would require its own reference architecture. C2's ADL and underlying formalism [18], [19], [22] are well suited for this: each component in a C2 architecture is a conceptual placeholder which can be instantiated by different implemented modules. The ease with which we were able to build the application family described in this paper without the aid of a reference architecture is indicative of C2's potential in this regard.

Unlike their inherent support for application families, DSSAs have tended to support reuse only to a limited degree. GenVoca [2] is an illustrative example. It has been particularly successful in producing a large library of reusable components. However, those components have been custom built for the GenVoca style. In order to reuse them, one must adhere to GenVoca's formalism and its hierarchical approach to component composition, which may result

9. Architectures built according to the C2 style will always have this property: since C2 does not assume a single address space for its components, inter-component constraint variable sets will always be disjoint.

in a high degree of dependency between communicating components. On the other hand, C2's style rules and underlying formalism are more flexible; C2 eliminates assumptions of shared address spaces and threads of control, allows both synchronous and asynchronous message-based communication, and separates the architecture from the implementation.

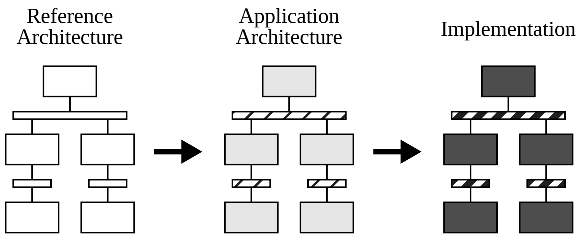


Fig. 10. A simplified, high-level view of the DSSA lifecycle. A generic reference architecture is instantiated to obtain an application-specific architecture, which is then used as the basis for implementation.

Several aspects of object-oriented (OO) programming have provided us with valuable lessons. In [19] we demonstrate how concepts from OO typing can be applied to software architectures. The work on OO design patterns [8] has similarities with architectural styles. However, OO design patterns support reuse of structures at a much lower level of abstraction than do styles.

Garlan, Allen, and Ockerbloom classify the causes of problems developers commonly experience when attempting OTS reuse and give four guidelines for alleviating them [9]. Our experience shows that C2 is well suited to address these problems. The first two guidelines deal with the internal architecture of OTS components, and are thus outside the scope of C2. The third guideline proposes techniques for building component adaptors, which is subsumed by C2 wrappers and domain translators. Finally, the authors emphasize the need for design guidance, which is a significant aspect of our approach to C2 [30], [31].

Shaw discusses nine "tricks" for reconciling component mismatch in an architecture [35]. Several of the tricks are related to reuse techniques employed in C2. For example, transformations, such as "Change A's form to B's form", "Provide B with import/export converters", and "Attach an adapter or wrapper to A," are subsumed by C2's wrappers and/or domain translators. The need for other transformations is eliminated altogether by C2 style rules. For example, "Make B multilingual" is unnecessary, as C2 assumes that components will be heterogeneous and multilingual.

X. Conclusion

The full potential of component-based software architectural styles cannot be realized unless reusing code developed by others becomes a common practice. A new architectural style can become a standard in its domain only if it makes reuse easier. We believe that C2 is such a style for GUI software, with the potential for broader applicability.

Several characteristics of the C2 style have enabled it to better support reuse of OTS components and construction of different members of an application family from existing parts. Although most of these characteristics are not unique to C2, our approach of combining them is. We believe the style rules are restrictive enough to make reuse easier while flexible enough to integrate components built outside the style and connect them in an architecture in various ways ("plug and play"):

- *Component heterogeneity* — No restrictions are placed on the implementation language or granularity of the components.
- *Substrate independence* — A component does not depend on the existence of components below it.
- *Internal component architecture* — The internal architecture of a C2 component separates communication from processing. The *dialog* isolates the *internal object* from changes in the rest of the architecture. The *domain translator* reduces the dependence of a component on components above it; a component can use different domain translators in different architectures.
- *Asynchronous message passing via connectors* — Components communicate only by exchanging asynchronous messages through connectors. This has the potential to greatly simplify control integration issues, such as those encountered by Garlan and colleagues in [9].[10] Coupled with partial communication and service utilization, this property also facilitates low-cost interchangeability of components to construct different members of the same family.
- *No assumption of shared address space* — Two components cannot assume that they will execute in the same address space. This eliminates complex dependencies, such as components sharing global variables, and simplifies modification of architectures.
- *No assumption of single thread of control* — Conceptually, components run in their own thread(s) of control. This allows components with potentially different threading models to be integrated into a single application.
- *Separation of architecture from implementation* - A conceptual C2 architecture can be instantiated in a number of different ways. Many potential performance issues or variations in functionality can be addressed by separating the architecture from actual implementation techniques. We have found that we can isolate some implementation decisions in the C2 framework, discussed in Section III. For example, a connector between two components in the same address space can use direct procedure calls to implement message passing.

The exercises discussed in Sections VI-VIII, as well as recent work described in [20], have enabled us to devise an initial set of heuristics for OTS component integration in C2. The only assumption we make is that OTS components provide application programmable interfaces (APIs). As our experience with reusing OTS components grows, we expect that this list will be expanded and refined:

- If the OTS component does not contain all of the needed functionality, its source code must be altered or a custom-built component must be supplied. In general, this is a difficult task, whose complexity is well recognized [9], [13], [23].[11]

10. While the style does not forbid synchronous communication, the responsibility for implementing synchronous message passing resides with individual components.

- If the OTS component does not communicate via messages, a C2 wrapper must be built for it. This was the case with both SkyBlue and Amulet.
- If the OTS component is implemented in a programming language different from that of other components in the architecture, an inter-process (IPC) connector must be employed to enable their communication. A number of existing systems provide this capability. For example, we were able to accomplish this task for C++ and Ada components relatively easily using the Q software bus [16].
- If the OTS component must execute in its own thread of control, an inter-thread connector must be employed. This was accomplished in the case of the Java AWT graphics toolkit.
- If the OTS component executes in its own process, an IPC connector must again be employed.
- If the OTS component communicates via messages, but its interface does not match interfaces of components with which it is to communicate, a domain translator must be built for it. Although we have done some preliminary work on domain translation in our Java class framework, this area needs further exploration.

The information above is summarized in Table 7.

Table 7: OTS Component Integration Heuristics for C2

Problem with OTS Component	Integration Method
Inadequate Functionality	Source Code Modification
No Message-Based Communication	Wrapper
Different Programming Language	IPC Connector
Different Thread of Control	Inter-Thread Connector
Different OS Process	IPC Connector
Message Interface Mismatch	Domain Translator

The series of exercises described in this paper demonstrate that C2 isolates changes inside components and limits any global effects of those changes through message-based communication. Furthermore, C2's principles of substrate independence and domain translation enable component substitutability. Finally, its component- and message-based nature allows partial communication and service utilization of components, which are essential to cost-effective reuse.

In a component-based style, such as C2, the number of possible architectures grows combinatorially as the number of behaviorally related components increases. Thus, the 18 components depicted in Table 1 can generate 384 distinct versions of KLAX. Of course, every possible architecture is not necessarily meaningful (e.g., the example of partial communication in Section VIII.B) nor particularly interesting. It is the responsibility of the architect to ensure that each constraint is properly handled somewhere in the architecture. This task can be aided by a design guidance tool, such as C2's Argo design environment [30], [31]. What we have shown is that C2 gives the architect added flexibility as to exactly where and how to include the constraint handling behavior.

11. Note that the component can still be reused "as is" if the developers are willing to risk degraded or incorrect performance, due to partial communication and partial component service utilization in the architecture. This was the case with several variations of KLAX, discussed in Section VIII.

This exercise demonstrated the potential for creating a library of components and an application family in the C2 style. In addition, we now have a constraint management component in the C2 style that will be reused across future applications.

XI. Acknowledgements

We would like to acknowledge the members of the C2 research group for their contribution on various aspects of C2. We also wish to thank the developers of SkyBlue and Amulet for providing the source code to their systems and adequate documentation to ease our usage of them.

XII. References

[1] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71-80, Sorrento, Italy, May 1994.

[2] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, volume 1, number 4, pages 355–398, October 1992.

[3] T. J. Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. In *Proceedings of the Third International Conference on Software Reuse*, pages 102-109, Rio de Janeiro, Brazil, November 1994.

[4] T. J. Biggerstaff and A. J. Perlis. *Software Reusability*, volumes I and II. ACM Press/Addison Wesley, 1989.

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

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

[7] P. Chan and R. Lee. *The Java Class Libraries: An Annotated Reference*. Addison-Wesley, 1996.

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

[9] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why It's Hard to Build Systems out of Existing Parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179-185, Seattle, WA, April 1995.

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

[11] 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.

[12] 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*, volume 1, number 3, pages 26–49, August/September 1988.

[13] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, volume 24, number 2, pages 131-183, June 1992.

[14] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, volume 21, number 9, pages 717-734, September 1995.

[15] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 3-14, San Francisco, CA, October 1996.

[16] 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*, pages 451-463, Berlin, Germany, March 1996.

[17] R. McDaniel and B. A. Myers. Amulet's Dynamic and Flexible Prototype-Instance Object and Constraint System in C++. Technical Report, CMU-CS-95-176, Carnegie Mellon University, Pittsburgh, PA, July 1995.

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

[19] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 24-32, San Francisco, CA, October 1996.

[20] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, pages 190-198, Boston, MA, May 17-19, 1997. Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, pages 692-700, Boston, MA, May 17-23, 1997.

[21] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. To appear in *Proceedings of the Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 22-25, 1997.

[22] N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium 1996*, pages 28-40, Los Angeles, CA, April 17, 1996.

[23] R. T. Monroe and D. Garlan. Style-Based Reuse for Software Architecture. In *Proceedings of the Fourth International Conference on Software Reuse*, pages 84-93, Orlando, FL, April 1996.

[24] P. Oreizy. Issues in the Runtime Modification of Software Architectures. Technical Report, UCI-ICS-96-35, University of California, Irvine, August 1996.

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

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

[27] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, volume 17, number 4, pages 40-52, October 1992.

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

[29] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, volume 7, number 4, pages 57–66, July 1990.

[30] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Extending Design Environments to Software Architecture Design. In *Proceedings of the Eleventh Knowledge-Based Software Engineering Conference (KBSE'96)*, pages 63-72, Syracuse, NY, USA, September 1996.

[31] 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.

[32] 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 Ray, CA, November 1994.

[33] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, volume 5, number 2, pages 79-109, April 1986. Actually appeared June 1987.

[34] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, volume 21, number 4, pages 314-335, April 1995.

[35] M. Shaw. Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging. In *Proceedings of IEEE Symposium on Software Reusability*, pages 3-6, Seattle, WA, April 1995.

[36] K. J. Sullivan and D. Notkin. Reconciling Environment Integration and Software Evolution. *ACM Transactions on Software Engineering and Methodology*, volume 1, number 3, pages 229–268, July 1992.

[37] 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.

[38] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins. A Component- and Message-Based Architectural Style for GUI Software. In *Proceedings of the 17th International Conference on Software Engineering (ICSE 17)*, pages 295-304, Seattle, WA, April 1995.

[39] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, volume 22, number 6, pages 390-406, June 1996.

[40] 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*, volume 2, number 2, pages 105–144, June 1995.

[41] W. Tracz. DSSA (Domain-Specific Software Architecture) Pedagogical Example. *ACM SIGSOFT Software Engineering Notes*, volume 2, number 4, pages 49-62, July 1995.

[42] S. Vestal. MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center, April 1996.

[43] D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 176-190, Portland, OR, USA, October 1994.

[44] The UIMS Tool Developers Workshop. A Metamodel for the Runtime Architecture of an Interactive System. *SIGCHI Bulletin*, volume 24, number 1, pages 32–37, January 1992.