Reusing Off-the-Shelf Components to Develop a Family of Applications in the C2 Architectural Style

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 [GS93, PW92]. Typically, styles reflect and leverage key properties of an application domain and recurring patterns of application design within the domain. As such, they 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 [Kru92, Big94, GAO95, Sha95]. 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 [TMA+95, TMA+96], 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 [San94] and Amulet's one-way formula solver [MM95]. 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. The material in this section is condensed from a more detailed exposition on the style, given in [TMA+96]. Section III presents a detailed overview of the architecture and implementation of KLAX, the application used as the basis for our exercises. Section IV 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 V discusses the design and implementation issues encountered in integrating SkyBlue with the architecture, while Section VI discusses replacing SkyBlue with Amulet's constraint manager. 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. Finally, our conclusions round out the paper.

II. Overview of C2

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. 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.

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. 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).

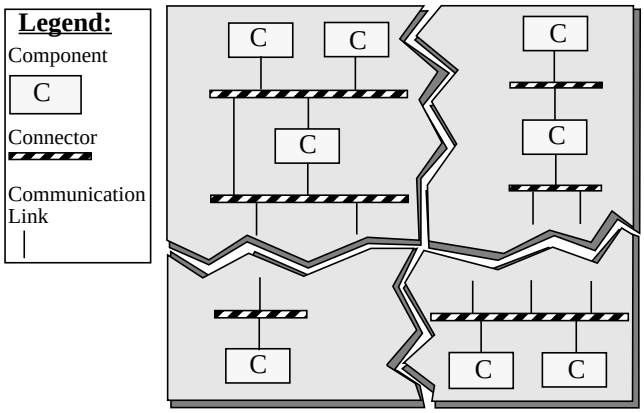

Fig. 1. A sample C2 architecture. Jagged lines represent the parts of the architecture not shown.

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 "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 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. The C2 design environment [RR96] is intended, among other things, to provide support for accomplishing this task.

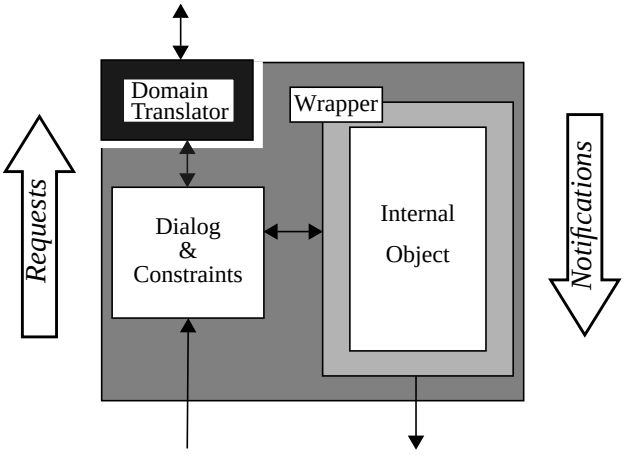

Fig. 2. The Internal Architecture of a C2 Component.

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.

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.

III. Overview of KLAX

The architecture into which SkyBlue and Amulet were integrated is a version of the video game KLAX. A description of the game is given in Fig. 3. 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.

KLAX Status

Fig. 3. A screenshot and description of our implementation of the KLAX video game.

The architecture of the system is depicted in Fig. 4. 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.

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 hope that a lower-level graphics component will render them. *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 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.

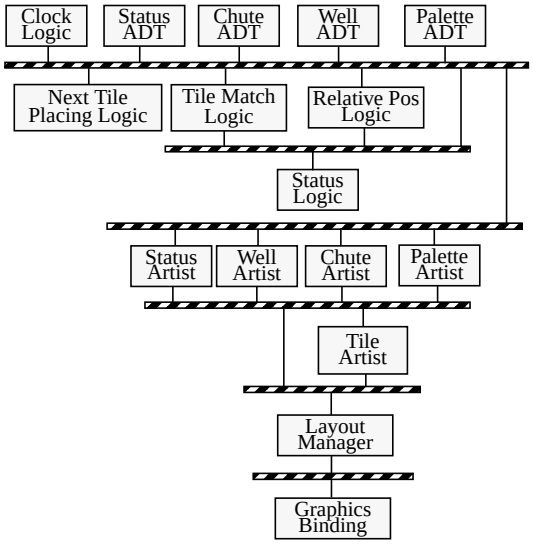

Fig. 4. Conceptual C2 architecture for KLAX.

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 is described in [TMA+96].

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 in [TMA+96], for a single conceptual architecture. This framework is also useful since it allowed us to integrate the Xlib toolkit [SG87], by wrapping it to become the C2 *GraphicsBinding* component. The KLAX implementation built using the framework consists of approximately 8100 additional lines of commented C++ code.

Performance of the implementations was good on a Sun Sparc2 workstation, easily exceeding human reaction time if the *ClockLogic* component was set to use 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 typically caused 3 to 20.

IV. 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. 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 in its present form. If we were unable to do so, there would be at least three possible 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.[2]
- *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. 3 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.

V. Integrating SkyBlue with KLAX

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 are local constraints. The *Resizing* constraint pertains to several game elements, and would thus belong in a separate component.

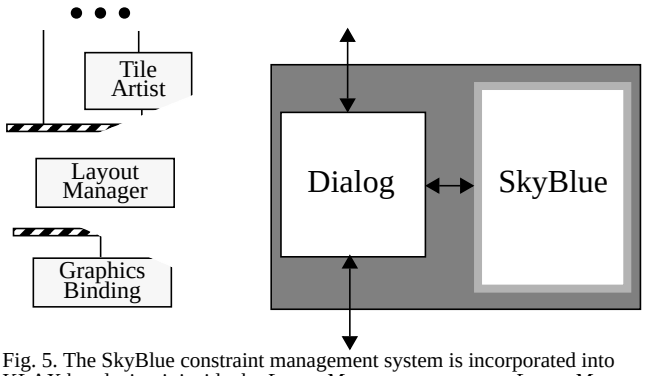

Fig. 5. The SkyBlue constraint management system is incorporated into KLAX by placing it inside the *LayoutManager* component. *LayoutManager*'s dialog handles all the C2 message traffic.

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. 4. 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 its 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. 5. *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.[3]

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. Only 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. Integrating Amulet with KLAX

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 effects [MORT96].

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. This exercise required extracting and recompiling the needed portion of Amulet.[4] Once the solver was extracted from the rest of Amulet, it was successfully substituted for SkyBlue and tested by one developer in 75 minutes. As anticipated, no architecture-wide changes were needed. The look-and-feel of the application remained unchanged. There was again no performance degradation.[5]

2. Location is an integer between 1 and 5.

3. 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 the *LayoutManager* from Fig. 5.

4. This may seem like unnecessary work. However, we were unable to locate implementations of any other constraint solvers, which was the deciding factor in our selection. Furthermore, the availability of Amulet's source code and its implementation language (C++) made it a good candidate for this project.

5. 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."

VII. KLAX Component Library

Integrating SkyBlue and Amulet with KLAX provided an opportunity for building multiple versions of *PaletteADT*, *PaletteArtist*, and *LayoutManager* components. The two integrations described above 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).[6]

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 3). 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 V). 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. Three such variations were described in Section III (using versions 1 of all five components), Section V (using versions 2 of the five components), and Section VI (replacing *LayoutManager*-2 with *LayoutManager*-3 in the architecture from Section V). 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 *PaletteArtist*, while the *Resizing* constraints are maintained globally by *LayoutManager*. Therefore, multiple instances of SkyBlue maintain the constraints in different KLAX components.

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. 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

6. 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).

- 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 some other component will maintain the *Palette Boundary* constraint. However, *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 *Graphics-Binding* component (see Section III) 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.

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 [San94, MM95]. 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. 6).

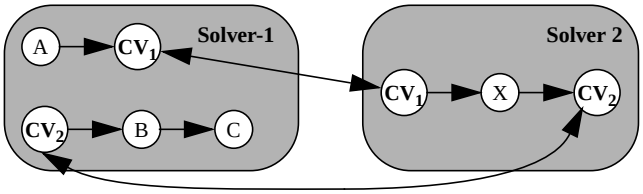

Fig. 6. To provide consistent constraint maintenance across constraint solvers, each conceptually common constraint variable (CV) is implemented with two actual variables. Changes in one are automatically reflected in the other.

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, may play in its resolution.

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. 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. 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.

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 [TMA+96] 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.[7] Thus, the 18 components depicted in Table 1 can generate 384 distinct versions of KLAX. Of course, every possible architecture is neither meaningful (e.g., the example of partial communication in Section VIII.B) nor particularly interesting. This exercise fulfilled its purpose nonetheless, since it demonstrated the ease with which a library of components and an application family in the C2 style may be created.

Finally, we now have a constraint management component in the C2 style that will be reused across future applications. Beyond this immediate benefit, integrating SkyBlue and Amulet with KLAX has taught us an invaluable lesson on the intricacies of incorporating OTS components into C2 architectures. We will build upon this experience in our exploration of other facets of C2.

X. 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.

7. In [MORT96] we demonstrate how such components, e.g., the different versions of the *LayoutManager*, comprise a type hierarchy.

XI. References

[Big94] T. J. Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. *IEEE International Conference on Software Reuse*, November 1994.

[GAO95] 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*, Seattle, WA, April 1995.

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

[Kru92] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, pages 131-183, June 1992.

[MM95] 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.

[MORT96] 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)*, San Francisco, CA, October 1996.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, pages 40-52, October 1992.

[RR96] J. E. Robbins and D. Redmiles. Software architecture design from the perspective of human cognitive needs. In *Proceedings of the California Software Symposium (CSS'96)*, Los Angeles, CA, USA, April 1996.

[San94] 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*, Marina del Ray, CA, November 1994, pages 137-146.

[SG87] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, June 1987.

[Sha95] 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*, April 1995.

[TMA+95] 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)*, Seattle, WA, April 1995, pages 295-304.

[TMA+96] 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*, pages 390-406, June 1996.