

Using Object-Oriented Typing to Support Architectural Design in the C2 Style

Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92697-3425
{neno,peymano,jrobbins,taylor}@ics.uci.edu
Phone: (714)824-3100 Fax: (714)824-4056

Abstract -- Software architectures enable large-scale software development. Component reuse and substitutability, two key aspects of large-scale development, must be planned for during software design. Object-oriented (OO) type theory supports reuse by structuring inter-component relationships and verifying those relationships through type checking in an architecture definition language (ADL). In this paper, we identify the issues and discuss the ramifications of applying OO type theory to the C2 architectural style. This work stems from a series of experiments that were conducted to investigate component reuse and substitutability in C2. We also discuss the limits of applicability of OO typing to C2 and how we addressed them in the C2 ADL.¹

Keywords -- software architectures, architectural styles, software design, object-oriented typing, reuse

I. Introduction

The promise of software architectures is that they enable component-based development of large-scale software [GS93, PW92]. Component reuse and substitutability are key to large-scale development and must be planned for during software design. The techniques for doing so in the small (e.g., separation of concerns or isolation of change) become insufficient in the case of development with (possibly preexisting) large components that encapsulate several abstract data types (ADTs) and provide significant functionality. The question then becomes what kind of support practitioners need to fully take advantage of the potential for reuse and substitutability in architecture-based software development.

[GKWJ94] recognize that objects are important for effective reuse. Furthermore, our experience in developing a collection of reusable and substitutable components for a family of applications indicated that object-oriented (OO) type theory may provide the needed answer. [Gar95] recognizes that an architectural style can be viewed as a system of types, where the architectural vocabulary (components and connectors) is defined as a set of types. If specified in an OO context, type hierarchies of architectural elements are also possible, where, e.g., one component is a subtype of another.

Specifying architectural elements as OO type hierarchies is a domain-independent approach that structures relationships between software components and enables us to verify those rela-

tionships via type checking. It allows us to link much of the discussion of component compositionality, reusability, and substitutability to the terminology of OO types [PS91, PS92, LW94]. Doing so serves three purposes:

- It enables readers whose primary expertise is in the area of OO type theory to relate the concepts and terminology of software architectures to those with which they are familiar, and vice versa. Establishing the similarities and differences between them will help identify the limits of applicability of techniques developed in one to the other.
- It helps clarify the composition properties of components and connectors. For example, determining when one component may be substituted for another is akin to subtyping in OO programming languages (OOPLs).
- Once this relationship is established, software architectures can further leverage the results of research in well understood areas of OOPL, such as analysis and code generation.

Drawing parallels between software architectures and OO types will shift the perception of software architectures from a network of concrete components to a network of abstract component placeholders that can possibly be bound to concrete components. This view of architectures provides a more flexible framework for dealing with component reusability and substitutability, and incremental refinement of architectures.

In this paper, we identify the issues and discuss the ramifications of applying OOPL type theory to architectures. The contribution of the paper is threefold: (1) it establishes the role of OO typing in architectural design, (2) it demonstrates the need for multiple type checking mechanisms in architectures, and (3) it specifies the features an architecture definition language (ADL) [AG94, DK76, LV95, NM94] should have to support component type hierarchies. While understanding the relationship between OO typing and architectures in general is important, rather than addressing this problem in the abstract, we focus on a particular architectural style, C2 [TMA+96]. We present conclusions from a series of experiments that were conducted to investigate the questions of component reuse and architectural flexibility in C2. Basing our conclusions on actual projects in a specific architectural style serves as the initial proof of concept, essential in reassessing our approach to studying architectures. Moreover, we consider the rules of C2 to be general enough for a majority of our conclusions to be potentially applicable across styles.

The paper is organized as follows. Section II briefly outlines the problem and our conclusions. It is followed by three sections which present the background on which our work is based: Section III summarizes the rules and intended goals of C2, Section IV presents an example application built in the C2 style and several of its variations, and Section V gives an overview of subtyping mechanisms in OOPL. Section VI provides an in-depth discussion of the relationship between the concepts and terminology of C2 and OO types. Discussions of related and future work and conclusions round out the paper.

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.

II. Motivation

Issues of component composability, substitutability, and reusability are critical to software architectures. The perspective of architectures as hierarchies of OO types can highlight many of the problems inherent in trying to accomplish these critical goals. Furthermore, it can help identify approaches to achieving them.

When dealing with collections of architectural components as type hierarchies, architectural design involves

- identifying the types needed in an architecture (abstract components);
- selecting and subtyping from suitable existing types (concrete off-the-shelf (OTS) components) to achieve the desired functionality. This step is potentially automatable, as it can exploit OOP type conformance rules;
- creating new types (custom-designed components).

However, OOP type checking is not sufficient to fulfill all the needs of software architectures. Unlike OOPs, architectures may contain components implemented in heterogeneous programming languages. Furthermore, software component frameworks may require subtyping methods not commonly found in OOPs. Finally, while OOPs generally adopt a single type checking mechanism (e.g., monotone subclassing [PS92]), our experience indicates that architectures often require multiple subtyping mechanisms. Therefore, software architectures need to expand upon the lessons learned from OOPs and provide such facilities in ADLs.

III. Overview of C2

C2 is a component- and message-based style designed to support the particular needs of applications that have a graphical user interface aspect, with the potential for supporting other types of applications as well. The style supports a paradigm in which UI components, such as dialogs, structured graphics models (of various levels of abstraction), and constraint managers, can more readily be reused. A variety of other goals are potentially facilitated as well. These goals include the ability to compose systems in which: components may be written in different programming languages, components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple users may be interacting with the system, multiple toolkits may be employed, multiple dialogs may be active (and described in different formalisms), and multiple media types may be involved.

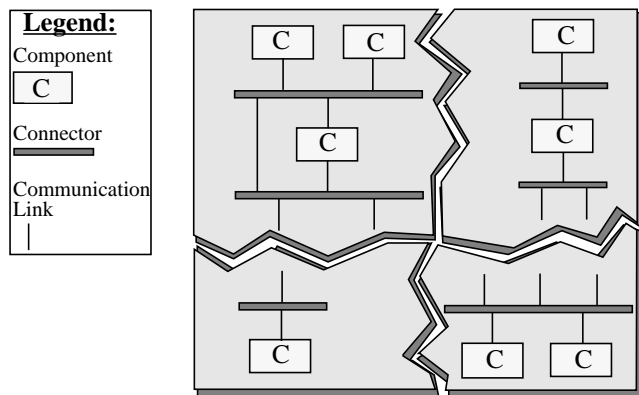


Fig. 1. A sample C2 architecture. Jagged lines represent the parts of the architecture not shown.

The C2 style can be informally summarized as a network of concurrent components hooked together by connectors, i.e., message routing devices. 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 connector (see Fig. 1).

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 it 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 solely achieved by exchanging messages. 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. Substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. To eliminate a component’s dependence on its “superstrate,” i.e., the components above it, the C2 style introduces the notion of event translation: 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. This issue has also been identified by the OO community [GHJV95, Hol93, YS94]. The C2 design environment [RR96] is, among other things, intended to provide support for accomplishing this task.

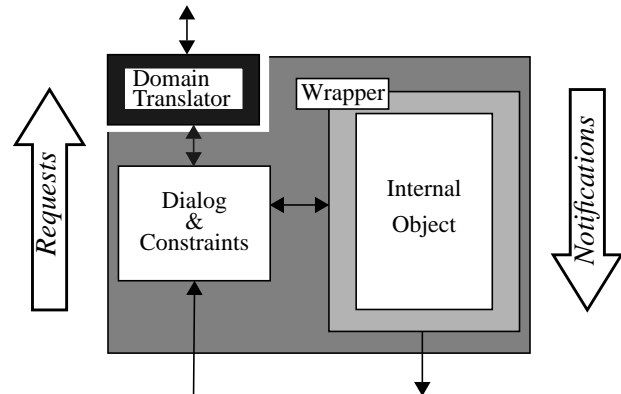


Fig. 2. The internal architecture of a C2 component.

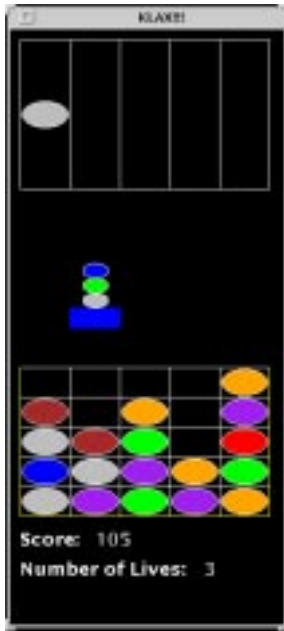
The internal architecture of a C2 component shown in Fig. 2 is targeted to the user interface domain. While issues concerning composition of an architecture are independent of a component’s internal structure, for purposes of exposition below, this internal architecture is assumed.

Each component may have its own thread(s) of control and there is no assumption of a shared address space among components. These properties simplify modeling and programming of multi-component, multi-user, and concurrent applications and enable exploitation of distributed platforms. A proposed conceptual architecture is distinct from an implementation, however, so that it is indeed possible for components to share threads of control and address spaces.

IV. An Example C2 Application

The experiments described in this section serve as a proof of concept in relating architectures to OO typing. An example application built in the C2 style is a version of the video game KLAX.² A description of the game is given in Fig. 3. The game is based on common computer science data structures and its layout maps naturally to modular artists.

2. KLAX is trademarked 1991 by Atari Games.



KLAX Chute
Tiles of random colors drop at random times and locations.

KLAX Palette
Player-controlled 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.

A. Conceptual KLAX Architecture

The design of the system is given in Fig. 4. The components that make up KLAX can be divided into three logical groups. At the top of the architecture are the components which encapsulate the game's state. These 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 artists components.

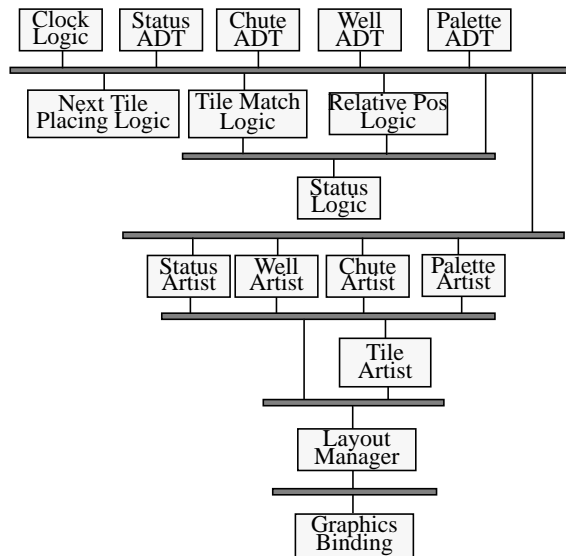


Fig. 4. Conceptual C2 architecture for KLAX. Note that the Logic and Artist layers do not communicate directly and are in fact siblings. The Artist layer is shown below the Logic layer since the components in the Artist layer perform functions closer to the user.

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 chute, the *RelativePositioning-Logic* determines if the *palette* is in a position to catch the tile. If

so, a request is sent to the *PaletteADT* to catch the tile. Otherwise, a notification is sent that a tile has been dropped. This notification is detected by the *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. The *TileArtist* provides a flexible presentation level for tiles. Artists maintain information about the placement of abstract tile objects. The *TileArtist* intercepts 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.

B. C2 Class Framework Used in the Development of KLAX

To support the implementation of the KLAX architecture, a C++ object-oriented component and message-passing framework was developed. The framework consists of 12 classes for C2 concepts, as shown in Fig. 5. It provides an abstract class hierarchy from which concrete C2 architectural elements are subtyped.

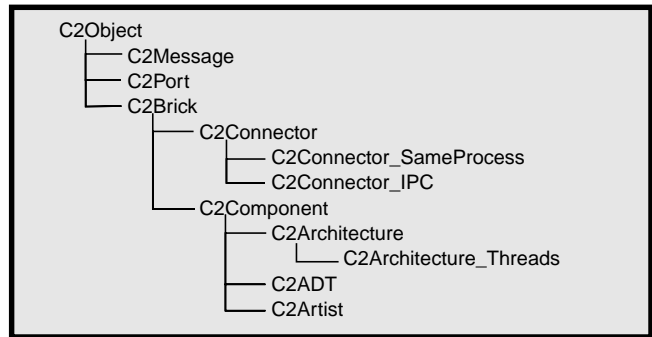


Fig. 5. The C++ object-oriented framework used in the development of KLAX.

The size of the framework is approximately 3100 commented lines of C++ code and it was built in 150 programmer hours. The framework supports a variety of implementation configurations for a given architecture. Multi-threaded and multi-process KLAX configurations are discussed in [TMA+96]. This framework also allowed us to integrate several external components. Furthermore, it provides structure for component implementations and eliminates many repetitive programming tasks.

C. Variations

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. In this section, we describe several variations that are pertinent to our discussion of relating architectures to OO types in Section VI.

1) *Multi-Lingual Implementation.* The *TileArtist* was implemented both in C++ and Ada. The only functional difference between them is that the C++ *TileArtist* displays the tiles as ovals and the Ada *TileArtist* as rectangles. The two artists can be swapped dynamically.³

2) *Transforming the Application.* A variation of the original architecture, shown in Fig. 6, involved replacing the original *TileMatchLogic*, *NextTilePlacingLogic*, and *TileArtist* components

with components which instead matched, placed, and displayed letters. This transformed the objective from matching the colors of tiles to spelling words. Each time a word is spelled correctly, it is removed from the well. The *SpellingLogic* component wrapped an existing spell-checker, written in C.

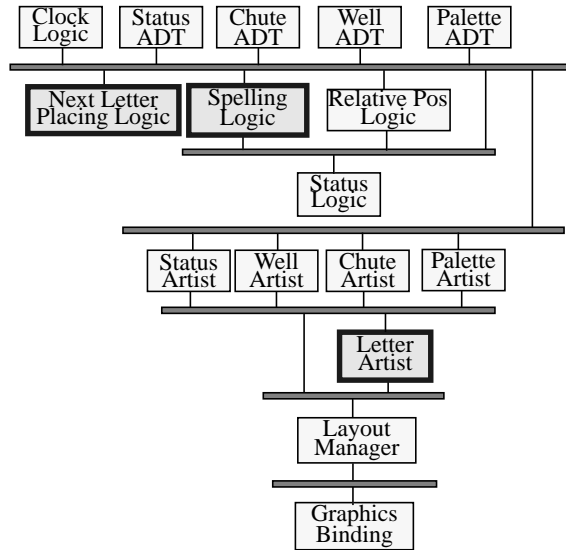


Fig. 6. A variation on KLAX. By replacing three components from the original architecture, the game turns into one whose object is to spell words horizontally, vertically, or diagonally.

3) *Integrating Off-The-Shelf Constraint Solvers.* Another variation of the original architecture involved incorporating a research-off-the-shelf constraint management system, SkyBlue [San94], into the application. Constraint-management code, such as specifying the left and right boundaries of palette’s movement, was dispersed throughout the original application. This variation replaced that code with SkyBlue constraints. Furthermore, SkyBlue was adapted to communicate with other KLAX components via C2 messages by placing it inside the *LayoutManager*, as shown in Fig. 7. A constraint management component in the C2 style was thus created.

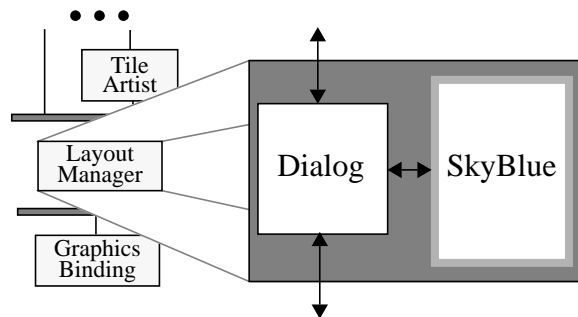


Fig. 7. The SkyBlue constraint management system is incorporated into KLAX by placing it inside the *Layout Manager* component. *Layout Manager*’s dialog handles all the C2 message traffic.

SkyBlue was then replaced with a constraint solver from the Amulet user interface development environment [MM95].⁴ This exercise was intended to explore any potential global (architecture-wide) effects of substituting one constraint manager for another, as

3. To accommodate Ada and C++ in the same application, the *TileArtist* executes in its own process.

4. For simplicity, unless otherwise noted, we will refer to Amulet’s constraint manager simply as “Amulet” in the remainder of the paper.

well the possibility of multiple constraint managers being active in the same architecture. As we had expected, changes that were required to substitute SkyBlue with Amulet were localized to the *LayoutManager*. Furthermore, we demonstrated the simultaneous usage of two constraint managers within an architecture, both at different levels of the architecture, and within a single component [MT96].

4) *Plug-and-Play.* Several variations of KLAX were built using the various versions of components that maintained all or some of their constraints internally (with in-line code or using SkyBlue and/or Amulet) or externally, in the *LayoutManager*. This exercise explored *partial communication and partial service utilization*, i.e., architectural configurations where components both request and provide either more or fewer services than are needed by the components with which they communicate [TMA+96].

Several important aspects of the C2 style were explored in these experiments. Four external components, Xlib, SkyBlue, Amulet, and the spelling checker, were integrated. Component substitution was used to provide alternative presentations of tiles and to transform the application into spelling KLAX, another game in the same application family. The experiments also demonstrated support for multi-lingual components in C, C++, and Ada. A reusable C2 framework that supports multiple implementations of a given C2 architecture was also explored. Finally, the experiments served as a proof of concept in relating architectures to OO typing.

V. Overview of OO Subtyping Mechanisms

Typing enables type checking, the determination of whether a formal parameter of one type may be legally bound to an object of another type. That notion of legality can help software developers keep program semantics close to programmer intentions, and thus discipline the (re)use of objects. Furthermore, a combination of type declarations and type inference support source code understandability and the generation of efficient executable code. OO typing mechanisms may operate on the interfaces of classes or their implementations. We focus on interface typing mechanisms because of their relevance to reuse of heterogeneous components in architectural design.

[PS92] describes a consensus in the OO typing community regarding the definition of a range of OO typing mechanisms. *Arbitrary subclassing* allows any class to be declared a subtype of another, regardless of whether they share a common set of methods. *Name compatibility* demands that there exist a shared set of method names available in both classes. *Interface conformance* constrains name compatibility by requiring that the shared methods have conforming signatures. *Monotone subclassing* requires that the subclass relationship be declared and that the subclass must preserve the interface of the superclass. *Behavioral conformance* [LW94] allows any class to be a subtype of another if it preserves the interface and behavior of all methods available in the supertype. *Strictly monotone subclassing* also demands that the subtype preserve the particular implementations used by the supertype.

Protocol conformance goes beyond the behavior of individual methods to specify constraints on the order in which methods may be invoked. Explicitly modeling protocols as state machines has practical benefits [LM95, Nie93, YS94]. However, the preconditions and postconditions of methods can be used to describe all state-based protocol constraints and transitions. Thus, behavioral conformance implies protocol conformance, and we need not address them separately.

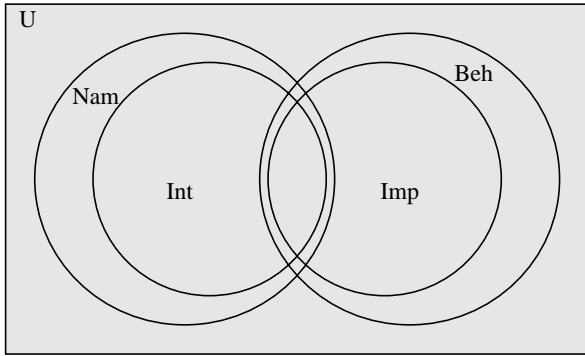


Fig. 8. A framework for understanding OO subtyping mechanisms as regions in a space of type systems.

Fig. 8 provides a framework for understanding these subtyping mechanisms as regions in a space of type systems, labeled U . The regions labeled *Int* and *Beh* contain systems that demand that two conforming classes share interface and behavior, respectively. The *Imp* region contains systems that demand classes share particular implementations of all superclass methods, which also implies that classes preserve the behavior of their superclasses. The *Nam* region demands only shared method names, and thus includes every system that demands interface conformance. Each typing mechanism described above can be denoted via set operations on these regions. For example, *behavioral conformance*, which requires that both interface and behavior of a class be preserved, corresponds to the intersection of the *Beh* and *Int* regions and is expressed as *Beh and Int*. We use this framework below in Section VI.B to describe how new and existing typing mechanisms are useful in C2-style architectures.

VI. Relating Architectural Concepts to OO Types

A conceptual C2 component may be viewed as an OOPL class, but they are not identical. The services a component provides are equivalent to a class specification, and the requests it sends correspond to OO messages. However, no OOPL concept corresponds to C2 notifications. State changes of C2 components are reified as notifications and no assumptions are made about the existence or number of their recipients, resulting in the possibility of messages being ignored in a C2 architecture.

The internal architecture of a C2 component also differentiates it from an OO class. A canonical C2 component contains three distinct building blocks: the dialog, the internal object, and the optional domain translator (see Fig. 2). The dialog presents the component's interface (both top and bottom). The domain translator, if present, modifies the top interface. The dialog and internal object may be very large, containing several ADTs and significant functionality. For example, *LayoutManager*'s internal object in Fig. 7 is an entire constraint solver, while *GraphicsBinding* from Fig. 4 encapsulated Xlib.

Given this internal architecture, it is possible to generate a subtype of a given component by subtyping from any or all of the three internal blocks. For example, let D be the dialog of component C and let D' be a subtype of D created by strictly monotone subclassing. Then, D' could be used in the place of D to create component C' . C' can then be considered a subtype of C . A specific example of this is discussed below.

The major differentiators between C2 and OOPL are the distinction between notifications and requests, component granularity, the internal component architecture, and the topological constraints imposed by the C2 style on a set of components in an architecture. However, the similarities between C2 components and OO classes allow us to explore the ramifications of OO sub-

typing on reusability and substitutability of C2 components and analysis of C2 architectures. For that reason and for the purpose of the discussion below, we will treat a component as a class in the OO sense, with its top and bottom interfaces treated as parts of a single interface.

This assumption allowed us to specify the complete framework of components used in the different variations of KLAX (Section IV). A portion of this framework is shown in Fig. 9. It can be viewed as a directed graph whose arcs represent the subtyping relationship. For the graph to be more informative and useful, the nodes (components) could be annotated with information such as the architectural style to which the component adheres, the implementation language, and the location of the component source code. The arcs should be annotated with the appropriate subtyping method.⁵ As a rule-of-thumb, one KLAX component may be substituted for another only if there exists a directed path between them.

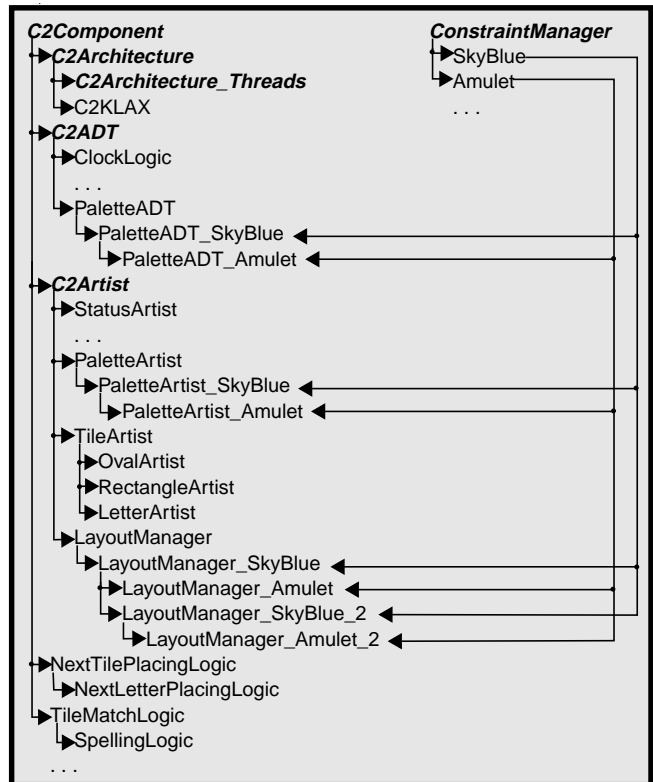


Fig. 9. Partial C2 component framework for KLAX.

In the following three subsections, we discuss the implications of describing C2 components with OO typing. Section VI.A discusses the effect of OO typing on architectural design. Section VI.B demonstrates the need for multiple type checking mechanisms in C2 architectures. Finally, Section VI.C discusses the influence of OO typing on the C2 ADL and is based upon the insights of the previous two subsections.

A. Role of OO Types in C2 Architectural Design

During architectural design, a system is described using conceptual components⁶ that represent placeholders for concrete com-

5. For clarity, the nodes and arcs have not been annotated in Fig. 9. Clearly, a different representation, such as partial and/or incremental views aided by hypertext, is needed for representing OO frameworks corresponding to large collections of components.

6. For brevity, we limit the rest of our discussion to components, although similar arguments could be made for connectors.

ponents. The behaviors of these conceptual components typically describe the minimal functionality needed to satisfy system requirements. As the architecture is refined, conceptual components are bound to concrete components. Although a one-to-one correspondence between conceptual and concrete components may exist, it is more likely that the correspondence is one-to-many, many-to-one, or many-to-many.⁷ Additionally, concrete components rarely correspond perfectly to conceptual components, necessitating some custom glue code. We came across several instances of these possibilities during the implementation of KLAX:

- *One concrete component for one conceptual component:* Since we had no preexisting components for this application family, a majority of the components were built specifically for our first game variation. Subsequent variations reused a majority of these components.
- *Multiple concrete components for one conceptual component:* One version of the *LayoutManager* was implemented using a combination of the SkyBlue and Amulet constraint managers, even though the full functionality of these OTS components was not utilized.
- *One concrete component for multiple conceptual components:* An OTS database component could have been used to implement all the conceptual game state components.
- *Multiple concrete components for multiple conceptual components:* In one variation of the implementation, the *LayoutManager* used Amulet and SkyBlue, while, at the same time, *PaletteADT* and *PaletteArtist* used SkyBlue.
- *Domain translation:* A different spelling checker component could have been used in place of the *SpellingLogic* component by adding a domain translator to its top domain and adding or modifying the domain translators of components below it in the architecture.

Determining whether a conceptual component can be bound to a concrete component is analogous to type checking method parameters in OOPLs. There are several important similarities and differences between type checking a C2 architecture and type checking method parameters in an OOPL.

- *Correctness and retrieval:* In OOPLs and architecture, type conformance is useful for determining correctness. In architecture, conceptual components may also be used to select and retrieve compatible concrete components from a component repository, potentially supporting semi-automatic generation of systems.
- *Degree of conformance:* In OOPLs, type compatibility is characterized as either legal or illegal. Although it is beneficial to characterize component compatibility in this way, determining the degree of compatibility is more useful. Degree of compatibility helps determine the amount of work necessary to retrofit a component for use in the system. Although a general technique for determining the degree of compatibility may be intractable, adequate heuristics and solutions for specific application domains may be possible. Such heuristics can be supported by a software architecture design environment [RR96].
- *Type bindings:* In OOPLs, a formal type is bound to a single actual type at any point in time. In architecture, one or more conceptual components may be bound to the combined functionality of several concrete components. This added flexibility also increases the difficulty of type checking an architecture.
- *Multiple binding rules:* In OOPLs, in the case where several formal types are compatible with an actual type, other typing rules, e.g., selecting the most specific type, are used. Similar choices must be made in an architecture since several concrete components may implement a conceptual component. For example, the selection rules for determining the most appropriate component

7. These relationships may be selective, where one of a number of candidate components is chosen, or cumulative, where several components are chosen.

may be based on non-functional requirements, such as memory requirements, source language, or licensing costs.

- *Late binding:* In OOPLs, formal parameters may be bound to actual parameters of several types. This is also needed in architecture, since concrete OTS components may be developed independently of conceptual components. As a result, explicit relationships between concrete and conceptual components may not exist. During architecture refinement, implicit relationships need to be inferred by type checking and made explicit.
- *Subclassing:* In OOPLs, subclassing is an effective mechanism for explicitly specifying subtyping relationships when creating types based on existing types. In architecture, subclassing relationships are useful for characterizing concrete component hierarchies when new components may be based on existing components. But subclassing relationships may be too restrictive for relating concrete components to conceptual components since the two may be specified independently of each other.

The similarities between OOPL typing and architectures provide insights into how techniques developed in the former may be applied to the latter. The differences demonstrate the need for further research to address the inadequacies in applying such techniques to software architectures.

B. Type Checking C2 Architectures

Given the subtyping demands of C2 architectures, we now describe how OOPL type checking mechanisms may be used to satisfy these needs. We have found the following OOPL type conformance mechanisms useful in characterizing C2 architectures:

- *Interface conformance (Int)*⁸, was useful when interchanging components without affecting dependent components. The *SpellingLogic* component in the Spelling KLAX architecture (Fig. 6) used interface subtyping to provide a new implementation for the *TileMatchingLogic* component of the original architecture (Fig. 4).
- *Behavioral conformance (Beh and Int)*, can be useful, e.g., in demonstrating correctness during component substitution. In KLAX, we employed behavioral subtyping to provide an Ada implementation of the original *TileArtist* written in C++. Using behavioral subtyping, concrete components may be grouped into sets of substitutable components, facilitating semi-automatic component selection during system generation.
- *Strictly monotone subclassing (Int and Imp)*, can be useful, e.g., when extending the behavior of an existing component while preserving correctness relative to the rest of the architecture. In KLAX, it was used to describe the relationship between a component and its debugging version. The debugging version implemented additional functionality to respond to queries from a debugger. Using strictly monotone subclassing allowed us to monitor the implementation of the original component.
- *Implementation conformance with different interfaces (Imp and not Int)*, is useful in describing domain translators in C2, which allow a component to be fitted into an alternate domain of discourse. Domain translators provide functionality similar to that of the adapter design pattern [GHJV95].
- *Multiple conformance mechanisms* allows creation of a new type by subtyping from several types, potentially using different subtyping mechanisms. In KLAX, for example, *LayoutManager_SkyBlue* from Fig. 9 used monotone subclassing of the original *LayoutManager*, and strictly monotone subclassing of SkyBlue.

As the KLAX example demonstrates, no particular type conformance mechanism is adequate in describing all the subtyping relationships in a component framework. Relaxing the rules of a particular method to support our needs would sacrifice type checking precision. In order to describe typing relationships accurately

8. All type conformance expressions in the remainder of the paper refer to regions of the diagram in Fig. 8.

while preserving type checking quality, we have opted to use multiple type conformance techniques to describe C2 architectures.

C. Role of OO Types in the C2 ADL

Relating software architectures to OO types changes our perception of architectures from a network of components to a network of conceptual component placeholders that can be bound to concrete components. This has direct ramifications on architectural design: selecting and subtyping from concrete OTS components that match the generic placeholders can potentially be automated.

As the means by which C2 architectures are “programmed,” C2’s ADL [MTW96] is a key facet of architectural design. The ADL specifies the instantiation and interconnection of required architectural elements. ADL analysis tools can then ensure desired properties, such as type correctness.

```

architecture ::=
  architecture architecture_name is
    conceptual_components conceptual_component_list
    [ connectors connector_list ]
    [ architectural_topology topology ]
  end architecture_name;

system ::=
  system system_name is
    architecture architecture_name with
      component_instance_list
  end system_name;

component_instance_list ::=
  { conceptual_component_name is_bound_to
    { concrete_component_expression }+ }+

concrete_component_expression ::=
  concrete_component_name [ with (parameter_instantiation) ];

parameter_instantiation ::=
  identifier <= value { ; identifier <= value }

```

Fig. 10. A portion of the C2 ADL syntax that enables the binding of conceptual to concrete components.

The conceptual component placeholders in architectural design can be thought of as formal parameters in the ADL, while the OTS components that instantiate them are viewed as actual parameters, as shown in Fig. 10. An example from KLAX demonstrating the binding of conceptual to concrete components is shown in Fig. 11. The conceptual *TileArtist* component was originally implemented as a generic component. Both *OvalArtist* and *LetterArtist* were subtyped from this component (see Fig. 9), enabling us to create the two variations of KLAX from a single conceptual architecture.

```

system Geometric_KLAX is
  architecture KLAX with
    ...
    TileArtist is_bound_to OvalArtist;
  ...
end Geometric_KLAX;

system Spelling_KLAX is
  architecture KLAX with
    ...
    TileArtist is_bound_to LetterArtist;
  ...
end Spelling_KLAX;

```

Fig. 11. An example from KLAX demonstrating the binding of conceptual to concrete components in the C2 ADL.

Another role of the C2 ADL is in supporting multiple type-checking mechanisms. This stems in part from the fact that C2 architectures may incorporate components implemented in heterogeneous programming languages and hence cannot rely on a single subtyping method provided by any one language.

The syntax used by the C2 ADL to support subtyping is shown in Fig. 12. The “internal_block” represents the part of the internal component architecture for which the subtyping relationship is intended. The “all” value of the internal_block specifies the entire component.

```

component_subtyping ::=
  subtype_name is_subtype
    { internal_block <= internal_block
      supertype_name (type_conformance_expression); }+
  end_subtype;

type_conformance_expression ::=
  type_conformance { binary_operator [not] type_conformance }

internal_block ::= all | domain_translator | dialog | object

type_conformance ::= nam | int | imp | beh

binary_operator ::= and | or

```

Fig. 12. A portion of the C2 ADL syntax that enables subtyping from existing components.

An example from KLAX that illustrates the manner in which multiple subtyping methods are supported by the ADL is given in Fig. 13. As depicted in the directed graph in Fig. 9, *LayoutManager_SkyBlue* was subtyped from two components: the original *LayoutManager* and *SkyBlue*. It preserved *LayoutManager*’s interface, while the internal object combined the two components’ functionalities.

Note that the ADL only denotes the component subtyping relationships. It is the task of the developer to ensure that the appropriate changes (e.g., inserting Amulet in the place of SkyBlue in KLAX) are made to implement the new types. Furthermore, the ADL addresses only a subset of the issues identified in Section VI.A. The remaining issues, e.g., component retrieval, are outside the scope of an ADL and should be handled elsewhere in the architecture development environment.

```

LayoutManager_SkyBlue is_subtype
  all <= all LayoutManager (int and beh);
  object <= all SkyBlue (int and imp);
end_subtype;

LayoutManager_Amulet is_subtype
  all <= all LayoutManager_SkyBlue (int and beh);
end_subtype;
...

```

Fig. 13. An example demonstrating how multiple subtyping methods are used in a single C2 architecture.

VII. Related Work

This work draws from a number of other researchers and systems. The C2 style and the work in the area of OOPL and OO typing in particular were summarized in Section III and Section V respectively. Below, we present a cross-section of approaches in software architectures that attempt to build upon the benefits of OO typing.

The Aesop system [GAO94] describes an OO approach to defining new architectural styles using subclassing. This work employs OO typing at the level of canonical components for a particular style, which can be considered a level of abstraction above

our approach. Another difference is that while our approach allows the utilization of multiple type checking mechanisms, this approach requires behavioral conformance between subclasses and their superclasses.

[BO92] presents a strongly typed domain-independent model of hierarchical system design and construction based on interchangeable software components. As in C2, a component is a cluster of classes and it may be parameterized. Components that realize the same interface constitute a realm and may be interchanged in an architecture. Therefore, as was the case with Aesop, this approach adopts only a single type checking mechanism.

Realms, in effect, constitute very shallow but broad type hierarchies: all components within a realm are at the same level, and there is no clustering according to behavior, implementation, or other criteria. We believe that our approach is more flexible as it allows components that share an interface to be structured further by multiple and nested subtyping relationships. Furthermore, components which would belong to different realms (e.g., *LayoutManager* and its subtype *LayoutManager_SkyBlue*) are interchangeable under certain conditions.

A representative ADL is Rapide [LV95]. An interface in Rapide defines a component type and is similar to realms. An interface also specifies external behavior and constraints on that behavior. Rapide allows specification of components and architectures at different levels of abstraction, from a high level architecture to an executable system. Mappings are generated from one level of abstraction to another. [MQR95] presents a similar approach to architecture refinement. However, neither of the two approaches currently provides means for determining substitutability of one component for another as does OO subtyping.

Finally, domain-specific software architectures (DSSAs) [Tra95] address component reuse and substitutability by identifying components and topologies reused across architectures in a given application domain. However, DSSA reuse guidelines are closely tied to specific domains. Our reuse guidelines stem from domain-independent OO type theory.

VIII. Future Work

One aspect of our future work involves building tools to support the application of OOPL type theory to C2 architectures, e.g., automating the checking of types specified in the C2 ADL. We are planning to integrate our ideas into Argo [RR96], our C2 software architecture design environment. In relation to subtyping and type checking, we are also working on:

- relating type systems and the C2 style rules discussed in Section III,
- determining which conformance rules are most useful for specifying the top and bottom interfaces of components,
- determining which subtyping mechanisms are most useful in specifying composition properties for internal blocks of a C2 component,
- using type information to generate C2 domain translators.

Another aspect of our research concerns finding additional ways of relating concepts of OOPLs and software architecture. Both fields are active research areas, and their intersection contains many possibilities for future work. Several concepts directly relevant to C2 include:

- incorporating dynamic type checking for systems in which new components may be added at runtime,
- determining the trade-offs between static and dynamic type checking at the architectural level,
- using type information to select among several potential components and generate efficient concrete architectures,
- using type inference and type feedback at the architectural level [HU94],
- using type information to support architectural analysis.

IX. Conclusion

Issues of component composability, substitutability, and reuse are critical to large-scale software development. Viewing software architectures as hierarchies of OO types highlights potential approaches to addressing these critical issues.

In this paper, we have identified the implications of applying OO type theory to the C2 architectural style. Relating the two clarifies the composition properties of components and enables more aggressive architecture analysis. Specifically, by making subtyping relationships between components explicit, we can determine when one component may be substituted for another. By extending type checking mechanisms beyond those available in OOPLs, e.g., to allow multiple binding rules, we can express a richer set of relationships. More broadly, applying other concepts of OOPL typing to architecture enables a wider class of architectural analysis techniques. We have also related these concepts to the C2 ADL, which provides a language for capturing these relationships.

Our work stems from a series of experiments that were conducted to investigate the questions of component reuse and substitutability in C2. The similarities between OO typing and C2 have allowed us to build a large framework of components used in constructing a family of applications. However, OO type checking is not sufficient to fulfill all the needs of software architectures. Unlike OOPLs, architectures may contain components implemented in heterogeneous programming languages. Furthermore, software components may require subtyping methods not commonly found in OOPLs, such as implementation conformance with different interfaces. Finally, while OOPLs generally adopt a single type checking mechanism, C2 architectures require multiple subtyping mechanisms. We have expanded upon the lessons learned from OOPL and provided these facilities in the C2 ADL.

X. Acknowledgments

We would like to acknowledge K. Anderson, K. Nies, E. J. Whitehead, Jr., and D. Dubrow for their contribution on various aspects of C2. We also thank K. Nies, S. Shukla and H. Ziv for helpful comments on early drafts of the paper.

XI. References

- [AG94] 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.
- [BO92] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, pages 355-398, October 1992.
- [DK76] F. DeRemer and H. H. Kron. Programming in the Large Versus Programming in the Small. *IEEE Transactions on Software Engineering*, pages 80-86, June 1976.
- [Gar95] D. Garlan. What is Style? In David Garlan, editor, *Proceedings of the First International Workshop on Architectures for Software Systems*, pages 96-100, April 1995.
- [GAO94] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175-188, New Orleans, LA, USA, December 1994.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.
- [GKWJ94] M. L. Griss, W. Kozaczynski, A. I. Wasserman, C. Jette, and others. Object-Oriented Reuse. In

- Proceedings of the Third International Conference on Software Reuse*, Rio de Janeiro, Brazil, November 1994.
- [GS93] D. Garlan and M. Shaw. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing, 1993.
- [Hol93] U. Holzle. Integrating Independently-Developed Components in Object-Oriented Languages. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, pages 36-56, Kaiserslautern, Germany, July 1993.
- [HU94] U. Holzle and D. Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 326-336, Orlando, FL, June 1994.
- [LM95] D. Lea and J. Marlowe. Interface-Based Protocol Specifications of Open Systems Using PSL. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, pages 374-398, Aarhus, Denmark, August 1995.
- [LV95] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, pages 717-734, September 1995.
- [LW94] B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.
- [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.
- [MT96] N. Medvidovic and R. N. Taylor. Reuse of Off-the-Shelf Constraint Solvers in C2-Style Architectures. Technical Report UCI-ICS-96-28, University of California, Irvine, July 1996.
- [MTW96] N. Medvidovic, R. N. Taylor, E. J. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium (CSS'96)*, Los Angeles, CA, April 1996.
- [MQR95] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, pages 356-372, April 1995.
- [Nie93] O. Nierstrasz. Regular Types for Active Objects. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'93)*, Washington, D.C., USA, October 1993.
- [NM94] O. Nierstrasz and T. D. Meijler. Requirements for a Composition Language. In *Proceedings of ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, pages 147-161, Bologna, Italy, July 1994.
- [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.
- [PS91] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 146-161, Phoenix, AZ, USA, October 1991.
- [PS92] J. Palsberg and M. I. Schwartzbach. Three Discussions on Object-Oriented Typing. *ACM SIGPLAN OOPS Messenger*, 3(2):31-38, 1992.
- [RR96] J. 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.
- [TMA+96] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, pages 390-406, June 1996.
- [Tra95] W. Tracz. "DSSA (Domain-Specific Software Architecture) Pedagogical Example." *Software Engineering Notes*, July 1995.
- [YS94] 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*, pages 176-190, Portland, OR, USA, October 1994.