Formal Modeling of Software Architectures at Multiple Levels of Abstraction

Nenad Medvidovic
Department of Information and Computer Science
University of California
Irvine, CA 92717-3425 USA
(714) 824-4047
neno@ics.uci.edu

Richard N. Taylor
Department of Information and Computer Science
University of California
Irvine, CA 92717-3425 USA
(714) 824-6429
taylor@ics.uci.edu

E. James Whitehead, Jr.
Department of Information and Computer Science
University of California
Irvine, CA 92717-3425 USA
(714) 824-2776
ejw@ics.uci.edu

ABSTRACT

Software architectures are multi-dimensional entities that can be fully understood only when viewed and analyzed at four different levels of abstraction: (1) internal functionality of a component, (2) the interface(s) exported by the component to the rest of the system, (3) interconnection of architectural elements in an architecture, and (4) rules of the architectural style. This paper presents the characteristics of each of the four levels of architectural abstraction, outlines the kinds of analyses that need to be performed at each level, and discusses the kinds of formal notations that are suitable at each level. We use the pipe-and-filter and Chiron-2 (C2) architectural styles as illustrations. In particular, we present formal models of C2 at the last three levels of abstraction as a first step in enabling a C2 design environment to perform the necessary analyses of architectures. We discuss the benefits of the formal definitions and our experience to date.[1]

Keywords

Software architectures, architectural styles, formalism, architecture definition languages, interface definition languages

INTRODUCTION

Software architectural styles, such as Unix's pipe-and-filter style or AI's blackboard architectures, are key design idioms [9][23]. Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements (software components, connectors, etc.) and their overall interconnection structure. Development tools that operate on architectural specifications are as important as tools that work on individual components. In particular, architectural design environments [5][7] can provide a platform on which designers can construct an architectural model of a software system, have that model checked for syntactic and semantic correctness, receive domain-specific feedback about various design qualities, keep track of unfinished steps in the design process, and generate running programs for that system, while preserving the properties of the model [20].

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.

Software architectures are multi-dimensional entities that can be fully understood only when viewed at four levels of abstraction: internal functionality of a component, the interface(s) exported by the component to the rest of the system, interconnection of architectural elements in an architecture, and rules of the architectural style.

The work done by a design environment must span all four levels of abstraction. Analysis of architectures may include, but is not restricted to, checking for adherence to the style, correctness of components and architectures, interface matching among interacting components, generation of component domain translators in the cases of interface mismatches, concurrency issues, such as deadlock and starvation, and implementation issues, such as operating system process splits and host loads. In order to perform such analyses, a formal model is needed for each level of abstraction. Different formal notations [30] may be best suited for various levels' models.

In this paper, we:

- present arguments for modeling software architectures at the four distinct levels of abstraction and assess the benefits of doing so,
- argue for the potential utility of formalization at each of the four levels,
- discuss the maturity of existing formal modeling techniques at each level, and
- discuss ways in which a specific architectural style, Chiron-2 (C2) [27], can be formally modeled at the different levels of abstraction, model representative features of C2 at each level, and discuss the lessons learned from doing so.

The paper is organized as follows: An overview of the C2 style is given in the next section. The section subsequent to that gives a detailed exposition on the multi-level perspective of software architectures. It demonstrates the utility and applicability of the technique both by discussing ways in which an existing architectural style (pipe-and-filter) can be modeled and by modeling C2 at each level. "Value of the C2 Modeling Formalisms" discusses the benefits and drawbacks of formally modeling C2 at the different levels of abstraction. Finally, the conclusion rounds out the paper.

OVERVIEW OF C2

C2 is a component- and message-based architectural style designed to support the particular needs of applications that have a graphical user interface aspect, with potential for supporting other types of applications. 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 supported as well. These goals include the ability to compose systems in which: components may be written in different programming languages; components may be running in a distributed, heterogeneous environment without shared address spaces; architectures may be changed dynamically; multiple users may be interacting with the system; multiple toolkits may be employed; multiple dialogs may be active (and described in different formalisms); and multiple media types may be involved.

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 attached to the bottom of a single connector and the bottom of a component may be attached 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. Note that 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).

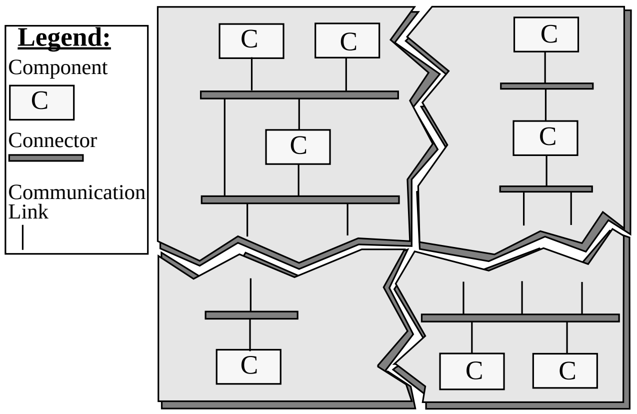


Fig. 1. A sample C2 architecture. Jagged lines represent the parts of the architecture not shown.

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 solely 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 well 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. One goal of the C2 design environment [25] is to provide support for accomplishing this task.

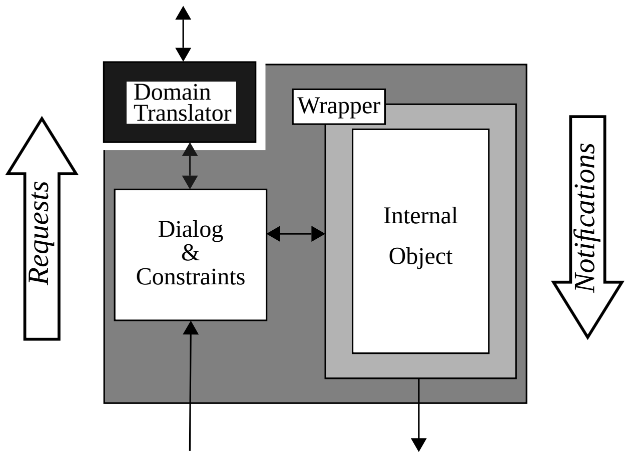


Fig. 2. The Internal Architecture of a C2 Component. A domain translator subcomponent may be present to assist in mapping between the component's internal semantic domain and that of the connector above it.

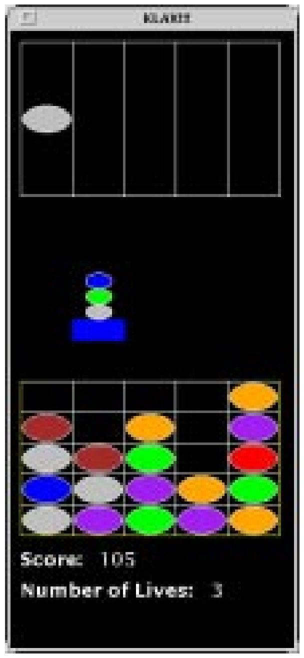
The internal architecture of a C2 component shown in Fig. 2 is targeted to the user interface domain. While issues concerning composition of components to form an architecture are independent of a component's internal structure, for purposes of exposition below, this internal architecture is assumed.

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. Note that separating components into different threads of control is not a requirement of the style. Moreover, 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.

An example C2 application

An example of an 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. 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.



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^{TM} video game.

The design of the system is given 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 data structure components are placed at the top since they receive no notifications, but respond to requests and emit notifications of internal state changes. ADT notifications are directed to the next level where they are received by both the game logic components and the artist components.

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

The artist components also receive notifications of ADT state changes, causing them to update their depictions. Each artist maintains the state of a set of abstract graphical objects which, when modified, send state change notifications in hope that a lower-level graphics component will render them. The *tile artist* provides a flexible presentation level for tiles. Artists maintain information about the placement of abstract tile objects. The *tile artist* intercepts any notifications about tile objects and recasts them to notifications about more concrete drawable objects. For example, a "tile-created" notification might be translated into a "rectangle-created" notification. The *layout manager* component receives all notifications from the artists and offsets any coordinates to ensure that the game elements are drawn in their correct positions.

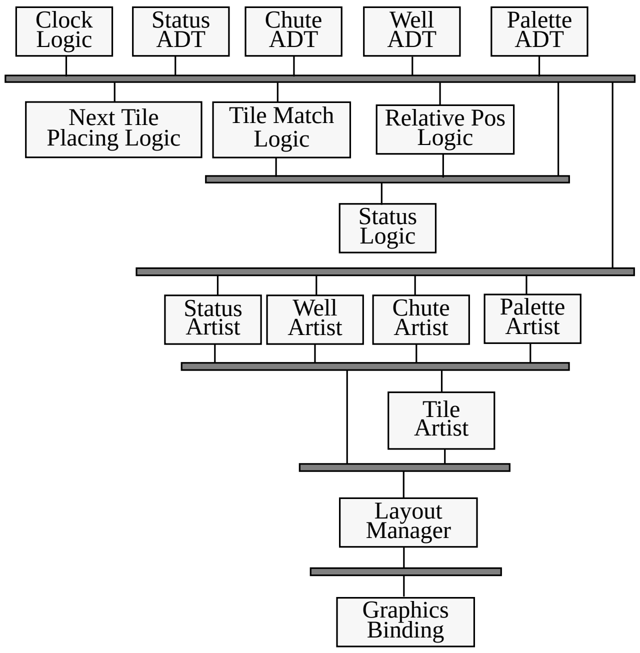


Fig. 4. Conceptual architecture for KLAX in the C2 style. Note that the Logic and Artist layers do not communicate directly and are in fact siblings. The Artist layer is shown below the Logic layer since the components in the Artist layer perform functions closer to the user.

2. KLAX is trademarked 1991 by Atari Games.

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

The KLAX architecture is intended to support a family of "falling-tile" games. The components were designed as reusable building blocks to support different game variations. One such variation on KLAX is shown in Fig. 5. This variation involved replacing the original tile matching, tile placing, and tile artist components with components which instead matched, placed, and displayed letters. The objective was thus transformed from matching the colors of tiles to spelling words. Each time a word was spelled correctly, it would be removed from the well. The *spelling logic* component wrapped an existing spell-checker.

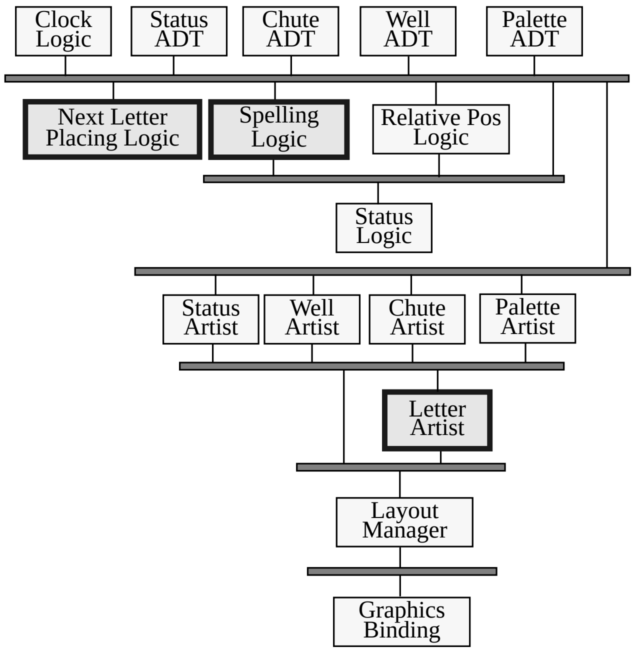


Fig. 5. A variation on KLAX. By replacing three components from the original architecture, the game turns into one whose object is to spell words horizontally, vertically, or diagonally.

MULTI-LEVEL PERSPECTIVE OF SOFTWARE ARCHITECTURES

Software systems built with a focus on architectures are complex, multi-dimensional entities that span four distinct levels of abstraction:

- specification of architectural elements,
- interfaces exported by the architectural elements,
- architectures (particular configurations of architectural elements), and
- architectural style rules (meta-architectures).

In order to develop a system using an architecture-based approach and fully realize the potential of a component-based development environment, a software architect needs to understand system characteristics at all four levels. Due to their common purpose, some overlap in motivating and describing the different levels is unavoidable.

Our discussion proceeds from the most familiar (component specification) to the newest (styles). To demonstrate the utility and applicability of this taxonomy, we discuss the ways in which an existing architectural style, pipe-and-filter, can be modeled at each level. With the exception of the component specification level, we also provide a model of C2 at each level.[3]

Several aspects of the C2 style, most notably its practicability, have been explored through the construction of several diverse experimental systems. We believe that our preliminary findings are encouraging. At the same time, we recognize that much additional work is needed on the style and supporting tools.

One of the facets that must be studied further is a formal representation of architectures built according to the style. The primary notation used to model individual C2 architectures thus far has been graphical ("boxes and arrows"). While its expressive power appears to be sufficient for this purpose, this notation has introduced ambiguities in interpreting architectures. Furthermore, its inherent imprecision makes the kinds of formal analyses of architectures that we would want to perform in the C2 design environment [25] unachievable. By producing a formal model at the four levels, while retaining the existing graphical notation, we intend to preserve the representational simplicity of architectures and the potential for their direct manipulation in the design environment, but also add the ability to formally reason about them.

Component Specification

A formal specification of a component's internal functionality is sought for three reasons. The first is the need to have rigorously verified components as reusable building blocks. In [3], Boehm and Scherlis state that the greatest disincentive against reuse is the perceived risk of using code developed by others. A small error in any one component can have disastrous effects in an architecture. In order to minimize this risk, components must be formally specified and verified.

The second reason for a formal specification of components is expressed in two of Krueger's truisms of software reuse [13]: to reuse an artifact effectively, one must know what it does and it must be easier to find it than to build it. A formal specification of a component can remedy both problems. While it requires a higher degree of rigor from a software practitioner than would a natural language description, modeling a component formally is a more precise, and often more concise, way of specifying its functionality. Furthermore, such specifications could assist in automating the task of locating components in a software component marketplace [31]. A developer would provide a specification of desired component properties to a locator application, which would, in turn, search both local and

3. We do not intend to conduct new research or propose new solutions in the area of component specification, since this is a well understood area of formal methods research. Instead, we intend to utilize existing formal techniques.

remote software component catalogs for components whose formal specifications most closely match the target specification. An approach for doing so with a local component repository is discussed in [18].

Finally, a formal specification of a component can be input to a code generation tool. Such a tool may be able to only partially generate component code. Furthermore, that code may need to be manually optimized to satisfy performance requirements. However, such a code generation process would be valuable regardless, as it would relieve human software developers of at least some programming responsibilities.

Examples of formal notations in which components can be specified are axiomatic notations, such as Anna [15], algebraic notations, such as Obj [11], or abstract model based notations, such as Z [26]. As a collection, component modeling techniques are the best understood and most mature of the formal modeling techniques available at the four levels of architectural abstraction. The reason for this is partly that these modeling formalisms predate the research in software architectures, and were developed with the intent of modeling procedures and source code modules.

Modeling Pipe-and-Filter Components
The Pipe-and-Filter architectural style, best known for its ability to create chains of input filters in the Unix shell, highlights the necessity of formal modeling of components. Central to the success of the Unix pipe-and-filter style are the robustness and correctness of commonly used filters, which results in a high level of user confidence in the filters and widespread filter reuse. Unix filters have typically not been formally specified, yet are quite robust, the result of thousands of hours of use over several decades. While long-term use is an effective way to yield robust components, this solution is not practical for new architectural styles; the use of formal modeling provides equivalent benefits over a far more compact time scale.

Interface Definition
Just as important as a component's abstract properties is the concrete interface a component makes available for accessing its functionality and data. Therefore, a formal specification of a component's interface is needed in addition to the specification of its internal functionality. Such a specification enables the component locator application, discussed above, to select the best possible match to the needs of a given architecture among components with similar or identical functionalities [30]. Candidate components whose interfaces are deemed too different from the desired one would hence be discarded. Furthermore, a formal interface definition affords automated translation of the selected component's domain in cases where a perfect match is not found. Finally, interface definition can help establish a component's adherence to the architectural style. For example, a candidate component for an architecture in the C2 style may not make direct procedure calls from its bottom side, as this would violate the principle of substrate independence.

Examples of interface definition languages (IDLs) are those used in CORBA [21] and Polylith [22]. Rapide [14] also provides a means for modeling component interfaces. Some common programming language constructs, such as Ada's package specifications, can also be viewed as IDLs. This is, therefore, a relatively well explored and understood area of research. Much of this work has been conducted independently of its application in software architectures and, as in the case of component specifications, it often predates research on architectures. For example, the architecture description language (ADL) LILEANNA [28] extends existing Ada constructs to enable architecture-based development in the language.

Both Rapide and LILEANNA are considered ADLs, yet they either provide new or incorporate existing facilities for component interface specification.[4] This combination of architecture and interface definition in a single language is relatively common. However, we believe that separating them provides a better conceptual understanding of the architectural level currently being modeled and affords the designer more power, as the most suitable IDL may be chosen independently of the ADL.

Modeling Pipe-and-Filter Component Interfaces
Due to the characteristics of the pipe-and-filter style, modeling the interfaces of filters is inappropriate. Since each filter is considered a single transform between its input and output data streams, there is only one entry point for each component. Similarly, there are always two parameters, the input and output data streams. However, just because formal interface modeling is unnecessary for the pipe-and-filter style, it should not be misinterpreted as an indictment of formality at this level of abstraction. Rather, it is the simplicity of the pipe-and-filter style itself and its use of untyped data streams to foster interoperability that makes filter interface modeling unnecessary.

Modeling C2 Component Interfaces
In this section, we provide a partial syntax of the prototype C2 IDL, pertinent to our discussion of modeling C2 component interfaces. The syntax is specified in BNF.[5] Furthermore, we give an example of how a KLAX component from Fig. 4 is modeled in the IDL.

A C2 component is modeled in the IDL as follows:

```
component ::=
    component component_name is
        interface component_message_interface
        parameters component_parameters
        methods component_methods
        [behavior component_behavior]
        [context component_context]
    end component_name;
```

A component's interface is specified as follows:

4. LILEANNA also provides facilities for specification of component semantics. It combines ANNA with the library-interconnection-language (LIL) [10], which uses Obj to specify component semantics.
5. In BNF, "**::=**" means "is defined as", "|" means "or", "**[...]**" denotes an optional item and "**{...}**" a repetitive item. For simplicity, single character terminals, such as a semicolon, are not surrounded by quotes. Spacing and indentation are used solely for readability.

```
component_message_interface ::=
    top_domain_interface
    bottom_domain_interface

top_domain_interface ::=
    top_domain is
        out interface_requests
        in interface_notifications

bottom_domain_interface ::=
    bottom_domain is
        out interface_notifications
        in interface_requests

interface_requests ::=
    {request;} | null;

interface_notifications ::=
    {notification;} | null;

request ::=
    message_name (request_parameters)

request_parameters ::=
    [to component_name] [parameter_list]

notification ::=
    message_name [parameter_list]
```

Component semantics may be specified by the interface to a limited degree. The manner in which the C2 IDL does so is shown below.

```
component_behavior ::=
    startup
    cleanup
    {internal_state_change | message_transition}
```

The most interesting aspect of component behavior as specified in the interface is *message_transition*:

```
message_transition ::=
    received_messages
        [invoked_methods] generated_messages

received_messages ::=
    received_messages
        [notification_sequence] [request_sequence]

invoked_methods ::=
    invoke_methods method_name {, method_name};

generated_messages ::=
    [message_generation_frequency request_sequence]
    [message_generation_frequency notification_sequence]

message_generation_frequency ::=
    always_generate | may_generate

request_sequence ::=
    request_name {logical_operator request_name};

notification_sequence ::=
    notification_name {logic_operator notification_name};
```

Given the above syntax, we can model components in a C2 architecture, such as KLAX. Component *WellArtist* from Fig. 4 is specified as follows:

```
component WellArtist is
    interface
        top_domain is
            out
                SendWellState (to WellADT);
            in
                TileAdded (l : location; t : tile_type);
                TilesAdvanced ();
                TilesRemoved (well : well_type);
                WellState (well : well_type);
                WellFull (l : location; t : tile_type);
        bottom_domain is
            out
                ViewportCreated (vport : vport_obj);
                ViewportDestroyed (vport : vport_obj);
                ObjCreated (object : draw_obj);
                ObjDestroyed (object : draw_obj);
            in
                null;
    parameters
        null;
    methods
        procedure Create_Viewport ();
        procedure Destroy_Viewport ();
        procedure Draw_Tile (row , col , tag: integer);
        procedure Erase_Tile (row , col : integer);
        procedure Redisplay_Well (wstate : well_type);
    behavior
        startup
            invoke_methods Create_Viewport;
            always_generate ViewportCreated, SendWellState;
        cleanup
            invoke_methods Destroy_Viewport;
            always_generate ViewportDestroyed;
        received_messages TileAdded;
            invoke_methods Draw_Tile;
            always_generate ObjCreated;
        received_messages TilesAdvanced;
            invoke_methods Draw_Tile, Erase_Tile;
            always_generate ObjCreated and ObjDestroyed;
        received_messages TilesRemoved;
            invoke_methods Erase_Tile;
            always_generate ObjDestroyed;
        received_messages WellState;
            invoke_methods Redisplay_Well;
            may_generate ObjCreated or ObjDestroyed;
        received_messages WellFull
            always_generate null;
    context
        internal artist;
end WellArtist;
```

The *WellArtist* is internal to the KLAX architecture, i.e., it is neither a top-most nor a bottom-most component in the architecture. The artist can issue only one request, *SendWellState*. This request is sent to *WellADT* at startup time, but may also be issued during the game to update the depiction of the KLAX well in the case of lost messages due to, e.g., network failure. The artist responds to multiple notifications in order to properly update the depiction of the well. The notifications it receives on its top side result in invocations of internal object's methods and, subsequently, generation of notifications on the artist's bottom side. Finally, the *WellArtist* does not respond to any requests issued by components below it.

The notation used in the example may appear long and complex at first. However, it is easy to recognize a number of patterns in the component's interface model. For example, "received_messages" is always followed by "invoke_methods" and either "always_generates" or "may_generate." Therefore, the designer's task can easily be simplified by providing component templates and appropriate editor support.

Note that the C2 IDL supports communication scenarios in which a sequence of received notifications and/or requests may result in a sequence of outgoing notifications and/or requests. The above example presents a special case where the lengths of all incoming and outgoing message sequences are one.

Architectural Description

The next level at which software architectures are described is that of actual architectural configurations, that is, architectural elements and connections among them. This information is needed to determine whether meaningful work will be performed by an architecture, i.e., whether the appropriate components are connected and whether their interfaces match. In concert with the previous two levels, architectural descriptions enable assessment of concurrent and distributed aspects of an architecture, such as the potential for deadlocks and starvation. An ADL must be able to express dynamic (run-time) changes to architectures as well. Furthermore, architectural description is necessary to establish adherence to architectural style rules, such as C2's rule that there are no direct communication paths between components. Finally, architectural descriptions enable analyses of architectures to determine whether an architecture is "too deep," which may affect performance due to message traffic across many levels and/or process splits, or "too broad," which may result in too many dependencies among components (a "component soup" architecture).

Much research has recently been done on ADLs in the software architectures community. There are a number of existing ADLs. Examples are Rapide [14], MetaH [29] and Wright [2]. However, there is still no consensus on what an ADL is and is not and what aspects of an architecture should be modeled by an ADL. For example, Rapide is a general-purpose system description language that allows modeling of component interfaces and their externally visible behavior in addition to modeling the architecture, while Wright formalizes the semantics of architectural connections with a communicating-sequential-processes-like notation [12]. Furthermore, no distinction is made between ADLs on one hand and formal specification languages, simulation languages, and programming languages on the other. Instead, for example, Rapide can be viewed as both an ADL and a simulation language. Such ambiguity indicates that ADLs are not as well understood as formalisms at either of the preceding two levels (component semantics and interfaces).

Modeling Pipe-and-Filter Architectures

The Unix shell employs a simple, effective ADL for describing filter interconnections. The ADL consists of a variable set of filter names (the set of all filters in the user's path), an interconnection operator (a bar "|", pictorially representing a pipe), and a small set of redirection operators (the greater than and less than signs, ">", "<", redirecting output and input respectively). Along with a small number of syntax rules, the combination of filters and operators allows the specification of all Unix pipe-and-filter architectures. These architectures are easy to understand due to the simplicity of the formalism, and they are easy to analyze since component interconnections are so visible.

While the Unix shell ADL for pipe-and-filter cannot be used to model other architectural styles, it does highlight the benefit of small, simple, formalisms highly tailored to a specific architectural style. This is in contrast to more powerful, and also more complex ADLs such as Rapide.

Modeling C2 Architectures

C2's ADL is a means by which C2 architectures are "programmed." The ADL is a programming-language independent modeling technique that specifies the instantiation of required architectural elements and their interconnections. In this section, we provide a partial syntax of the prototype C2 ADL, pertinent to our discussion of modeling C2 architectures, specified in BNF (see Footnote 5). Furthermore, as an example, we model the KLAX architecture from Fig. 4 in the ADL.

A C2 architecture is modeled in the ADL as follows:

```
architecture ::=
    architecture architecture_name is
        components component_list
        component_instances component_instance_list
        [connectors connector_list]
        [architectural_topology topology]
    end architecture_name;
component_list ::=
    top_most {component_name;}
    internal {component_name;}
    bottom_most {component_name;}
component_instance_list ::=
    instance_name instantiates component_name
        [with (parameter_instantiation)];
    {instance_name instantiates component_name
        [with (parameter_instantiation)];}
connector_list ::=
    {connector;}
connector ::=
    connector connector_name is
        message_filter message_filter_type;
    end connector_name;
message_filter_type ::=
    no_filtering | notification_filtering | prioritized | msg_sink
```

A connector with the *no_filtering* policy becomes a simple message router. *Notification_filtering* is a means of message registration in C2. *Prioritized* message broadcast allows a connector to define a priority ranking over its connected components whereby a notification is sent to each component in order of priority until a termination condition has been met. Finally, the *msg_sink* filtering policy allows a connector to ignore each message sent to it.[6]

The topology of a C2 architecture is specified as follows:

```
topology ::=
    {connector connector_name connections
        top_ports connection_sequence
        bottom_ports connection_sequence}

connection_sequence ::=
    connection; {connection;} | null;

connection ::=
    connection_block_name
        [where [not] connection_constraint]
        [message_filter port_message_filter_type]

connection_block_name ::=
    component_instance_name | connector_name

connection_constraint ::=
    boolean_expr | msg_sequence | environment_command
```

Connection constraints provide a means for dynamically changing architectures, as will be demonstrated in the KLAX example below. However, these dynamic changes must be planned for ahead of time and specified at architecture definition time. An open research issue is what facilities are needed to handle unplanned dynamic changes to architectures and whether there are ways of enabling such changes by the ADL.

6. [27] provides a detailed explanation of filtering policies in C2.

For purposes of brevity and clarity, only a partial model of the KLAX architecture is given below:

```
architecture KLAX is
    components
        top_most
            ClockLogic;
            StatusADT;
            ...
        internal
            NextTilePlacingLogic;
            LayoutManager;
            ...
        bottom_most
            GraphicsBinding;
    component_instances
        ClockLogic_1 instantiates ClockLogic;
        StatusADT_1 instantiates StatusADT;
        ...
    connectors
        connector Conn1 is
            message_filter no_filtering;
        end Conn1;
        ...
    architectural_topology
        connector Conn1 connections
            top_ports
                ClockLogic_1;
                StatusADT_1;
                ChuteADT_1;
                WellADT_1;
                PaletteADT_1;
            bottom_ports
                NextTilePlacingLogic_1
                    where not Ctrl-S;
                NextLetterPlacingLogic_1
                    where Ctrl-S;
                TileMatchLogic_1
                    where not Ctrl-S;
                SpellingLogic_1
                    where Ctrl-S;
                RelativePositionLogic_1;
                Conn2;
                Conn3
                ...
end KLAX;
```

Note that "*Ctrl-S*" represents an example of the *environment_command* connection constraint. This example depicts the dynamic substitution of two of three components used in building "spelling KLAX" from the original application, as shown in Fig. 5.

Architectural Style Rules

It is not necessary for software architectures to adhere to the rules of a particular style in order to perform meaningful work. However, adhering to a style carries certain benefits with it. Architectural styles are formulated and evolved to consistently repeat successes and avoid failures from previous projects. A style also enables software architects to reuse successful design patterns [6]. A formal definition of the style enables a design environment to check an architecture for conformity to style rules. A formal definition of the style may also enable automatic generation of templates for components, connectors, domain translators, main procedures, etc. Finally, a formal definition may highlight those characteristics of the style that can be modified to create a family of architectural styles suited for different application domains.

The goal of the C2 style in particular is to enable development of GUI architectures with interchangeable and reusable software components. The components are heterogeneous and must not depend on their underlying technologies (other components or the operating system). Software systems for which the style is intended are real-time, distributed, and concurrent. C2 style's requirements and intended application domain are nontrivial. Therefore, we must enable automated analysis of architectures built in the style to ensure their conformance to the style, as well as to the application-specific requirements. The formal definition of the style is necessitated by the fact that a solid formal foundation is the basis of analysis.

As a research area, architectural styles are more recent than the three preceding areas. Analogously, the techniques available for formal modeling of architectural styles are less mature than those at the other levels of architectural abstraction. [1] and [8] present rare attempts at formally modeling styles. The formal definition of the C2 style, given below, is based [1]'s approach.

Modeling the Pipe-and-Filter Style

A formal model of the pipe-and-filter style is presented in [1], using the Z notation to describe the style in terms of three basic syntactic classes for architectural modeling: components, connectors, and their configurations. In addition to demonstrating that formal modeling of architectural style is possible, [1] also shows its benefits: precise description of the architectural style in unambiguous language, development of specialized analysis techniques, and the ability to make comparisons between architectural styles. Similar to this specification is the formal specification of the style provided in [7], which, when input to the Aesop system, yields an automatically generated environment for the development of architectures in the style. This demonstrates the utility of formal specification of architectural style for the construction of architectural design environments.

Modeling the C2 Style

This section presents a formal definition of major features of the C2 architectural style, the fourth level of architectural abstraction. The complete formal model of the style is given in [16].[7] Coupled with the formal models at the preceding three abstraction levels, this work represents the important first step in providing a comprehensive formal basis for the style and enabling the design environment to perform the necessary analyses of architectures.

A canonical C2 component is defined below. This definition corresponds to the internal architecture of a component shown in Fig. 2. Since a component's dialog can decide when and whether to handle a particular message (or sequence of messages) that it receives at its top and bottom ports [27], the *msg_to_handle* function is defined to select one or more messages at a port. A *state_transition* in a component, whose properties are specified in the last formula in the schema, is defined as processing messages received at either the top or the bottom port and possibly generating outgoing messages. For each incoming message it processes, a component may generate multiple outgoing messages at each port.

7. The formal definition of C2 is given in Z [26]. See the Appendix for a summary of Z concepts and terminology.

$$
\begin{array}{l}
\textit{C2Component} \\
\hline
name : COMP\_NAME \\
top\_port,\ bot\_port : PORT \\
top\_in,\ top\_out,\ bot\_in,\ bot\_out : PORT \nrightarrow \mathbb{P}\ MSG \\
top\_domain,\ bot\_domain : \mathbb{P}\ MSG \\
dialog\_in : \mathbb{P}\ MSG \nrightarrow OBJ\_STATE \\
wrapper,\ dialog\_top\_out : OBJ\_STATE \nrightarrow \mathbb{P}\ MSG \\
msg\_to\_handle : NEXT\_MSG; \\
domain\_trans : \mathbb{P}\ MSG \nrightarrow \mathbb{P}\ MSG \\
internal\_states : \mathbb{P}\ OBJ\_STATE \\
start\_state : OBJ\_STATE \\
state\_transitions : (OBJ\_STATE \times (PORT \nrightarrow \mathbb{P}\ MSG)) \rightarrow \\
\quad (OBJ\_STATE \times \{\ (PORT \nrightarrow \mathbb{P}\ MSG),\ (PORT \nrightarrow \mathbb{P}\ MSG)\ \}) \\
\hline
top\_port \neq bot\_port \\
\\
\operatorname{dom} top\_in = \{\ top\_port\ \} \\
\operatorname{dom} top\_out = \{\ top\_port\ \} \\
\operatorname{dom} bot\_in = \{\ bot\_port\ \} \\
\operatorname{dom} bot\_out = \{\ bot\_port\ \} \\
\\
top\_domain = top\_in(top\_port) \cup top\_out(top\_port) \\
bot\_domain = bot\_in(bot\_port) \cup bot\_out(bot\_port) \\
\\
\forall state1,\ state2 : OBJ\_STATE;\ ps1,\ ps2,\ ps3 : PORT \nrightarrow \mathbb{P}\ MSG \\
\ \bullet\ ((state1,\ ps1),(state2,\ \{\ ps2,\ ps3\ \})) \in state\_transitions \Rightarrow \\
\qquad\qquad state1 \in internal\_states \\
\qquad \wedge \quad state2 \in internal\_states \\
\qquad \wedge \quad (\operatorname{dom} ps1 = \{\ top\_port\ \} \vee \\
\qquad\qquad \operatorname{dom} ps1 = \{\ bot\_port\ \}) \\
\qquad \wedge \quad (\operatorname{dom} ps2 = \{\ top\_port\ \}) \\
\qquad \wedge \quad (\operatorname{dom} ps3 = \{\ bot\_port\ \}) \\
\qquad \wedge \quad (ps1(top\_port) \subseteq top\_in(top\_port) \vee \\
\qquad\qquad ps1(bot\_port) \subseteq bot\_in(bot\_port)) \\
\qquad \wedge \quad ps2(top\_port) \subseteq top\_out(top\_port) \\
\qquad \wedge \quad ps3(bot\_port) \subseteq bot\_out(bot\_port)
\end{array}
$$

Note that all definitions involving components assume that they are internal components, i.e., they are neither top- nor bottom-most in an architecture. However, the top- and bottom-most components are easily described as special cases of the given definitions by omitting from the schemas references to their sides, top or bottom, that are outermost in an architecture.

The following schema expresses substrate independence. A component must utilize the domain translator for the messages it both receives and sends on its top side. At the same time, it has no knowledge and makes no assumptions about its substrate, so that the wrapper around the internal object emits messages in the component's domain of discourse on its bottom side, unbeknownst to the dialog.

$$
\begin{array}{l}
\textit{HandleMessageFromAbove} \\
\hline
\Delta C2ComponentState \\
\hline
comp' = comp \\
\\
((current\_state,\ top\_in\_data), \\
\ (current\_state',\ \{\ top\_out\_data',\ bot\_out\_data'\ \})) \in \\
\qquad comp.state\_transitions \\
\\
top\_out\_data'(comp.top\_port) = \\
\qquad top\_out\_data(comp.top\_port) \cup \\
\qquad comp.domain\_trans(\\
\qquad\quad comp.dialog\_top\_out(\\
\qquad\qquad comp.dialog\_in(\\
\qquad\qquad\quad comp.domain\_trans(\\
\qquad\qquad\qquad comp.msg\_to\_handle(\\
\qquad\qquad\qquad\quad top\_in\_data(comp.top\_port)))))) \\
\\
bot\_out\_data'(comp.bot\_port) = \\
\qquad bot\_out\_data(comp.bot\_port) \cup \\
\qquad comp.wrapper(\\
\qquad\quad comp.dialog\_in(\\
\qquad\qquad comp.domain\_trans(\\
\qquad\qquad\quad comp.msg\_to\_handle(\\
\qquad\qquad\qquad top\_in\_data(comp.top\_port))))) \\
\\
top\_in\_data(comp.top\_port) = \\
\qquad top\_in\_data'(comp.top\_port) \cup \\
\qquad comp.msg\_to\_handle(top\_in\_data(comp.top\_port)) \\
\\
bot\_in\_data'(comp.bot\_port) = bot\_in\_data(comp.bot\_port)
\end{array}
$$

For clarity, the expressions defining the new values for *top_out_data* and *bot_out_data* above (denoted with "**′**") have been broken across several lines. Going from the bottom of each expression upward, every line represents a step in processing messages from selecting a sequence of incoming messages to producing outgoing messages. For example, *top_out_data* **′** is obtained by the following five steps:

1. select a set of incoming messages from the top_port: *comp.msg_to_handle(top_in_data(...)),*
2. perform domain translation on those messages: *comp.domain_trans (1),*
3. interpret the translated messages in the dialog and invoke the appropriate internal object methods: *comp.dialog_in(2),*
4. interpret the values returned by the internal object's methods and generate a set of outgoing messages: *comp.dialog_top_out(3),* and
5. perform domain translation on the outgoing messages: *comp.domain_trans(4).*

A C2 connector has multiple ports on its top and bottom sides, one for each component attached to it. It is defined as follows:

$$\begin{array}{l}
\textit{C2Connector} \\
\hline
top\_ports,\ bot\_ports : \mathbb{P}\ PORT \\
top\_in,\ top\_out,\ bot\_in,\ bot\_out : PORT \rightarrow\!\!\!\!\!+ \ \mathbb{P}\ MSG \\
Filter\_TB : FILTER \\
Filter\_BT : FILTER \\
\hline
top\_ports \cap bot\_ports = \emptyset \\
\\
\mathrm{dom}\ top\_in = top\_ports \\
\mathrm{dom}\ top\_out = top\_ports \\
\mathrm{dom}\ bot\_in = bot\_ports \\
\mathrm{dom}\ bot\_out = bot\_ports \\
\\
\bigcup(\mathrm{ran}\ bot\_out) \subseteq \bigcup(\mathrm{ran}\ top\_in) \\
\bigcup(\mathrm{ran}\ top\_out) \subseteq \bigcup(\mathrm{ran}\ bot\_in) \\
\hline
\end{array}$$

A connector may have the ability to filter messages, so that the messages it emits on its bottom side are a subset of those that come in from above and the messages it emits on its top side are a subset of those that come in from below (See "Architectural Description"). It is thus possible to define filtering functions *Filter_TB* and *Filter_BT* that determine for each port whether a particular message will be filtered out or propagated. The below schema shows how the *Filter_TB* function is used to decide whether a given message *msg* is filtered out or propagated to a bottom port *port2*. For simplicity, the two functions are assumed to filter out a message by propagating a null message.

$$\begin{array}{l}
\textit{RoutMessageFromAbove} \\
\hline
\Delta C2ConnectorState \\
\hline
conn' = conn \\
\\
\forall\, msg : MSG;\ port1 : conn.top\_ports \mid msg \in top\_in\_flow(port1) \\
\quad \bullet\ \forall\, port2 : conn.bot\_ports \\
\qquad \bullet\quad top\_in\_flow(port1) = top\_in\_flow'(port1) \cup \{\ msg\ \} \\
\qquad\quad \wedge\ \ top\_out\_flow'(port1) = top\_out\_flow(port1) \\
\qquad\quad \wedge\ \ bot\_in\_flow'(port2) = bot\_in\_flow(port2) \\
\qquad\quad \wedge\ \ bot\_out\_flow'(port2) = \\
\qquad\qquad\qquad bot\_out\_flow(port2) \cup \{\ conn.Filter\_TB(port2,\ msg)\ \} \\
\hline
\end{array}$$

The property that a component may only be attached to single connectors on its top and bottom sides is expressed as follows:

$$\begin{array}{l}
\textit{ComponentToConnectorLinks} \\
\hline
C2Link \\
components : \mathbb{P}\ C2Component \\
connectors : \mathbb{P}\ C2Connector \\
\hline
\forall\, comp : components \\
\quad \bullet\ \exists_1\, conn1,\ conn2 : connectors;\ tport,\ bport : PORT \mid \\
\qquad\quad tport \in conn2.top\_ports \\
\qquad \wedge\ \ bport \in conn1.bot\_ports \\
\qquad \wedge\ \ conn1 \neq conn2 \\
\qquad\qquad \bullet\ (comp.top\_port,\ bport) \in Link\ \wedge \\
\qquad\qquad\quad (comp.bot\_port,\ tport) \in Link \\
\hline
\end{array}$$

Finally, in any given architecture, there is no guarantee that all of a component's services will be utilized by components above and below it or that the component will understand all the requests sent to it. This property is a byproduct of component reusability. A component may be used in multiple architectures, and different aspects of it may be needed in each. Since components communicate via connectors, it is possible to specify pairwise relationships between the domains of any connector and each component attached to it, and express the utilization of a component's services in terms of that relationship. For example, partial utilization of a component's services by a connector, where the component will receive some, but not all, of the messages it is capable of handling, is defined as follows:

$$\begin{array}{l}
\textit{PartialServiceUtilization} \\
\hline
vc : ValidC2Connections \\
components : \mathbb{P}\ C2Component \\
connectors : \mathbb{P}\ C2Connector \\
\hline
\forall\, c : components;\ b : connectors;\ bport : PORT \\
\quad \bullet\ \ (bport \in b.top\_ports \wedge (c.bot\_port,\ bport) \in vc.Link \Rightarrow \\
\qquad\quad b.top\_out(bport) \cap c.bot\_in(c.bot\_port) \neq \emptyset\ \wedge \\
\qquad\quad c.bot\_in(c.bot\_port) \cap b.top\_out(bport) \subset c.bot\_in(c.bot\_port)) \\
\quad\ \wedge\ (bport \in b.bot\_ports \wedge (c.top\_port,\ bport) \in vc.Link \Rightarrow \\
\qquad\quad b.bot\_out(bport) \cap c.top\_in(c.top\_port) \neq \emptyset\ \wedge \\
\qquad\quad c.top\_in(c.top\_port) \cap b.bot\_out(bport) \subset c.top\_in(c.top\_port)) \\
\hline
\end{array}$$

VALUE OF THE C2 MODELING FORMALISMS

There are a number of benefits incurred from the interface, architecture, and style modeling formalisms for C2, discussed in the previous section. At the same time, the formalisms fall short in certain respects and a number of issues remain unresolved.

The C2 IDL and ADL were used successfully as the design notation in the KLAX project, described in "An example C2 application". In projects preceding KLAX, a crossection of which is discussed in [27], no particular design notation was used. This usually resulted in a mixture of annotated boxes-and-arrows diagrams and prose. The designs were difficult to understand or update in this form and both evolvability and scalability of these systems suffered.

The IDL and ADL were, therefore, a needed complement to these descriptions. The two languages provide structure to the design process and are simple and easy to understand. They are also easily extensible to accommodate new understanding of C2 concepts. Finally, a notation equivalent to a subset of the ADL was used in Argo, the C2 design environment [25], to automatically generate main procedures for the different variations of KLAX.

The current shortcomings of the IDL and ADL are largely an indicator of future work. We currently do not support syntactic or semantic checking of the two languages in Argo. These facilities are precursors to further architectural analysis and code generation tools. We also recognize the potential benefits of treating collections of C2 components as object-oriented (OO) type frameworks. We have completed some preliminary work on identifying the issues and extending the ADL syntax to accommodate the view of C2 architectures as a network of OO types [17]. However, we still need to develop a deeper understanding of the ramifications of doing so and provide a complete language and toolset to support subtyping and type checking in C2.

The formal definition of the C2 style also has several benefits. The definition fulfills its primary role: it enables enforcement of style rules, such as substrate independence, single top and bottom component ports, multiple connector ports, and connection of the bottom of a component to the top of a connector, and vice versa. Furthermore, the template for components in the C2 style, shown in Fig. 2 and defined in the schema *C2Component*, may be generated from the model. The model also formalizes the notions of full, partial, and no service utilization between components in an architecture, enabling the design environment to pinpoint components that require domain translation. The definition is flexible; it can be easily modified to reflect any new knowledge we acquire about the style by adding or modifying the necessary Z schemas. For example, the definition can be easily extended to include implementation characteristics of architectures, such as operating system, host, process, task, and address space.

The formal model also enabled our better understanding of the style. Merely going through the process of adding rigor to the definition of a style is a verification process. It crystallized our understanding of what the C2 style is and exposed several inconsistencies and inaccuracies that existed in its less formal definition. In particular, formalizing C2 corrected a mistake in the prior definition of service utilization axioms. It, furthermore, affirmed the suspected need for the explicit representation of communication links, or paths, between individual component and connector ports. Finally, it clarified the definition of communication between a component and a connector as being on individual port-to-port basis, as opposed to component-to-entire-connector communication, as was previously expressed incorrectly.

On the other hand, the style definition has several shortcomings. Many of them may be attributed to the formal notation we chose.

The style definition may be viewed as cumbersome and difficult to read. Z’s rigor and insistence on completeness, which are the strengths of any formal notation, hamper understandability in this case. For example, the formula in the *C2Component* schema that describes the properties of a *state_transition* is very long and may require careful examination. Yet, it was informally described simply as “processing messages received at either the top or the bottom port and possibly generating outgoing messages.” Furthermore, the property that a component may only be attached to single connectors above and below it, specified in the *ComponentToConnectorLinks* schema, is currently expressed in Argo with a single *if* statement. Several other style rules are also expressed more simply algorithmically in Argo than they are in Z. Therefore, an operational notation seems preferable to a model-based one for at least some aspects of C2.

Another shortcoming of Z is that some C2 style rules cannot be defined in it. For example, Z is inadequate for C2’s real-time, concurrent, and distributed aspects. These must be specified using a different formal method, such as temporal logic, in a notation such as graphical interval logic (GIL) [4].

[1] demonstrated Z’s utility in modeling two architectural styles, pipe-and-filter and the event system (ES). However, these two styles are substantially simpler than C2. It can even be argued that ES is subsumed by C2. This does not contradict our claim that C2 is a simple style. On the contrary, we believe that C2 is easy to understand conceptually and any attempts to simplify the Z definition of the style would ultimately decrease its understandability. For example, it is possible to eliminate the topology imposed by C2 and reduce C2 to a centralized-bus style, such as Field [24]. The topology would then become implicit in the single connector’s message routing and filtering logic. This would substantially simplify the corresponding Z schemas, but the conceptual understanding of the style would be greatly diminished. These shortcomings of Z imply that, just as there are multiple formalisms for modeling components, interfaces, and architectures, a single formal notation is insufficient for modeling all styles.

CONCLUSION

Constructing systems on the basis of a software architecture approach offers many benefits. Since many of these benefits rely on the ability to analyze a system and its constituents, there are clear and important roles for formal models to play. We have discussed four levels of abstraction, meaningful in our development strategy, at which formal specifications are necessary. All four levels are not equally well understood and our discussion of modeling techniques establishes a spectrum of consensus in the research community, going from best understood (component semantics) to least understood (architectural styles).

We have been pleased to be able to successfully model C2’s component interfaces, architectures, and style rules, and have obtained benefits in so doing. At the same time, our formal models of component interfaces and architectures are currently only prototypes, while the formal definition of the style, expressed in Z, has proven both unnecessarily complex and inadequate in formalizing certain aspects of architectures developed according to the style. Additional formal specification techniques are required to model C2’s real-time, distributed, and concurrent characteristics. Furthermore, complete tool support for the four levels of abstraction is needed as part of Argo to facilitate architecture-based development in C2.

We are dismayed by the prospects of having to use several different formal specification techniques, sometimes even at a single level of architectural abstraction, to address all our formalization needs. While we are not so naive as to think that there is soon going to be one technique which is “good for everything,” we certainly hope to direct the attention of formal methods researchers to the problems of utilizing heterogeneous specification mechanisms, particularly in the area of software architectures.

ACKNOWLEDGEMENTS
We would like to acknowledge K. Anderson, J. Robbins, K. Nies, P. Oreizy, and D. Dubrow for their contribution on various aspects of C2. We also thank D. Richardson, K. Anderson, J. Robbins, and H. Yessayan for their insightful comments on earlier drafts of the paper. Finally, we thank the CSS reviewers for their helpful reviews and especially S. Sankar for his assistance in revising the paper.

REFERENCES

1. Abowd, G., Allen, R., and Garlan., D. Using style to understand descriptions of software architecture. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 9-20, Los Angeles, CA, December 1993.
2. Allen, R. and Garlan, D. Formal connectors. Technical Report CMU-CS-94-115, Department of Computer Science, Carnegie Mellon University, March 1994.
3. Boehm, B. W. and Scherlis, W. L. Megaprogramming. In *Proceedings of the Software Technology Conference 1992, pages 63-82*, Los Angeles, April 1992. DARPA.
4. Dillon, L. K., Kutty, G., Melliar-Smith, M. P., Moser, L. E., and Ramakrishna, Y. S. Graphical specifications for concurrent *software systems. In Proceedings of the Fourteenth International Conference on Software Engineering*, pages 214-224, Melbourne, Australia, May 1992.
5. Fischer, G., Girgensonh, A., Nakakoji, K., and Redmiles, D. Supporting software designers with integrated domain-oriented design environments. IEEE Transactions on Software Engineering, pages 511-522, June 1992.
6. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, Massachusetts, 1995.
7. Garlan, D., Allen, R., and Ockerbloom, J. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175–188, New Orleans, LA, USA, December 1994.
8. Garlan, D. and Notkin, D. Formalizing design spaces: implicit invocation mechanisms. *In VDM'91: Formal Software Development Methods,* Noordwijkerhout, The Netherlands, October 1991, Springer-Verlag, Pages 31-44.
9. Garlan, D. and Shaw, M. *An introduction to software architecture: advances in software engineering and knowledge engineering,* volume I. World Scientific Publishing, 1993.
10. Goguen, J. A. Reusing and interconnecting software components. *IEEE Computer*, pages 16-28, February 1986.
11. Goguen, J. A. and Winkler, T. Introducing OBJ3. Technical Report SRI-CSL-88-99. SRI International 1988.
12. Hoare, C. A. R. *Communicating sequential processes*. Prentice Hall, 1985.
13. Krueger, C. W. *Software reuse*. Computing Surveys, pages 131-184, June 1992.
14. Luckham, D. and Vera, J. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, pages 717-734, September 1995.
15. Luckham, D. *ANNA, a language for annotating Ada programs: reference manual*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1987.
16. Medvidovic, N. Formal definition of the chiron-2 software architectural style. UCI-ICS Technical Report UCI-95-24, University of California, Irvine, July 1995.
17. Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. Using object-oriented typing to support architectural design in the C2 style. Submitted to *The ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'96), February 1996.*
18. Monroe, R. T. and Garlan, D. Style-based reuse for software architectures. In *Proceedings of the Fourth International Conference on Software Reuse*, April 1996.
19. Moorman, A. Z. and Wing J. M. Signature matching: a key to reuse. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, pages 182-190, Los Angeles, CA, USA, December 1993.
20. Moriconi, M., Qian, X., and Riemenschneider, R. A. Correct architecture refinement. *IEEE Transactions on Software Engineering*, pages 356-372, April 1995.
21. Orfali, R., Harkey, D., and Edwards, J. *The essential distributed objects survival guide*. John Wiley & Sons, Inc., 1996.
22. Purtillo, J. M. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*, pages 151-174, January 1994.
23. Perry, D. E. and Wolf, A. L. Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes*, pages 40-52, October 1992.
24. Reiss, S. P. Connecting tools using message passing in the Field environment. *IEEE Software*, pages 57-66, July 1990.
25. Robbins, J. E. and Redmiles, D. 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.
26. Spivey, J. M. *The Z notation: a reference manual*. Prentice Hall, New York, 1989.
27. Taylor, R. N., Medvidovic, N, Anderson, K. M., Whitehead, Jr., E. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering,* to appear.
28. Tracz, W. LILEANNA: a parameterized programming language. In *Proceedings of the 2nd International Workshop on Software Reuse*, March 1993.
29. Vestal, S. *Software programmer's manual for the Honeywell Aerospace compiled kernel (MetaH language refer-*

ence manual). Honeywell systems and Research Center, March 1993.

30. Wing, J. A specifier's introduction to formal methods. *IEEE Computer*, September 1990.

31. Whitehead, Jr., E. J., Robbins, J. E., Medvidovic, N., and Taylor, R. N. Software architectures: foundation of a software component marketplace. In David Garlan, editor, *Proceedings of the First International Workshop on Architectures for Software Systems*, pages 276-282, April 1995.

APPENDIX: Summary of the Z Notation

The Z notation is a language for modeling mathematical objects developed at the Programming Research Group at the University of Oxford. Z is based on first order logic and set theory. It uses standard logical connectives ($\vee$, $\wedge$, $\Rightarrow$, etc.) and set-theoretic operations ($\in$, $\cup$, $\cap$, etc.) with their standard semantics. In this appendix, we outline the aspects of the Z notation used in this paper. A more extensive summary of Z can be found in [1]. For a complete reference, see [26].

A Z specification is a collection of types and predicates that must hold on the types' values. Z provides basic types, such as $\mathbb{N}$ for natural numbers and $\mathbb{Z}$ for integers. Other basic types can be introduced by enclosing them in square brackets. For example, the types for person names and addresses are specified as follows:

[*NAME*, *ADDRESS*]

To declare that a particular *person* is of type *NAME*, we write *person : NAME*. If *person* has already been declared, the above predicate is expressed as $person \in NAME$.

Composite types in Z are constructed from basic types using the following type constructors:

- $\mathbb{P}X$ is the powerset of *X*, i.e., the set of all subsets of *X*.
- $X \times Y$ is the cross-product of *X* and *Y*, i.e., a set of all ordered pairs *(x,y)* such that $x \in X$ and $y \in Y$.
- Functions used in this paper are:
  - $X \mathrel{+\!\!\!\rightarrow} Y$, the set of all partial functions between *X* and *Y*. A partial function need not be defined over the entire domain, and
  - $X \rightarrow Y$, the set of all total functions. Total functions are defined on all elements of the domain type.

An abbreviation or type synonym in Z allows introduction of new global constants. For example, a function that returns the names of all people residing at a given address is defined as:

$INHABITANTS == ADDRESS \mathrel{+\!\!\!\rightarrow} \mathbb{P}NAME$

Other Z operations and notational conventions used in this paper are:

- If *f* is a function, then dom *f* is the domain of *f* and ran *f* is the range of *f*.
- $\forall decl \mid pred_1 \bullet pred_2$ is read "for all variables in *decl* satisfying $pred_1$, we have that $pred_2$ holds."
- $\exists decl \mid pred_1 \bullet pred_2$ is read "there exist variables in *decl* satisfying $pred_1$, such that $pred_2$ holds."

Z has a special type constructor, called the *schema*. A schema is a collection of variables with a set of constraints over that collection. For example, *Town* is a schema for a town with the set of residences and people residing in them:

```
─ Town ──────────────────────────────
│ residences : ℙ ADDRESS
│ residents : INHABITANTS
└────────────────────────────────────
```

To select the residents of *t* : *Town*, we write *t.residents*.

A schema can also specify invariants, written under the dividing line, that must hold between the values of variables. To model the invariant that the set of *residents* in type *Town* includes only those whose residence is in the given *Town*, we state that *residences* is the domain of the *residents* function.

```
─ SingleTown ────────────────────────
│ residences : ℙ ADDRESS
│ residents : INHABITANTS
├──────────
│ residences = dom residents
└────────────────────────────────────
```

Z allows for schema inclusion to facilitate a more modular approach to specification. The invariant above can also be specified as

```
─ SingleTown ────────────────────────
│ Town
├──────────
│ residences = dom residents
└────────────────────────────────────
```

Finally, f *Schema* is a schema type, then Δ*Schema* represents two *Schema* states, one before and the other after an operation. The state after the operation is denoted with "**′**". Hence,

```
─ TownGrowth ────────────────────────
│ ΔTown
└────────────────────────────────────
```

is equivalent to

```
─ TownGrowth ────────────────────────
│ Town
│ Town′
└────────────────────────────────────
```