UNIVERSITY OF CALIFORNIA,
IRVINE

Architecture-Based
Specification-Time Software Evolution

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Nenad Medvidovic

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor David Rosenblum
Professor David Redmiles

1999

ABSTRACT OF THE DISSERTATION

Architecture-Based
Specification-Time Software Evolution

by

Nenad Medvidovic

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1999

Professor Richard N. Taylor, Chair

Software architectures shift the focus of developers from lines-of-code to coarser-grained architectural elements and their overall interconnection structure. Architectures have the potential to substantially improve the development and evolution of large, complex, multi-lingual, multi-platform, long-running systems. In order to achieve this potential, specific architecture-based modeling, analysis, and evolution techniques must be provided. To date, software architecture research has produced an abundance of techniques for architecture modeling and analysis, while largely neglecting architecture-based evolution.

This dissertation motivates, presents, and validates a methodology for software evolution at architecture specification-time. The methodology consists of a collection of techniques that, individually and in concert, support flexible, systematic evolution of software architectures in a manner that preserves the desired architectural relationships and properties. The methodology is comprehensive in its scope: it addresses the evolution of individual architectural building blocks—components and connectors—as well as entire architectures; it also supports the transfer of (evolved) architecture-level decisions into implemented systems. The unique aspects of the methodology are: *component evolution* via heterogeneous subtyping, well suited to a wide range of design and reuse circumstances; *connector evolution*, facilitated by evolvable interfaces and heterogeneous communication protocols; *architecture evolution*, facilitated by minimal component interdependencies and heterogeneous, flexible connectors; *analysis* of architectures for consistency, where the architect possesses the authority to override the analysis tool; off-the-shelf component and connector *reuse*, necessary for economic viability in large-scale software development; and *implementation generation*, aided by a well-bounded implementation space and accomplished via a component-based, evolvable environment.

The dissertation is validated empirically, by constructing a series of demonstration applications, and analytically, by evaluating the manner and degree to which the applications validate the claims of the dissertation. The dissertation is concluded by examining its impact on the tension between flexibility and formality, which characterizes current software architecture research.

CHAPTER 1: Introduction

Software has become an integral part of life, so ubiquitous that its presence often goes unnoticed. It is critical to supporting and enabling from the most basic of everyday activities to the unprecedented achievements in science, industry, and military technology. The increasing need for and importance of software is also reflected in the explosive growth of the software industry. As our dependence on software and understanding of its potential grow, so do the demands on software engineers to build larger, more complex systems, constructed from heterogeneous parts, which must execute on multiple computing platforms and provide uninterrupted service.

The ability to satisfy such demands depends on a large number of factors, the study of which forms the underpinnings of software engineering research. One such factor is the ability to evolve existing systems in response to changing requirements. The costs of system maintenance (i.e., evolution) are commonly estimated to be as high as 60% of the overall development costs [28]. Practitioners have traditionally faced many problems with curbing these costs. The problems are often the result of poor understanding of a system's overall architecture, unintended and complex dependencies among its components, decisions that are made too early in the development process, and so forth. These problems are only exacerbated in the case of large, complex, multi-lingual, multi-platform, long-running systems.

Support for software evolution in-the-large includes techniques and tools that aid interchange, reconfiguration, extension, and scaling of software modules and/or systems. Evolution in the current economic context also requires reuse of third-party components. Traditional development approaches do not provide the necessary support. In particular, approaches such as structural programming or object-oriented analysis and design fail to properly decouple computation from communication within a system, thus supporting only limited reconfigurability and reuse. Conventional evolution techniques have also typically been programming language specific (e.g., inheritance) and applicable on the small scale (e.g., separation of concerns or isolation of change). This is only partially adequate in the case of development with preexisting, large, heterogeneous components that originate from multiple sources.

The research hypothesis of this dissertation is that *software architecture* is the appropriate abstraction for supporting evolution in-the-large. Software architecture research is directed at reducing the costs of developing applications and increasing the potential for commonality among different members of a closely related product family [70], [83]. Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements (*components* and *connectors*) and their overall interconnection structure (*configurations*). Additionally, architectures separate computation in a system (performed by components) from interaction among the system's computational units (facilitated by connectors). This enables developers to abstract away the unnecessary details and focus on the "big picture:" system structure, high level communication protocols, assignment of software components and connectors to hardware components, development process, and so forth. The basic promise of software architecture research is that better software systems can result from modeling their important aspects throughout, and especially early in the development. Choosing which aspects of a system to model and how to evaluate them are two decisions that frame software architecture research [49].

Evolution is an aspect of a software system that should be planned for throughout development [28]. Doing so at the level of software architecture, an early model of a solution to the customer's requirements, thus becomes critical. However, current research has predominantly focused on other areas, e.g., formal specification and analysis of architectures. While some researchers are investigating the issues in architecture-based, run-time system reconfigurability [43], [61], no current approaches address the problem of specification-time evolution, no specific techniques are provided to support the reuse of existing components and connectors during evolution, and no effective methods or tools exist to enable the mapping of (evolved) architectures to their implementations in a property-preserving manner.

The goal of our work is to develop just such a principled, architecture-based method for supporting reuse-driven development of flexible, extensible, and evolvable software. One observation that guides this research is that architectures can evolve at the level of any of their top-level constructs: components, connectors, or configurations. This dissertation presents a comprehensive methodology for specification-time, architecture-based evolution that addresses all three levels:

- evolution of components via heterogeneous subtyping,
- evolution of connectors via context-reflective interfaces and heterogeneous information filtering mechanisms, and
- evolution of architectural configurations via heterogeneous, flexible connectors and minimal component interdependencies.

Furthermore, the methodology supports transferring of architectural decisions to implementations and reuse of existing components and connectors.

The different facets of our methodology have been incorporated into a specific *architectural style,* C2, in order to bound the scope of the dissertation to a well defined investigation, demonstration, and evaluation platform. Architectural styles are key design idioms that reflect and leverage underlying characteristics of an application domain and recurring patterns of application design within the domain. We have used the C2 style to model graphical user interface (GUI) intensive applications and software development tool suites. Some of C2's properties, e.g., implicit invocation, have been adopted from previous research and their benefits are well understood. Others, e.g., substrate independence, are unique to C2, but show potential for general applicability. This dissertation exploits both categories of properties specifically for the purpose of supporting evolution. We also introduce heterogeneous subtyping, a novel technique that, though applied in the context of C2, is style-independent. Using all these techniques in concert is unique to this dissertation.

The specific contributions of the dissertation are as follows.

Component Evolution. We define a *taxonomy* that divides the space of potentially complex subtyping relationships into a small set of well defined, manageable subspaces. This taxonomy is used as the basis of a flexible *type theory* for software architectures. By adopting a richer notion of typing, this theory is applicable to a broad class of design, evolution, and reuse circumstances across application domains, architectural styles, and architecture description languages (ADLs). Additionally, the type theory enables architecture-level *analysis* by establishing type conformance between interoperating components. The rules of type conformance are defined in a manner that is better suited than other existing techniques to support the "large scale development with off-the-shelf reuse" philosophy on which architecture research is largely based. We have also designed a *simple ADL* that embodies the principles of the type theory.

Connector Evolution. Unlike the evolution of components, which is supported with a specific technique, the connectors employed in our approach are *inherently evolvable*. The interface exported by a connector is *context-reflective*, i.e., it evolves to support any components that interact through the connector. This adds a degree of freedom in composing components and enables architecture reconfiguration and extension. A connector also evolves by altering its communication *filtering policy* to support data broadcast, point-to-point exchange, or no interaction among components. Different filtering policies may impact an application's performance and may also aid in testing and debugging the application.

Configuration Evolution. Configuration evolution is supported in this dissertation by employing *flexible connectors*, discussed above, *minimizing component interdependencies*, and providing *heterogeneous connector implementations*. Flexible connectors are key to supporting architectural reconfiguration: *addition*, *removal*, *replacement*, and *reconnection* of architectural elements. We minimize component interdependencies by combining two well-understood techniques, *implicit invocation* and *asynchronous communication*, with a novel one, *substrate independence*. Also unique to this dissertation is the ability to evolve only the interaction aspects of an architecture, keeping the functionality unchanged, by interchanging implementations of a connector that support different *types of interaction* and *degrees of concurrency*.

Implementation Support. We support the implementation of architectures with a simple, extensible *implementation infrastructure*, a set of techniques to enable *reuse* of off-the-shelf (OTS) components and connectors, and an *environment* for architecture-based development. In tandem, they preserve desired architectural properties in the implementation, reduce development time, and improve the reliability of the resulting software. The environment contains several unique features. It supports *specification*, *analysis*, and *evolution* of architectures described in our ADL. It also provides tool support for *partial generation* of an application from its architecture, which, in turn, facilitates reuse of OTS components. The environment itself is *component-based*; its architecture was designed to be easily *evolvable* to support multiple ADLs, types of analysis, architectural styles, and implementation platforms. Our approach is fully *reflexive*: the environment can be used to describe, analyze, evolve, and (partially) implement itself, using the very ADL it supports. Also implemented into the environment is the notion of *architect's discretion* to override the results of architectural analysis in the case of errors (s)he believes not to be critical.

The claims of this dissertation have been explored and its contributions demonstrated in a series of example applications.

The remainder of the dissertation is organized as follows. Chapter 2 presents the C2 architectural style, our research, demonstration, and validation platform. It purpose is also to introduce certain concepts and issues relevant in the discussion of related work in Chapter 3. The two subsequent chapters present our methodology: Chapter 4 presents techniques for supporting the evolution of components, connectors, and architectural configurations, while Chapter 5 discusses our support for mapping architecture-level decisions, including evolution, to their implementation(s). Chapter 6 demonstrates the application of our methodology to several extensive examples and shows how the work described in the dissertation validates the above contributions. Chapter 7 discusses future work and is followed by conclusions in Chapter 8.

CHAPTER 2: The C2 Architectural Style

In order to explore and validate our ideas and apply them in practice, we chose a specific architectural style, C2, as our research and demonstration platform [89]. Our intent in this case is reflective of UC Irvine software architecture group's research philosophy: *generalize from specific experience* [88]. The C2 architectural style was originally designed to support the particular needs of applications that have a significant GUI aspect. However, the style clearly has the potential for supporting other types of applications and we have since used it to achieve software tool interoperability. C2 draws its key ideas from many sources, including other architectural styles, such as client-server, pipe-and-filter, and blackboard, as well as from experience with the Chiron-1 user interface development system [90].

A key motivating factor behind development of the C2 style is the emerging need, in the user interface community, for a more component-based development economy [95]. User interface software frequently accounts for a very large fraction of application software, yet reuse in the UI domain is typically limited to toolkit (widget) code. The C2 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 concurrently in a distributed, heterogeneous environment without shared address spaces, architectures may be changed at runtime, multiple users may be interacting with the system, multiple toolkits may be employed, multiple dialogs may be active and described in different formalisms, and multiple media types may be involved.

The C2 architectural style can be informally summarized as a network of concurrent components hooked together by message routing devices. Central to the 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 here to support an intuitive understanding of the 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.

All components have their own thread(s) of control and there is no assumption of a shared address space. At minimum, this means that components may not assume that they can directly invoke other components' operations or have direct access to other components' data. It is important to recognize that a conceptual architecture is distinct from its implementation, as there are many ways of realizing a given conceptual architecture. This topic will be further discussed below.

A simple example, adopted from [89], serves to illustrate several of these points. In Figure 2-1, we diagram a system in which a program alternately pushes and pops items from a stack; the system also displays the stack graphically, using the visual metaphor of a stack of plates in a cafeteria. The human user can "directly" manipulate the stack by dragging elements to and from it, using a mouse. As the user drags elements around on the display, a scraping sound is played.

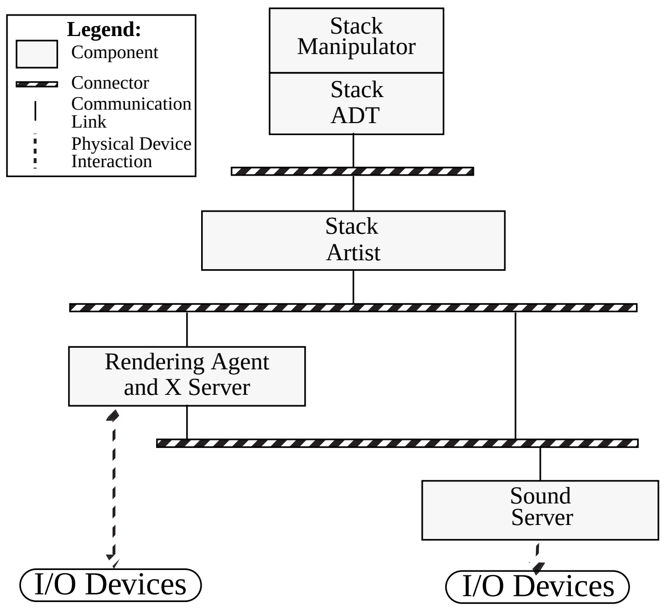


Figure 2-1. An audio-visual stack manipulation system.

Whenever the stack is pushed, a sound appropriate for a spring being compressed is played; whenever the stack is popped, the sound of a plate breaking is played.

Visual depiction of the stack is performed by the “artist” that receives notification of operations on the stack and creates an internal abstract graphics model of the depiction. The rendering agent monitors manipulation of this model and ultimately creates the pictures on the workstation screen. To produce the audio effects, the sound server at the bottom of the architecture monitors the notifications sent from the artist and the graphics server; depending on the events detected, the various sounds are played. Performance is such that playing of the sound is very closely associated with mouse movement; there is no perceptible lag. The artist and rendering agent are completely unaware of the activities of the sound server; similarly, the stack manipulator is completely unaware that its stack object is being visualized.

Key elements of the C2 style are *components* and *connectors*. A *configuration* of a system of components and connectors is an *architecture*. There is also a set of principles governing how the components and connectors may be legally composed, discussed below. Components and connectors both have a defined top and bottom. The top of a component may be connected to the bottom of a single connector. The bottom of a component may be connected to the top of a single connector. There is no bound on the number of components or connectors that may be attached to a single connector. Components can only communicate via connectors; direct communication is disallowed. When two connectors are attached to each other, it must be from the bottom of one to the top of the other. Both components and connectors have semantically rich interfaces. Components communicate by passing *messages*; *notifications* travel down an architecture and *requests* up. Connectors are responsible for the routing and potential multi-cast of the messages.

The remainder of this chapter further elaborates on the properties of C2.

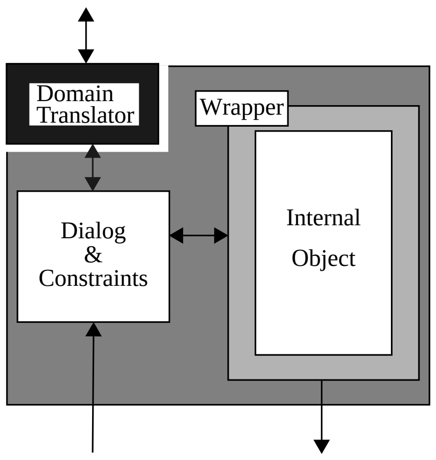


Figure 2-2. The Internal Architecture of a C2 Component.

2.1 C2 Components

Components may have state, their own thread(s) of control, and must have a top and bottom domain. The top domain specifies the set of notifications to which the component responds, and the set of requests that the component emits up an architecture. The bottom domain specifies the set of notifications that the component emits down an architecture and the set of requests to which it responds. The elements of a bottom domain's sets are closely related, as will be discussed later. The two sets comprising the top domain do not necessarily have any relation.

For purposes of exposition below, a specific internal architecture of a component, targeted at the GUI domain, is assumed.[1] Components contain an object with a defined interface, a wrapper around the object, a dialog and constraint manager, and a domain translator, as shown in Figure 2-2.[2] The object can be arbitrarily complex. For example, one component's object might be a complete structured graphics model of the contents of a window. The object's wrapper provides the following service: whenever one of the access routines of the object's interface is invoked, the wrapper reifies that invocation and any return values as a notification in the component's bottom domain and sends the notification to the connector below the component.[3] Thus the types of notifications emitted from a component are determined by the interface to its internal object.

The access routines of the object may only be invoked by the dialog portion of a component. This code, which may have its own thread of control, may act upon the object for any reason, but the intended style includes three situations:

- in reaction to a notification that it receives from the connector above it. The dialog receives a notification in its top domain and determines what, if anything, to do as a result of receiving the notification.
- to execute a request received from the connector below it. The component receives a request in its bottom domain and determines what, if anything, to do with the request. For instance, it

1. Issues concerning composition of an architecture are independent of a component's internal structure, so this assumption is not at all restrictive.
2. With the exception of a specific example discussed in Chapter 6, the "dialog and constraints" portion of a component will be referred to simply as "dialog" in the remainder of this dissertation.
3. Components can alternatively be formulated such that the wrapper sends to the connector the state, or part of the state, of the internal object.

could choose to delay processing of the request, ignore it, perform it without any additional processing, or perhaps perform some other action.

- to maintain some constraint, as defined in the dialog. This case is best understood by considering its user interface purpose: constraint managers are commonly employed in GUI applications to resize fields, planarize graphs, or otherwise keep parts of objects in some defined juxtaposition. The constraint portion of a component can play this role either as part of the previous two cases, or the constraint manager may autonomously manipulate the component's object.

The dialog portion of a component may, in addition, choose to send a request to the connector above it. A domain translator subcomponent may also be present, to assist in mapping between the component's internal semantic domain and that of the connector above it.

2.2 Notifications and Requests

Components in an architecture communicate asynchronously via messages. Messages consist of a name and an associated set of typed parameters. There are two types of messages: notifications and requests. A notification is sent downward through a C2 architecture while a request is sent up. Notifications are announcements of state changes of the internal object of a component. As noted above, the types of notifications that a component can emit are fully determined by the interface to the component's internal object.

Requests, on the other hand, are directives from components below, generated by their dialogs, requesting that an action be performed by some set of components above. The requests that a component can receive are determined by the interface to the component's internal object, similar to the way that notifications are determined. The difference is that a notification is a statement of what interface routine was invoked and what its parameters and return values were, whereas a request is a statement of a desired invocation of one of the object's access functions.

2.2.1 Domain Translation

Since a component has no knowledge of the interfaces of components below it and does not directly issue requests to those components, a component is independent of its substrate layers. This substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. One issue that must be addressed, however, is the potential dependence of a given component on its "superstrate," i.e., the components above it. If each component is built so that its top domain closely corresponds to the bottom domains of those components with which it is specifically intended to interact in a given architecture, its reusability value is greatly diminished. For that reason, the C2 style introduces the notion of domain translation. Domain translation is a transformation of the requests issued by a component into the specific form understood by the recipient of the request, as well as the transformation of notifications received by a component into a form it understands. This transformation process is encapsulated in the domain translator part of a component, as shown in Figure 2-2.

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

- *Message name matching* — a mismatch may occur because a message name is different than expected. For example, a component may issue a "stack_pop" request to a component which has a “pop_stack” entry point. In this case, domain translation involves a simple name replacement.

- *Parameter matching* — a mismatch may occur in the number, ordering, type, and units of parameters. As an example of parameter matching difficulties, suppose component A issues a “make_alarm” request giving a time delay in seconds before component B issues an “alarm” notification. A parameter mismatch occurs if component B only understands compound time values of seconds and milliseconds, or only understands time values if they are given in milliseconds.

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

Simple domain translations, such as name replacement and parameter order swapping could be specified by the system architect using the facilities of the development environment. Simple translations will frequently be automatable, particularly in cases where there exists an approximate one-to-one correspondence between the messages received by a component and those it actually understands. More commonly, however, this task will at least partly be guided by the software architect. A human agent is needed to provide semantic interpretation for both the component’s top domain and the interface presented by the connector above it. More difficult domain translations such as the generation of missing parameters and unit conversions may require manual generation of domain translators using either a scripting language or a programming language.

Domain translation unavoidably adds overhead to the message passing process. This is likely to be less than the cost of passing the message itself, especially across thread or process boundaries, and is not a major source of inefficiency, however. Domain translation can be viewed as a tradeoff between slightly diminished message passing efficiency and the ability to reuse components “as-is.”

The need for domain translation can be considerably reduced by the adoption of standard interfaces for similar components. Exemplifying this approach are domain-specific software architectures (DSSAs) [91], where similar components are characterized by similar interfaces, certain component configurations are common, and usual patterns of component usage are known to both the architect and the design environment.

Note that many potential C2 components, such as commercial user interface toolkits, have interface conventions that do not match up with C2’s notifications and requests. Typically these systems will generate events of the form “this window has been selected” or “the user has typed the ‘x’ key” and send them *up* an architecture. These toolkit events will need to be converted by C2 bindings to the toolkits into C2 request messages. Conversely, notifications from a C2 architecture will have to be converted to the type of invocations that a toolkit expects. In order for these translations to occur and be meaningful, careful thought has to go into the design of the internal objects of the bindings to the toolkits such that they contain the required functionality and are reusable across architectures and applications. This is not an unreasonable task: we have already accomplished this for both Motif and OpenLook in Chiron-1 [90], as well as for Xlib [77] and Java’s AWT [14] in C2, as discussed in Chapter 5.

2.3 Connectors

Connectors bind components together into a C2 architecture. They may be connected to any number of components as well as other connectors. A connector's primary responsibility is the routing and broadcast of messages. A secondary responsibility of connectors is message filtering. Connectors may provide a number of filtering and broadcast policies for messages, such as the following:

- *no filtering*: Each message is sent to all connected components on the relevant side of the connector (bottom for notifications, top for requests).
- *notification filtering*: Each notification is sent to only those components that have registered for it.
- *message filtering*: Each message is sent only to those components that can understand and respond to it. This filtering mechanism enables "point-to-point" communication in a C2 architecture.
- *prioritized*: The connector defines a priority ranking over its connected components, based on a set of evaluation criteria specified by the software designer during the construction of the architecture. This connector then sends a notification to each component in order of priority until a termination condition has been met.
- *message sink*: The connector ignores each message sent to it. This is useful for isolating subsystems of an architecture as well as incrementally adding components to an existing architecture. A developer can connect a new component to the architecture and then "turn on" its connector, by changing its filtering policy, when the component is ready to be tested or used.

2.4 Architecture Composition and Properties

An architecture consists of a specific configuration of components and connectors. The meaningfulness of an architecture is a function of the connections made. This section formalizes several key relationships. In addition to aiding precise exposition, the formalizations are the basis for automated analyses of candidate architectures, e.g., by a development environment, such as the one discussed in Chapter 5.

Let *bottom_in* be the set of requests received at the bottom side of a component or connector. Let *bottom_out* be the set of notifications that a component or connector emits from its bottom side. Furthermore, let *top_in* be the set of notifications received on the top side of a component or connector, and let *top_out* be the set of requests sent from its top side.

Figure 2-3 represents the external view of a component C_i. C_i*.top_out* and C_i*.top_in* are defined by the component's dialog: they are the requests it will be submitting and notifications it will be handling. C_i*.bottom_out* are the notifications the component will be making, reflecting changes to its internal object. C_i*.bottom_in* are the requests the component accepts. Those requests can be defined as a function, *N_to_R*, of the notifications:

$$C_i.bottom\_in = N\_to\_R(C_i.bottom\_out)$$

This function is a bijection; it has an inverse function, *R_to_N*, that will uniquely map the requests to notifications.

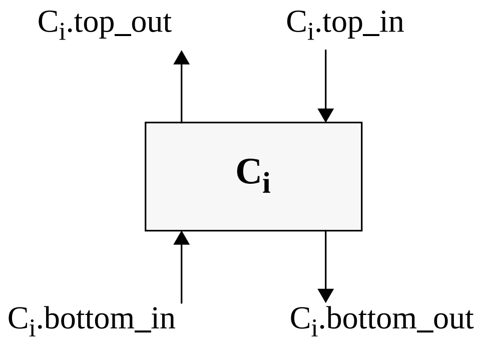


Figure 2-3. C2 Component Domains.

Pairwise relationships can be specified between the domains of a connector and any component attached to it. These relationships are expressed in terms of the potential for communication between them. Connector B_i and the j-th component attached to its top, C_{tj}, are considered *fully communicating* if every request the connector sends up to the component through the connector's j-th port (*top_out*$_j$) is "understood."

$$\mathit{Full\text{-}Comm}(B_i, C_{tj}) \equiv B_i.\mathit{top\_out}_j \subseteq C_{tj}.\mathit{bottom\_in}$$

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. Since components communicate via connectors, it is possible to specify pairwise relationships between the domains of any connector and each component attached to it. We can express the ability of a component to understand and respond to messages it receives and the utilization of a component's services in terms of that relationship.

B_i and C_{tj} are *partially communicating* if the component understands some, but not all of the requests the connector sends:

$$\begin{aligned} &\mathit{Partial\text{-}Comm}(B_i, C_{tj}) \equiv \\ &\quad (B_i.\mathit{top\_out}_j \cap C_{tj}.\mathit{bottom\_in} \neq \varnothing) \wedge \\ &\quad (B_i.\mathit{top\_out}_j \cap C_{tj}.\mathit{bottom\_in} \subset B_i.\mathit{top\_out}_j) \end{aligned}$$

A component and a connector are *not communicating* as follows:

$$\mathit{No\text{-}Comm}(B_i, C_{tj}) \equiv (B_i.\mathit{top\_out}_j \cap C_{tj}.\mathit{bottom\_in} = \varnothing)$$

The relationship between a connector B_i and a component C_{bk} below it can be defined in a similar manner, by substituting '*bottom_out*' for '*top_out*' and '*top_in*' for '*bottom_in*' in the above equations.

The degree of utilization of a component's services, i.e., the relationship between a component and a connector from the perspective of the messages the component *receives* from the connector can be defined through a simple substitution of terms in the three equations above. For instance, if B_i.*top_out* is a non-empty proper subset of C_{tj}.*bottom_in*, then C_{tj} is being *partially utilized*.

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

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

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

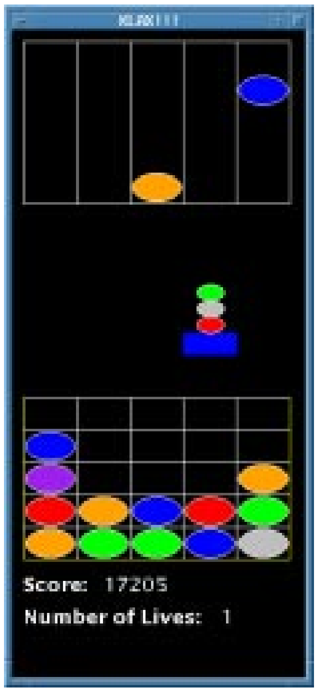
Finally, by utilizing the functions and relationships specified above, it is possible to express a number of other relationships in a given configuration (e.g., B_i*.bottom_out* can be expressed as a function of C_{tj}*.bottom_in*). All such relationships can be deduced from the complete formal definition of the C2 style, given in Appendix A. We use this definition as a basis for our ADL, which enables modeling, analysis, evolution, and implementation of C2-style architectures.

2.5 Example Architecture in the C2 Style

To further illustrate the concepts and properties behind the C2 style, we present an extended example architecture. We revisit this architecture throughout this dissertation to illustrate and clarify our discussion. The example architecture is a version of the video game KLAX.[4] A description of the game is given in Figure 2-4. This particular application was chosen as a useful test of the C2 style concepts in that the game is based on common computer science data structures and the game layout maps naturally to modular artists. Also, the game play imposes some real-time constraints on the application, bringing performance issues to the forefront.

The architecture of the application is given in Figure 2-5. The components that make up the KLAX game can be divided into three logical groups. At the top of the architecture are the components which encapsulate the game's state. These data structure components are placed at the top since game state is vital for the functioning of the other two groups of components. These ADT components receive no notifications, but respond to requests and emit notifications of internal state changes. ADT notifications are directed to the next level where they are received by both the game logic components and the artists components.

4. KLAX is trademarked 1991 by Atari Games.



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

Figure 2-4. A snapshot and description of our implementation of the KLAX^{TM} video game.

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 *RelativePositionLogic* determines if the palette is in a position to catch the tile. If so, a request is sent to the *PaletteADT* component 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 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 *TileArtist* provides a flexible presentation level for tiles. Artists maintain information about the placement of abstract tile objects. The *TileArtist* intercepts any notifications about tile objects and recasts them to notifications about more concrete drawable objects. For example, a “tile-created” notification might be translated into a “rectangle-created” notification. The *LayoutManager* component receives all notifications from the artists and offsets any coordinates to ensure that the game elements are drawn in the correct juxtaposition.

The *GraphicsBinding* component receives all notifications about the state of the artists’ graphical objects and translates them into calls to a window system. User events, such as a key press, are translated into requests to the artist components. A keystroke typically results in 10 to 30 message sends throughout the KLAX architecture; a tick of the clock typically causes 3 to 20 message sends.

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 of the original architecture, shown in Figure 2-6, involved replacing the original tile matching, tile placing, and tile artist components with components which instead matched,

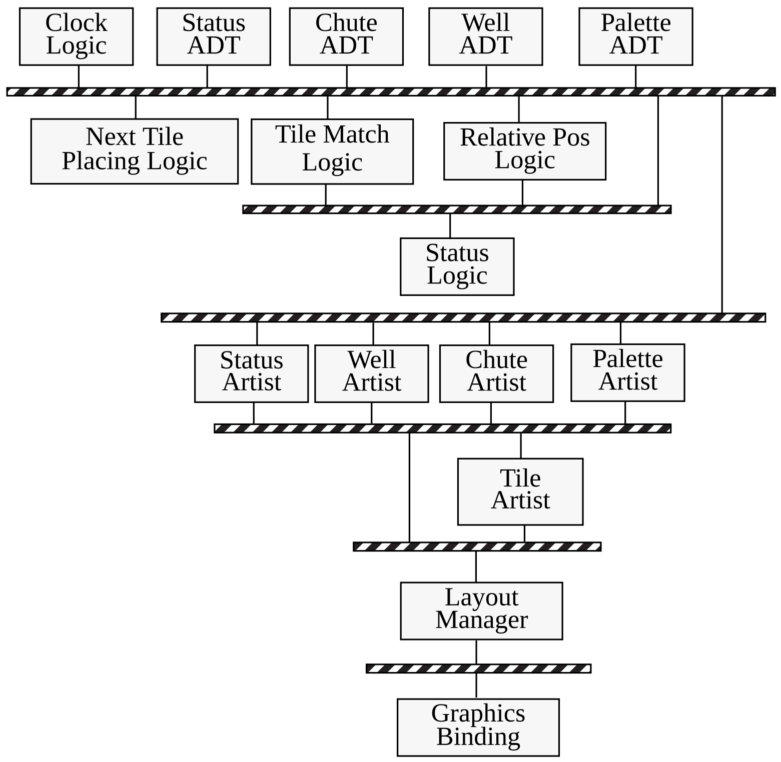


Figure 2-5. Conceptual C2 architecture for KLAX.
Note that the Logic and Artist layers do not communicate directly and are in fact "siblings." The Artist layer is shown below the Logic layer since the components in the Artist layer perform functions conceptually closer to the user.

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. No modifications to the rest of the architecture were needed to implement this variation.

2.6 Summary

The C2 architectural style is characterized by several principles, the collection of which distinguish it from other styles. Subsets of these principles, of course, characterize a variety of other systems.

- *Substrate independence* — a component is not aware of the components below it. In particular, the notification of a change in a component's internal object is entirely transparent to its dialog. Instead, the wrapper does this automatically when the dialog accesses the internal object. However, even the wrapper only generates a notification, not knowing whether any component will receive it and respond. Substrate independence fosters substitutability and reusability of components across architectures.
- *Message-based communication* — all communication between components is solely achieved by exchanging messages. This requirement is suggested by the asynchronous nature of applications that have a GUI aspect, where both users and the application perform actions concurrently and at arbitrary times and where various components in the architecture must be notified of those actions. Message-based communication is extensively used in distributed environments for which this architectural style is suited.

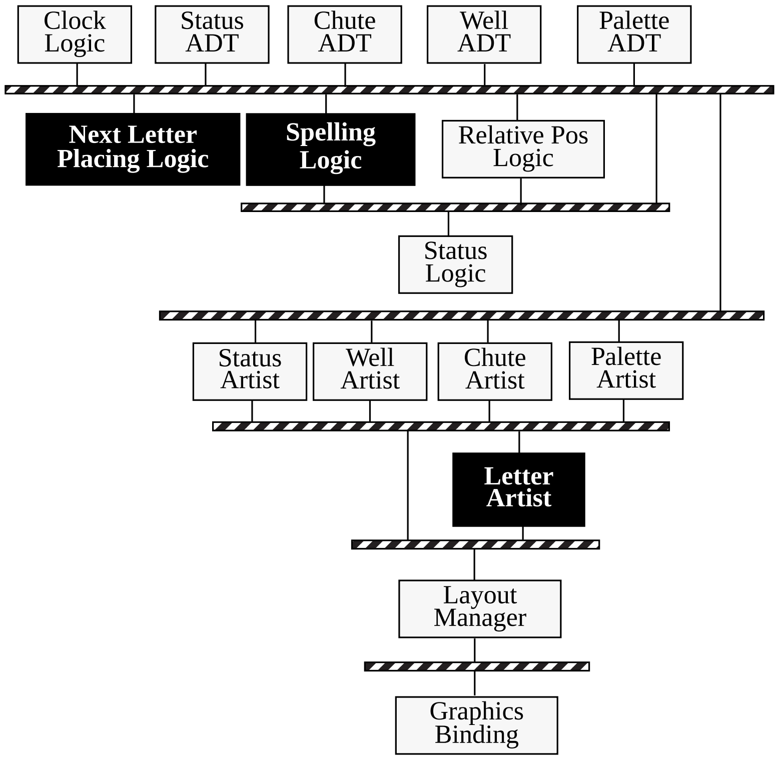


Figure 2-6. “Spelling” KLAX.
By replacing three components (highlighted in the diagram) from the original architecture, the game turns into one whose object is to spell words horizontally, vertically, or diagonally.

- *Multi-threaded* — this property is also suggested by the asynchronous nature of tasks in the GUI domain. It simplifies modeling and programming of multi-user and concurrent applications and enables exploitation of distributed platforms.
- *No assumption of shared address space* — any premise of a shared address space would be unreasonable in an architectural style that allows composition of heterogeneous components, developed in different languages, with their own threads of control, internal objects, and domains of discourse.
- *Implementation separate from architecture* — many potential performance issues can be remedied by separating the conceptual architecture from actual implementation techniques. For example, while the C2 style disallows any assumptions of shared threads of control and address spaces in a conceptual architecture, substantial performance gains may be made in a particular implementation of that architecture by placing multiple components in a single process and a single address space where appropriate. Furthermore, modelling the exchange of messages among components by procedure calls where appropriate could yield performance gains.

Although this dissertation exploits the above principles to support software evolution, their potential benefits are not restricted to evolution. Any (subset) of these principles can be employed to achieve other properties, such as distribution, safety (e.g., via redundant components), or scalability.

CHAPTER 3: Related Work

The research of this dissertation has been influenced by work in several areas: layered systems, implicit invocation systems, distributed systems, component interoperability models, object-oriented (OO) typing, behavioral specification of software, software environments, software reuse, and software architectures, architectural styles, and ADLs. Given the dissertation's focus on architectural models as critical, early points for software development and evolution, it is most naturally related and comparable to the research in software architectures and ADLs as the usual anchors of that research. However, much of the research that has influenced us is outside the software architecture arena. We thus first outline the relationship between our work and non-architectural approaches. We then provide a detailed overview of the work in software architectures.

3.1 Layered Systems

In contrast to existing systems, such as Field [73] and SoftBench [13], X Windows [77], Chiron-1 [90], Arch [71], and Slinky [92], which support only a fixed number of layers in an architecture, the C2 architectural style allows layering to vary naturally with the application domain. In this, the C2 style is similar to GenVoca [6], whose components may be composed in a number of layers that naturally reflects the characteristics of a particular domain. Unlike GenVoca, which uses explicit invocation, C2 provides a layering mechanism based on implicit invocation [26]. This allows the C2 style to provide greater flexibility in achieving substrate independence in an environment of dynamic, multi-lingual components: component recompilation and relinking can be avoided and on-the-fly component replacement enabled [61].

C2's explicit treatment of connectors directly distinguishes our work from more traditional layered systems, such as network systems (e.g., Avoca [6]) and operating systems. Connectors provide a level of indirection that reduces dependencies among computational elements (components). Coupled with implicit invocation and domain translation, this indirection gives developers more flexibility in building systems out of (existing) components whose interfaces do not match perfectly, and enhancing such systems incrementally as additional (needed) functionality becomes available. For example, a C2 connector can decide to route some of the requests that were initially handled (and possibly ignored) by component X to the new component Y, which can process them faster and/or provide higher-precision results. From the standpoint of components which are below them in a C2 architecture, X and Y comprise a single component, as illustrated in Figure 3-1.

3.2 Implicit Invocation

In C2, implicit invocation occurs when a component invokes its internal object in reaction to a notification. The invocation is implicit because a component issuing notifications does not know if those notifications will cause any reaction, nor does it explicitly name an entry point into a component below it. The benefits of implicit invocation are described in the context of mediators by Sullivan and Notkin [85], [86]. While many systems, such as Chiron-1 and VisualWorks [66][1], employ implicit invocation for its benefits in minimizing module interdependencies, the C2 style

1. VisualWorks is a Smalltalk GUI library based on the Model-View-Controller paradigm [37], where the model broadcasts change of state notifications to views and controllers.

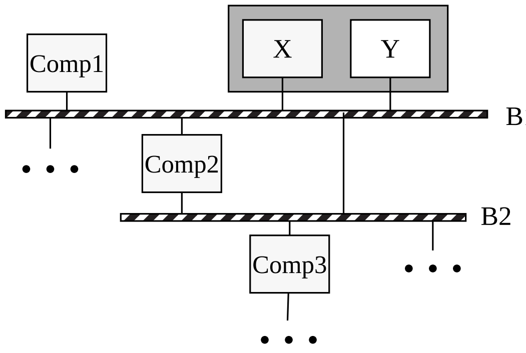


Figure 3-1. An example (partial) architecture built according to C2 style rules.
The architecture demonstrates C2's support for reconfigurability: component Y has been added to an existing architecture and connector B1 routes to it some of the requests that were previously delivered to component X. None of the components below B1 (e.g., Comp2 and Comp3) need to be updated in any manner, as they are effectively still communicating with a single component.

also provides a discipline for ordering components which use implicit invocation, yielding substrate independence.

Another example of implicit invocation is the Weaves system [29], in which concurrently executing tool fragments communicate by passing (pointers to) objects. This passing of objects causes Weaves to be used in a data flow manner. Weaves allows data moving between output and input portals of connected tool fragments. Unlike C2, which allows messages to flow in both directions along a communication link, data flow in weaves is unidirectional ("left to right"); flow in the other direction is achieved by explicitly creating communication cycles. Similarly to C2, communication in Weaves occurs via connectors (transport services). The granularity of Weave tool fragments is on the order of a single procedure. This, coupled with unidirectional data flow and communication cycles, can result in Weave architectures consisting of large numbers of tool fragments and transport services with a complicated interconnection structure.

3.3 Distribution

Existing systems tend to be rigid in terms of mapping their components to OS processes. At one extreme, X Windows applications contain exactly two processes, a client and a server. While there is greater process flexibility in VisualWorks and Weaves, both of these systems assume a shared address space. It is only with systems such as GenVoca, Field or SoftBench, and C2 that simultaneous satisfaction of arbitrary numbers of processes in a non-shared address space is achieved. Similarly to Field/SoftBench, C2 employs connectors to achieve distribution. Unlike them, however, C2 allows any number of connectors in a single application, removing the potential bottleneck and single point of failure of the single Field/Softbench software bus.

3.4 Component Interoperability

Existing component interoperability models, such as JavaBeans [31], OLE [11] and OpenDoc [62], provide standard formats for describing services offered by a component and runtime facilities to locate, load, and execute services of other components. Since these models are concerned with low-level implementation issues and provide little or no guidance in building a system out of components, their use is neither subsumed by or restricted by C2. In fact, these models may be used to realize an architecture in the C2 style, as demonstrated by Natarajan and

Rosenblum [58]. C2 provides its own interoperability infrastructure that exhibits the basic properties of the existing interoperability models. Furthermore, as this dissertation will show, C2's connectors can be used as a platform for incorporating multiple interoperability models in a single architecture.

3.5 Typing

This dissertation's method for supporting component evolution — heterogeneous subtyping — is influenced by object-oriented programming languages (OOPLs). A useful overview of OOPL subtyping relationships is given by Palsberg and Schwartzbach [65]. They describe a consensus in the OO typing community regarding the definition of a range of OO typing relationships. *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, while possibly extending it. *Behavioral conformance* allows any class to be a subtype of another if it preserves the interface and behavior of all methods available in the supertype. Finally, *strictly monotone subclassing* additionally 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.

OOPLs generally adopt only one of the subtyping/subclassing mechanisms along this spectrum (e.g., monotone subclassing). Unlike OOPLs, however, architectures may contain components implemented in heterogeneous programming languages and cannot rely on the subtyping support provided by a single language. Furthermore, software components may require subtyping methods not commonly found in OOPLs (e.g., preserving the behavior of a supertype, but altering its interface). Finally, while OOPLs generally adopt a single subtyping mechanism, our experience indicates that architectures often require multiple such mechanisms. For that reason, OOPL typing alone cannot fulfill the needs of software architectures.

This dissertation adapts and expands OOPL typing for use with software architectures (see Chapter 4) . We have developed a framework for understanding subtyping relationships as regions in a space of type systems; any of the relationships identified by Palsberg and Schwartzbach can be expressed as set operations on that space. This allows an architect to specify the most appropriate relationship between a supertype and its subtype.

3.6 Behavioral Specification and Conformance

As indicated above, our type theory enables the modeling and subtyping of component behavior. Behavioral specification and conformance have been explored by a number of researchers, including Abadi and Leino [1], America [4], Leavens and colleagues [16], [39], Fischer and colleagues [18], [79], Liskov and Wing [40], and Zaremski and Wing [99]. All these approaches are applied on low-level abstractions, either at the programming language or detailed design level. They all specify strict conformance criteria, based on one of several possible variations of the logical implication or equivalence relationships between component behaviors, as discussed by Zaremski and Wing [99].

The different approaches diverge in their granularity of types, certain, mostly minor, details of the required relationships, and the specific languages to which they are applied. For example, Fischer et al. require conformance of individual methods, while Dhara and Leavens [16] do so for entire components. Also, Liskov and Wing are influenced by, e.g., America's work, but expand his and similar definitions by adding a notion of history, essentially protocol conformance, to their definition.

This dissertation's approach to modeling behaviors and ensuring their conformance is similar to the above approaches: we model component behavior with invariants and operation preconditions and postconditions. However, we base our type theory on the ideas of heterogeneous subtyping foreshadowed in the previous subsection and apply it at the level of architectures. Our type theory differs from existing work in that we separate a component's behavior from its interface, allowing an operation to export multiple interfaces. We also separate functionality provided by a component from the functionality it expects in an architecture. We do so in a manner that is better suited to support development with third-party components than any of the existing approaches.

We also distinguish between *substituting* a supertype and its subtype and *evolving* the supertype. All existing approaches focus exclusively on the former. However, substitutability rules can be exceedingly rigid at the expense of the latter. We treat subtyping as a means of specifying new functionality in a systematic manner as well as ensuring the correctness of an architecture. Also, unlike the above approaches, we allow type-incorrect specifications (i.e., architectures) under certain conditions.

Finally, as already mentioned, Liskov and Wing define protocol conformance. Explicitly modeling protocols has been shown to have practical benefits [3], [60], [98], [99]. However, component invariants and method pre- and postconditions can be used to describe all state-based protocol constraints and transitions. Thus, our notion of behavioral conformance implies protocol conformance, and we do not address them separately.

3.7 Software Environments

A software development environment is a collection of capabilities integrated to support developers and managers in their work. To be effective, an environment must exhibit properties such as interoperability, heterogeneity, evolvability, and flexibility [35]. We have used these properties as general guidelines in constructing an environment for C2 style architecture-based software development.

Specifically, our work has roots in DSSA environments (e.g., ADAGE [5]). The DSSA approach is based on developing a generic *reference architecture* for all systems in a given domain of applicability [91]. The reference architecture is then instantiated for every individual system within the domain, as shown in Figure 3-2. By making reference architectures explicit, DSSAs employ a systematic approach to developing application families. The approach adopted by DSSAs to address the problem of modeling architectures and generating an implementation from an architecture is similar to ours: by restricting the software development space to a family of applications that share a specific architectural style, reference architecture, and possibly implementation infrastructure, DSSA environments have been very successful at transferring architectural decisions to running systems.

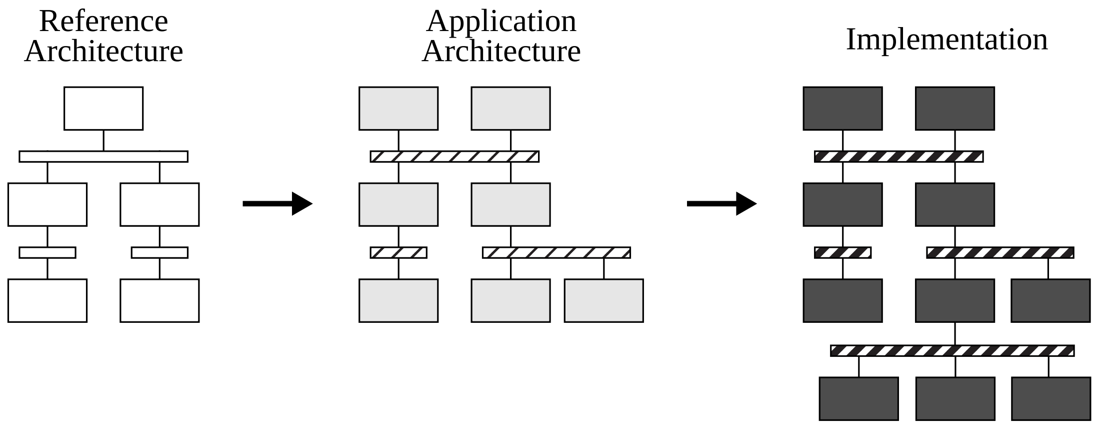


Figure 3-2. A simplified, high-level view of the DSSA lifecycle.
A generic reference architecture is instantiated to obtain an application-specific architecture, which is then used as the basis for implementation.

Unlike DSSA environments, we do not develop a reference architecture as a basis for building an application family. However, there is also nothing inherent in the C2 style or its supporting environment (see Chapter 5) that prohibits one from doing so. Quite the contrary, as an architectural style, C2 can be applied to multiple domains, each of which would require its own reference architecture. Each component in a C2 architecture is a conceptual placeholder that can be instantiated by different implemented modules. The ease with which we have been able to build application families (discussed in Chapter 6) without the aid of a reference architecture is indicative of C2's potential in this regard.

Our work has also drawn inspiration from the Inscape environment for software specification, implementation, and evolution [68]. Inscape addresses many of the same issues addressed by C2 (scale, evolution, complexity, programming-in-the-large, practicality of the approach), but does so at a level of abstraction below architecture. For example, Inscape requires the semantics of data objects to be modeled, while we treat them as unelaborated (*basic*) types. Both approaches model operations in terms of pre- and postconditions; Inscape also specifies *obligations*, predicates that must eventually be satisfied after an operation is invoked. Unlike our approach, which uses type-theoretic principles to evolve components, Inscape supports component evolution at a finer level of granularity and in a less systematic manner by adding, removing, and changing pre- and postcondition predicates, and also by changing the implementation itself. Finally, our approach is fully reflexive, i.e., our environment can be applied on itself; it is unclear whether and how Inscape could be used in its own development and evolution.

3.8 Software Reuse

Explicit focus on architectures, and architectural styles in particular have a great potential to facilitate both OTS component reuse and development of families of applications. At the same time, basing reuse on general-purpose software architectures is difficult. One of the challenges lies in effectively identifying, classifying, searhing, and retrieving existing components and architectures that are needed in a new situation [38]. This problem can be remedied by adopting higher-level abstractions that are applicable across applications. C2 is a style that attempts to exploit commonalities across systems, and reuse individual components as well as successful structural and interaction patterns.

The two goals of maximizing reuse and building system families do not always go hand in hand. For example, unlike their inherent support for application families, DSSAs have tended to support reuse only to a limited degree. GenVoca [6] is an illustrative example of such limited reuse. It has been particularly successful in producing a large library of reusable components. However, those components have been custom built for the GenVoca style. In order to reuse them, one must adhere to GenVoca's formalism and its hierarchical approach to component composition, which may result in a high degree of dependency between communicating components. On the other hand, C2's style rules are more flexible: C2 eliminates assumptions of shared address spaces and threads of control, allows both synchronous and asynchronous message-based communication, and separates the architecture from the implementation.[2]

Garlan, Allen, and Ockerbloom classify the causes of problems developers commonly experience when attempting OTS reuse and give four guidelines for alleviating them [24]:

- make architectural assumptions explicit in components,
- construct large components using orthogonal subcomponents,
- provide techniques for bridging mismatches, and
- develop sources of architectural design guidance.

C2 is well suited to address these problems. The first two guidelines deal with the internal architecture of OTS components, which is outside the reuser's control, and therefore also outside the scope of issues addressible by C2. The third guideline proposes techniques for building component adaptors, which is subsumed by C2 wrappers and domain translators. The final guideline emphasizes the need for design assistance, which has been a significant aspect of the work on C2 to date[74].

Finally, Shaw discusses nine "tricks" for reconciling component mismatch in an architecture [81]. Several of the tricks are related to reuse techniques employed in C2. For example, transformations, such as "Change [component] A's form to [component] B's form", "Provide B with import/export converters", and "Attach an adapter or wrapper to A," are subsumed by C2's wrappers and/or domain translators. The need for other transformations is eliminated altogether by C2 style rules. For example, "Make B multilingual" is unnecessary, as C2 assumes that components will be heterogeneous and multilingual.

3.9 Architecture Description Languages

ADLs have recently become an area of intense research in the software architecture community [20], [27], [96]. In order to support architecture-based development, formal modeling notations and analysis and development tools that operate on architectural specifications are needed. ADLs and their accompanying toolsets have been proposed as the answer to this need. A number of ADLs have been developed for modeling architectures both within a particular domain and as general-purpose architecture modeling languages: Aesop [23], C2SADEL[3] [53], Darwin [43], MetaH [94], Rapide [42], SADL [57], UniCon [82], Weaves [29], and Wright [3].

Recently, initial work has been done on an architecture interchange language, ACME [25], which is intended to support mapping of architectural specifications from one ADL to another,

2. Although the C2 style focuses mainly on asynchronous communication, it also allows synchronous interaction between components. This issue is discussed further in Chapter 4.
3. For simplicity and in order to more easily distinguish it from SADL, which resulted from an unrelated project, we refer to C2SADEL simply as "C2" in the remainder of this chapter.

and hence enable integration of support tools across ADLs. Although, strictly speaking, ACME is not an ADL, it contains a number of ADL-like features. It is useful to compare and differentiate it from other ADLs to highlight the difference between an ADL and an interchange language. It is therefore included in this discussion.

This section is drawn from our extensive classification and comparison of ADLs [46], [51]. It focuses on the issues pertinent to this dissertation. The remainder of the section is organized as follows. Section 3.9.1 introduces our definition and taxonomy of ADLs. Sections 3.9.2-3.9.5 describe the elements of the taxonomy and assess the above-mentioned ADLs based on these criteria. A brief summary of our findings is given in Section 3.9.6.

3.9.1 The Comparison Framework

Loosely defined, "an ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module" [93]. The building blocks of an architectural description are (1) *components*, (2) *connectors*, and (3) *architectural configurations*.[4] In order to infer *any* kind of information about an architecture, at a minimum, *interfaces* of constituent components must also be modeled. Without this information, an architectural description becomes but a collection of (interconnected) identifiers, similar to a "boxes and lines" diagram with no explicit underlying semantics.

Other aspects of both components and connectors on which we focus here are desirable, but not essential: their benefits have been acknowledged and possibly demonstrated by some ADL, but their absence does not mean that a given language is not an ADL. These features are *interfaces* (for connectors), and *types*, *semantics*, and *evolution* (for both). Desirable features of configurations are *understandability, heterogeneity*, *scalability*, *evolution*, and *system families*. Finally, even though the suitability of a given language for modeling software architectures is independent of whether and what kinds of tool support it provides, an accompanying toolset will render an ADL both more usable and useful. We focus on tools for *active specification*, *analysis*, and *implementation generation*.

3.9.2 Components

A component is a unit of computation or a data store [70]. Therefore, components are loci of computation and state [82]. A component in an architecture may be as small as a single procedure (e.g., MetaH *procedures* and Weaves *tool fragments*) or as large as an entire application (e.g., hierarchical components in C2 and Rapide or *macros* in MetaH). It may require its own data and/ or execution space, or it may share them with other components.

Each surveyed ADL models components in one form or another and under various names. ACME, Aesop, C2, Darwin, SADL, UniCon, and Wright share much of their vocabulary and refer to them simply as *components*; in Rapide they are *interfaces*;[5] in Weaves, *tool fragments*; and in MetaH, *processes*. In this section, we present the aspects of components relevant to this dissertation and assess existing ADLs with respect to them.

4. "Architectural configurations" will, at various times in this dissertation, be referred to simply as "configurations" or "topologies."
5. *Interface* is a language construct; the designers of Rapide commonly refer to Rapide interfaces as "components."

3.9.2.1 Interface

A component’s interface is a set of interaction points between it and the external world. As in OO classes or Ada package specifications, a component interface in an ADL specifies those services (messages, operations, and variables) the component provides. In order to be able to adequately reason about a component and the architecture that includes it, ADLs should also provide facilities for specifying component needs, i.e., services required of other components in the architecture. An interface thus defines computational commitments a component can make and constraints on its usage. Interfaces also enable a certain, though rather limited, degree of reasoning about component semantics.

All surveyed ADLs support specification of component interfaces. They differ in the terminology and the kinds of information they specify. For example, each interaction point in ACME, Aesop, SADL, and Wright is a *port*. On the other hand, in C2, the entire interface is provided through a single port; individual interface elements are *messages*. Weaves combines the two approaches by allowing multiple component *ports*, each of which can participate in the exchange of interface elements, or *objects*. In Darwin, an interaction point is a *service*, in Rapide a *constituent*, and in UniCon a *player*. MetaH distinguishes between *ports*, *events*, and *shared data*.

Most, but not all, ADLs distinguish between provided and required services. Some do so only by distinguishing incoming from outgoing ports. Others also specify the types of data that are provided or expected. Finally, C2, Rapide, and Wright are notable in that they also require explicit specification of required services’ semantics.

The ports in ACME, Aesop, SADL, and Wright are named and typed. SADL distinguishes between input and output ports (*iport* and *oport*), while Aesop allows definition of architectural styles that do so (e.g., *inputs* and *outputs* for pipe-and-filter components). Wright goes a step further by specifying the expected behavior of the component at that point of interaction. The particular semantics of a port (whether they provide or require data) are specified in CSP [32] as interaction protocols.

Component interface specifications in Darwin specify services *provided* and *required* by a component, as well as types of those services. Each service type is further elaborated with an interaction mechanism that implements the service. For example, *trace* services are implemented with *events*, *outputs* are accomplished via *ports*, and *commands* accept *entry* calls.

MetaH specifies input and output ports on components (processes). Ports are strongly typed and connections among them type checked. They are the means for periodic communication: each port has an associated buffer variable and port-to-port communication results in assignment. Aperiodic communication is modeled by output events. Finally, sharable monitors or packages are the means of indicating shared data among components.

Rapide subdivides component interfaces into constituents: *provides*, *requires*, *action*, and *service*. *Provides* and *requires* refer to functions. Connections between them specify synchronous communication. *In* and *out actions* denote the events a component can observe and generate, respectively. Connections between *actions* define asynchronous communication. A service is an aggregation facility for a number of actions and functions. It is a mechanism for abstracting and reusing component interface elements.

Weaves distinguishes between *read* and *write* ports. In order to maximize the flexibility of interconnection, Weaves ports are type-indifferent and “blind” (connection-indifferent). They perform wrapping and unwrapping of data objects by means of *envelopes*, which hide the types of the underlying data objects. Each port also supplies a *Wait* method, which implements a port-specific waiting policy in case of transmission problems.

UniCon specifies interfaces as sets of players. Players are visible semantic units through which a component interacts by requesting or providing services and receiving external state and events. Each player consists of a name, a type, and optional attributes such as signature, functional specification, or constraints. UniCon supports a predefined set of player types: *RoutineDef*, *RoutineCall*, *GlobalDataDef*, *GlobalDataUse*, *ReadFile*, *WriteFile*, *ReadNext*, *WriteNext*, *StreamIn*, *StreamOut*, *RPCDef*, *RPCCall*, and *RTLoad*. *PLBundle* denotes a set of players.

A C2 component interface, on the other hand, consists of single top and bottom ports. Both incoming (*req*uired) and outgoing (*prov*ided) message traffic is routed through each port. An important distinction among C2 messages is between *requests* and *notifications*. Due to C2’s principle of substrate independence, a component has no knowledge of components below it in an architecture: any messages sent down an architecture must be notifications of that component’s internal state; requests may only be sent up.

3.9.2.2 Types

Software reuse is one of the primary goals of architecture-based development [10], [24], [48]. Since architectural decomposition is performed at a level of abstraction above source code, ADLs can support reuse by modeling abstract components as types. Component types can then be instantiated multiple times in an architectural specification and each instance may correspond to a different implementation of the component. Another benefit of explicitly modeling component types is enabling evolution, as discussed below.

All of the surveyed ADLs distinguish component types from instances. Rapide does so with the help of a separate types language [41]. Weaves distinguishes between *sockets* and tool fragments that populate them. With the exception of MetaH and UniCon, all ADLs provide extensible component type systems. MetaH and UniCon support only a predefined, built-in set of types. MetaH component types are *process*, *macro*, *mode*, *system*, and *application*.[6] Component types supported by UniCon are *Module*, *Computation*, *SharedData*, *SeqFile*, *Filter*, *Process*, *SchedProcess*, and *General*.

3.9.2.3 Semantics

Component semantics are modeled to enable evolution, analysis, enforcement of constraints, and consistent mappings of architectures from one level of abstraction to another. However, several languages do not model component semantics beyond interfaces. SADL and Wright focus on other aspects of architectural description (connectors and refinement). Wright’s connectors require specification of interaction protocols for each component interaction point; while it does not focus on it, Wright also allows specification of component functionality in CSP.

6. As MetaH is used to specify both the software and the hardware architecture of an application, *system* is a hardware construct, while *application* pertains to both.

Underlying semantic models and their expressive power vary across those ADLs that do support specification of component behavior. ACME and UniCon allow semantic information to be specified in components' property lists. ACME places no restrictions on the specification language; however, from its point of view, properties are uninterpreted, so that, strictly speaking, component semantics are outside the scope of the language. Although UniCon's main focus is on non-functional properties of components, it allows specification of event traces in property lists to describe component behavior.

Aesop does not provide any language mechanisms for specifying component semantics. Instead, it allows the use of style-specific languages for modeling semantics for each architectural style defined in Aesop.

MetaH allows specification of component implementation semantics with path declarations. A path declaration consists of an optional identifier, followed by the names of (more primitive) components in that path. MetaH also uses an accompanying language, ControlH, for modeling algorithms in the guidance, navigation, and control (GN&C) domain [9].

In Rapide, each component specification has an associated *behavior,* which is defined via state transition rules that generate partially ordered sets of events (posets). Rapide uses event patterns to recognize posets for triggering rules and evaluating constraints. During poset recognition, free variables in a pattern are bound to specific matching values in a component's poset. Event patterns are used both as triggers and outputs of component state transition rules. Weaves employ a similar, though more primitive semantic model. It specifies a partial ordering between a tool fragment's input and output objects.

Darwin uses the π-calculus [54] as its underlying semantic model. A system in the π-calculus is a collection of independent processes that communicate via named channels. π-calculus is used to model basic component interaction and composition properties, so that each syntactic Darwin construct concerned with requiring, providing, and binding services is modeled in it. It is important to note that using π-calculus in this manner only supports modeling the semantics of composite Darwin components (see [46]), while primitive components are treated as black boxes.

Finally, C2 specifies a component's semantics in first-order logic. Constraints on the operation of the component as a whole are expressed in its invariant. The semantics of both provided and required operations are modeled via preconditions and postconditions. The behavior of a C2 component is modeled independently of its interface, so that it is possible to replace the operation with which an interface element is associated and to associate the same operation with multiple interface elements.

3.9.2.4 Evolution

As design elements, components evolve. ADLs must support a disciplined evolution process by supporting techniques such as subtyping of components and refinement of their features. Only a subset of existing ADLs provide support for evolution. Even within those ADLs, evolution support is limited and often relies on the chosen implementation (programming) language. The remainder of the ADLs view and model components as inherently static.

MetaH and UniCon define component types by enumeration, allowing no subtyping, and hence no evolution support. Weaves focuses on evolving architectures, rather than individual tool fragments. ACME has recently introduced types, and supports structural subtyping via its *extends* feature.

Aesop supports behavior-preserving subtyping to create substyles of a given architectural style. Aesop mandates that a subclass must provide strict subtyping behavior for operations that succeed, but may also introduce additional sources of failure with respect to its superclass. It is unclear whether this functionality has been implemented in Aesop's supporting toolset.

Rapide allows its interface types to inherit from other types strictly by using OO methods, resulting in structural subtyping. SADL, on the other hand, supports specification of properties that must be satisfied by all elements of a given subtype. Both Rapide and SADL also provide features for refinement of components across levels of abstraction. This mechanism may be used to evolve components by explicating any deferred design decisions, which is somewhat similar to extending inherited behavior in OO languages. Note that, in a general case, subtyping is a restricted form of refinement. This is, however, not true in the case of Rapide and SADL, both of which place additional constraints on refinement maps in order to prove or demonstrate certain properties of architectures [46].

C2 stands out in its support for component evolution. It attempts to avoid dependence on subtyping mechanisms provided by any underlying programming language. Our method is based on the realization that architectural design is a complex activity in which architectures may incorporate components implemented in heterogeneous programming languages; therefore, an ADL cannot rely on a single subtyping method provided by any one language. Using programming language terminology, C2 models conceptual component placeholders as formal parameters, while the implemented components that instantiate them are actual parameters. Multiple subtyping and type-checking relationships among components are allowed: name, interface, behavior, and implementation subtyping, as well as their combinations.

A more complete summary of this section is given in Table 3-1. The table has been adapted from [46].

3.9.3 Connectors

Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. As in the case of components, surveyed ADLs model connectors in various forms and under various names. For example, languages such as ACME, Aesop, C2, SADL, UniCon, and Wright model connectors explicitly and refer to them as *connectors*. Weaves also models connectors explicitly, but refers to them as *transport services*. In Rapide and MetaH they are *connections*, modeled in-line, and cannot be named, subtyped, or reused (i.e., connectors are not first-class entities). Connectors in Darwin are *bindings* and are also specified in-line, i.e., in the context of a configuration only. In this section, we present the aspects of connectors that contribute strongly to their role as facilitators of architecture-centered evolution and compare existing ADLs with respect to them.

3.9.3.1 Interface

In order to enable proper connectivity of components and their communication in an architecture, a connector should export as its interface those services it expects. Therefore, a connector's interface is a set of interaction points between it and the components attached to it. It enables reasoning about the well-formedness of an architectural configuration.

Table 3-1: ADL Support for Modeling Components

Features / *ADL*	Interface	Types	Semantics	Evolution
ACME	interaction points are *ports*	extensible type system; parameterization enabled with templates	no support; can use other ADLs' semantic models in property lists	structural subtyping via the *extends* feature
Aesop	interaction points are *input* and *output ports*	extensible type system	(optional) style-specific languages for specifying semantics	behavior-preserving subtyping
C2	interface exported through top and bottom *port*s; interface elements are *prov*ided and *req*uired	extensible type system	causal relationships between input and output messages	heterogeneous subtyping
Darwin	interaction points are *services* (*provide*d and *required*)	extensible type system; supports parameterization	π-calculus	none
MetaH	interaction points are *ports*	Predefined, enumerated set of types	ControlH for modeling algorithms in the GN&C domain; implementation semantics via paths	none
Rapide	interaction points are *constituents* (*provides*, *requires*, *action*, and *service*)	extensible type system; contains a types sublanguage; supports parameterization	partially ordered event sets (posets)	inheritance (structural subtyping)
SADL	interaction points are input and output *ports* (*iports* and *oports*)	extensible type system; allows parameterization of component signatures	none	subtyping by constraining supertypes; refinement via pattern maps
UniCon	interaction points are *players*	predefined, enumerated set of types	event traces in property lists	none
Weaves	interaction points are *read* and *write ports*; interface elements are *objects*	extensible type system; types are component *sockets*	partial ordering over input and output objects	none
Wright	interaction points are *ports*; port interaction semantics specified in CSP	extensible type system; parameterizable number of ports and computation	not the focus; allowed in CSP	none

Only those ADLs that support modeling of connectors explicitly, independently of configurations in which they are used, support specification of connector interfaces. Weaves are somewhat of an exception: although transport services are modeled explicitly, their interfaces are not specified directly, but rather as *pads* of the encompassing sockets. A transport service itself is indifferent to the types of data it handles; its main task is to buffer and synchronize the communication among tool fragments.

ACME, Aesop, UniCon, and Wright refer to connector interaction points as *roles*. Explicit connection of component ports and connector roles is required in an architectural configuration. Roles are named and typed, and are in many ways similar to component ports (players in UniCon), discussed in Section 3.9.2.1. Aesop allows definition of architectural styles that distinguish between input and output roles (e.g., *sources* and *sinks* for pipe-and-filter connectors). Semantics of each role's interaction protocol in Wright are specified in CSP, similarly to port protocols. This allows for analysis of compatibility between connected component ports and connector roles. In UniCon, each role may include optional attributes, such as the type of players that can serve in the role and minimum and maximum number of connections. UniCon also supports only a predefined set of role types: *Source, Sink*, *Reader, Readee*, *Writer, Writee, Definer, Caller, User*, *Participant*, and *Load*.

A SADL connector only exports the type of data it supports in its interface. Other information about the connector, such as the number of components it connects, is implicit in the connector type (see Section 3.9.3.2) and/or specified as part of the architectural configuration (Section 3.9.4). Finally, although Darwin and Rapide define their connectors in-line, both languages allow abstracting away complex connection behaviors into “connector components,” which are then accompanied by a set of simple connections (bindings in Darwin).

In C2, connector interfaces, like component interfaces, are modeled with *ports*. Each port can export multiple messages; the sets of messages at two different ports belonging to the same connector need not be disjoint. C2 connectors are unique in that their interfaces are *context-reflective*. In other words, a connector’s interface is determined by and adapts to the (potentially dynamic) interfaces of components that communicate through it.

3.9.3.2 Types

Architecture-level communication may need to be expressed with complex protocols. To abstract away these protocols and make them reusable, ADLs should model connectors as types. This is typically done in two ways: as extensible type systems which are defined in terms of communication protocols and are independent of implementation, or as built-in, enumerated types which are based on their implementation mechanisms.

Only those ADLs that model connectors as first-class entities distinguish connector types from instances. This excludes languages like Darwin, MetaH, and Rapide. ACME, Aesop, C2, SADL, and Wright base connector types on protocols. ACME also provides a parameterization facility through connector templates and SADL via parameterized connector signatures and constraints that define connector semantics. Similarly to their components, Wright allows connectors to be parameterized by the number of roles and by the glue specification, while Weaves distinguishes between sockets from transport services that instantiate them. UniCon, on the other hand, only allow connectors of prespecified enumerated types. UniCon currently supports *Pipe*, *FileIO*, *ProcedureCall*, *DataAccess*, *PLBundler*, *RemoteProcCall*, and *RTScheduler* connector types.

MetaH does not support connector types, but it does define three broad categories of connections. In *port* connections, an *out* port of one component may be connected to an *in* port of another. *Event* connections allow outgoing events to be connected to incoming events (event-to-event), as well as to their recipient components (event-to-process and event-to-mode). Finally, *equivalence* connections specify objects that are shared among components.

3.9.3.3 Semantics

To perform useful analyses of component interactions, consistent refinement mappings across levels of architectural abstraction, and enforcement of interconnection and communication constraints, architectural descriptions provide connector protocol and transaction semantics. It is interesting to note that languages that do not model connectors as first-class objects, e.g., Rapide, may still model connector semantics.

ADLs often use a single mechanism for specifying the semantics of both components and connectors. For example, ACME allows connector semantics to be specified in its property lists using any specification language, but considers them uninterpreted; Rapide uses posets to describe communication patterns among its components; Wright models connector *glue* and event trace specifications with CSP; and UniCon allows specification of semantic information for connectors in property lists (e.g., a real-time scheduling algorithm or path traces through real-time code). Additionally, connector semantics in UniCon are implicit in their connector types. For example, declaring a component to be a *pipe* implies certain functional properties.

Exceptions to this rule are Aesop, C2, SADL, and Weaves. Aesop uses a different semantic model for its connectors than it does for components. Namely, Aesop does not use style-specific formal languages, but supports specification of operational connector semantics and (optionally) also employs Wright to specify connector semantics. C2 provides an insight into how a connector will behave by specifying its message filtering policies. SADL does not focus on modeling component semantics, but supports specification of connector semantics via axioms in SADL's constraint language. Finally, Weaves employs a set of naming conventions that imply a transport service's semantics. For example, a single-writer, single-reader queue transport service is named *Queue_1_1*.

3.9.3.4 Evolution

Component interactions are governed by complex and ever changing and expanding protocols. Maximizing connector reuse is achieved by modifying or refining existing connectors whenever possible. As with components, ADLs can support connector evolution with specific techniques such as subtyping and refinement.

Even fewer ADLs support evolution of connectors than do evolution of components. ADLs that do not model connectors as first-class objects (Darwin, MetaH, and Rapide) also provide no facilities for their evolution. Others either currently only focus on architecture-level evolution (e.g., Weaves), or provide a predefined set of connector types with no language features for evolution support (e.g., UniCon). Wright does not facilitate connector subtyping, but supports type conformance, where a role and its attached port may have behaviorally related, but not necessarily identical, protocols. ACME, Aesop, and SADL provide more extensive support for connector evolution, similar to their support for component evolution discussed in Section 3.9.2.4. ACME supports structural connector subtyping via its *extends* feature. Aesop supports behavior preserving subtyping, while SADL supports subtyping of connectors and their refinements across styles and levels of abstraction.

C2 does not provide techniques for connector evolution that are similar to its component subtyping. Instead, C2 connectors are inherently evolvable due to their context-reflective interfaces. Furthermore, the degree of information a C2 connector filters out can evolve, e.g., to account for newly added or removed components.

A more complete summary of this section is given in Table 3-2. The table has been adapted from [46].

3.9.4 Configurations

Architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structure. This information is needed to determine whether: appropriate components are connected, their interfaces match, connectors enable proper communication, and their combined semantics result in desired behavior. In concert with models of components and connectors, descriptions of configurations enable assessment of concurrent and distributed aspects of an architecture, e.g., potential for deadlocks and starvation, performance, reliability, security, and so on. Descriptions of configurations also enable analyses of architectures for adherence to design heuristics, e.g., to determine whether an architecture is

Table 3-2: ADL Support for Modeling Connectors

Features / ADL	Interface	Types	Semantics	Evolution
ACME	interaction points are *roles*	extensible type system, based on protocols; parameterization via templates	no support; can use other ADLs' semantic models in property lists	structural subtyping via the *extends* feature
Aesop	interaction points are *roles*	extensible type system, based on protocols	(optional) semantics specified using Wright	behavior-preserving subtyping
C2	interface with each component via a separate *port*; interface elements are *prov*ided and *req*uired	extensible type system, based on protocols	partial semantics specified via message filters	context-reflective interfaces; evolvable filtering mechanisms
Darwin	none; allows "connection components"	none	none	none
MetaH	none	none; supports three general classes of connections: port, event, and equivalence	none	none
Rapide	none; allows "connection components"	none	posets; conditional connections	none
SADL	connector signature specifies the supported data types	extensible type system; parameterized signatures and constraints	axioms in the constraint language	subtyping; connector refinement via pattern maps
UniCon	interaction points are *roles*	predefined, enumerated set of types	implicit in connector's type; semantic information can be given in property lists	none
Weaves	interaction points are the encapsulating socket *pads*	extensible type system; types are connector *sockets*	via naming conventions	none
Wright	interaction points are *roles*; role interaction semantics specified in CSP	extensible type system, based on protocols; parameterizable number of roles and glue	connector *glue* semantics in CSP	supports type conformance for behaviorally related protocols

"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). Finally, architectural description is necessary to establish adherence to architectural style constraints, such as C2's rule that there are no direct communication links between components.

Architectures are likely to describe large, long-lived software systems that may evolve over time. The changes to an architecture may be planned or unplanned; they may also occur before or during system execution. ADLs must support such changes through features for modeling specification-time evolution and execution-time, or run-time, evolution [61]. Another key role for modeling architectural configurations is to facilitate communication for the many stakeholders in the development of a system. The goal of configurations is to abstract away the details of individual components and connectors. They depict the system at a high level that can potentially be understood by people with various levels of technical expertise and familiarity with the problem at hand. This section investigates whether and to what degree various ADLs fulfill these roles.

3.9.4.1 Understandable Specifications

One of the major roles of software architectures is that they facilitate understanding of (families of) systems at a high level of abstraction. To truly enable easy communication about a system among developers and other stakeholders, ADLs must model structural (topological) information with simple and understandable syntax. The structure of a system should ideally be clear from a configuration specification alone, i.e., without having to study component and connector specifications.

Configuration descriptions in Darwin, MetaH, and Rapide tend to be encumbered with details of connectors, which are modeled in-line. On the other hand, ACME, Aesop, C2, SADL, UniCon, Weaves, and Wright provide separate, explicit abstractions for components and connectors and thus arguably have the best potential to facilitate understandability of architectural structure. Clearly, whether this potential is realized or not will also depend on the particular ADL's syntax. For example, UniCon allows the connections between players and roles to appear in any order, possibly distributed among individual component and connector specifications; establishing the topology of such an architecture may (unnecessarily) require studying a significant portion of the architectural description. Although somewhat subjective, a distinction can also be made between those notations that employ formalisms that are accessible to a narrower crossection of practitioners (e.g., Wright's CSP, or Rapide's posets) than others (e.g., C2's first-order logic).

3.9.4.2 Heterogeneity

A goal of software architectures is to facilitate development of large-scale systems, preferably with pre-existing components and connectors of varying granularity, specified by different designers, potentially in different formal modeling languages, implemented by different developers, possibly in different programming languages, with varying operating system requirements, and supporting different communication protocols. It is therefore important that ADLs be *open*, i.e., to specification and development with heterogeneous components and connectors.

Although no ADL provides explicit support for multiple specification languages, ACME, Aesop, C2, Darwin, and UniCon do allow it in principle. ACME's and UniCon's property lists are open, and will accept any modeling notation. To actually achieve architectural interchange in ACME, however, explicit mappings are required from architectural models described in one notation to another. Aesop allows style-specific modeling languages for component semantics, in addition to using operational semantics and/or Wright for modeling connectors. The possibility of using multiple notations for components within a *single* style is not precluded either. Darwin uses π-calculus to model external (visible) component characteristics and the semantics of composite components. At the same time, it leaves open the choice of specification languages for the semantics of primitive components. Although C2 currently bases its modeling, analysis, implementation, and evolution support on operation pre- and postconditions specified in first-order logic, nothing in the style or in the ADL precludes a choice of other formalisms.

Of the ADLs that support implementation of architectures, several are tightly tied to a particular programming language. For example, Aesop and Darwin only support development with components implemented in C++, while MetaH is exclusively tied to Ada and UniCon to C. On the other hand, Weaves supports interconnection of tool fragments implemented in C, C++, Objective C, and Fortran; and Rapide supports construction of executable systems specified in VHDL, C, C++, Ada, and Rapide itself. C2 currently supports development in C++, Ada, and Java. C2 is also unique in that it supports heterogeneous connector implementations, possibly by utilizing OTS middleware.

MetaH and Weaves place some additional restrictions on components. MetaH requires that each component contain a loop with a call to the predeclared procedure KERNEL.AWAIT_DISPATCH to periodically dispatch a process. Any existing components have to be modified to include this construct before they can be used in a MetaH architecture. Similarly, all Weaves tool fragments must implement a set of control methods: *Start*, *Suspend*, *Resume*, *Sleep*, and *Abort*. ADLs may also preclude reuse of many existing components and connectors by allowing only certain types of each. For example, UniCon can use existing filters and sequential files, but not spreadsheets, constraint solvers, or relational databases.

Finally, most surveyed ADLs support modeling of both fine and coarse-grain components. At one extreme are components that describe a single operation, such as *computations* in UniCon or *procedures* in MetaH, while the other can be achieved by *hierarchical composition*, where an entire architecture becomes a single component in another (larger) architecture.

3.9.4.3 Scalability

Architectures are intended to support large-scale systems. For that reason, ADLs must support specification and development of large systems that may further grow in size. For the purpose of this discussion, we can generalize the issues inherent in scaling software, so that an architectural configuration, such as that depicted in Figure 3-3, can be scaled up in two ways: by adding components and connectors along its boundaries (Figure 3-3a), and by adding elements to architecture's interior (Figure 3-3b). To support the former, ADLs can employ compositionality features, by treating the original architecture as a single, composite component, which is then attached to new components and connectors. Objectively evaluating an ADLs ability to support the latter, i.e., adding internal elements, is more difficult, but certain heuristics can be of help.

It is generally easier to expand architectures described in *explicit configuration ADLs*, such as ACME, Aesop, C2, SADL, UniCon, Weaves, and Wright, which model connectors as first-class entities, than those described in *in-line configuration ADLs* like Darwin, MetaH, and Rapide, which model them only as part of a configuration. Connectors in the latter are described solely in terms of the components they connect; adding new components or connectors may require direct modification of existing connector instances.

ADLs, such as C2, UniCon, Weaves, and Wright, that allow a variable number of components to be attached to a single connector are better suited to scaling up than those, such as ACME or Aesop, (or an earlier version of Wright described in [2]), which specify the exact number of components a connector can handle.[7] For example, ACME and Aesop could not handle the extension to the architecture shown in Figure 3-3b without redefining *Conn1* and *Conn2*, while C2, UniCon, Weaves, and Wright can.

3.9.4.4 Evolution

Support for software evolution is a key aspect of architecture-based development. Evolution at the architectural level encompasses component and connector addition, removal, replacement, and reconnection. An architecture evolves to reflect and enable evolution of a set of software systems based on that architecture. ADLs need to augment evolution support at the level of

7. The number of components attached to a Wright or UniCon connector must be specified at connector instantiation-time.

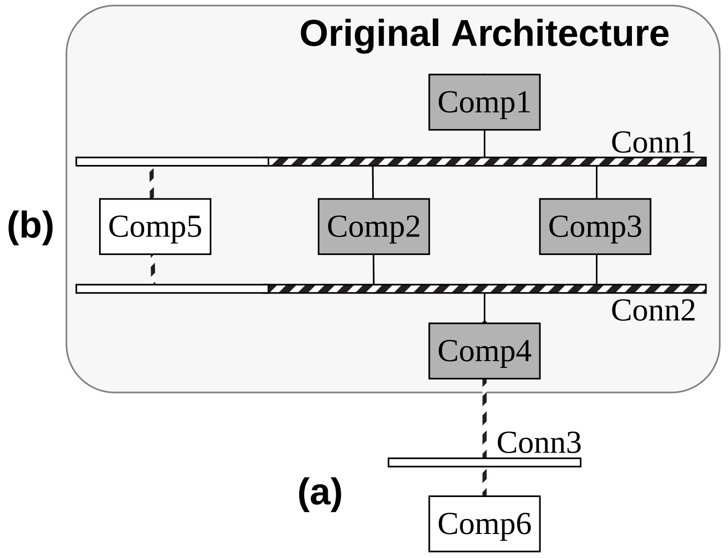


Figure 3-3. Scaling up an architecture.
(a) The architecture is expanded along its boundaries.
(b) New components/connectors are added to the architecture's interior.

components (Section 3.9.2.4) and connectors (Section 3.9.3.4) with features for incremental development.

Incrementality of an architectural configuration can be viewed from two different perspectives. One is the ability to accommodate addition of new components in the manner depicted in Figure 3-3.[8] The issues inherent in doing so were discussed above. The arguments that were applied to scalability also largely apply to incrementality: in general, explicit configuration ADLs can support incremental development more easily and effectively than in-line configuration ADLs; ADLs that allow variable numbers of components to communicate through a connector are well suited for incremental development, particularly when faced with unplanned architectural changes.

Additionally, an ADL's support for incrementality is inversely related to the degree of dependency among components in an architecture. In-line configuration ADLs (Darwin, MetaH, and Rapide), which model *connections*, rather than first-class connectors, embed a large degree of interdependency into their components. To add a component to a Darwin architecture, for example, existing components may have to be modified to establish the proper connections. Along similar lines, ADLs whose connectors are instantiated with the *exact* number, type, and interaction profile of components whose communication they can support (e.g., Wright or UniCon) ultimately hamper incrementality. If one is using such an ADL to evolve a configuration by adding a component, for example, a connector instance may need to be replaced in order to handle the new component. Similarly, if a component makes explicit assumptions about the components with which it will interact and requires those assumptions to be fully satisfied (e.g., Wright's port protocols), evolution of a configuration that includes such a component may not be possible.

8. A similar argument can be made for component removal. For the purposes of this discussion, we view reconnection as a combination of component addition with removal.

As this dissertation will show, it is exactly these types of considerations that have guided our choice of C2 properties: C2 components and connectors make minimal assumptions about other components and connectors in an architecture. This, in turn, allows the use of heterogeneous connectors, thus enabling a C2 architecture to better adapt to changing requirements.

Another view of incrementality is an ADL's tolerance and/or support for incomplete architectural descriptions. Incomplete architectures are common during design, as some decisions are deferred and others have not yet become relevant. Most existing ADLs and their supporting toolsets have been built around the notion that precisely these kinds of situations must be prevented. For example, Darwin, MetaH, Rapide, and UniCon compilers, constraint checkers, and runtime systems have been constructed to raise exceptions if such situation arise.[9] In this case, an ADL such as Wright, which focuses its analyses on information local to a single connector, is better suited to accommodate expansion of the architecture than, e.g., SADL, which is very rigorous in its refinement of *entire* architectures.

C2 makes no distinction between "incomplete" and "complete" architectural descriptions, as reflected in its connectors with context-reflective interfaces, support for partial communication and component service utilization, and focus on dynamic change [61]. Indeed, we consider an architecture in any state to be complete (i.e., it is analyzable, evolvable, and possibly describes meaningful functionality that can be transferred to an implementation). Conversely, any architecture is inherently incomplete, in that it is expected to continuously evolve to fulfill new requirements.

3.9.4.5 System Families

A potential benefit of explicit architectural models is that they can help identify or highlight ways to minimize the costs of developing new products, by sharing and reusing software structure and/or components. New software systems rarely provide entirely unprecedented functionality, but are rather "variations on a theme." They either belong to a family of systems, where much of the structure and functionality may be shared among different members, or to a style, where certain composition and communication characteristics recur within a given application domain or set of domains. In order for software architects to be able to adequately take advantage of existing (partial) solutions to their problems and of the potential cost savings incurred by system families, ADLs should provide adequate support for families.

System families can exist and grow at two different levels:

1. the architecture remains constant among the different members of the family, while their implementations vary;
2. the architecture also varies across the members of the family.

In the first case, the members of the family are functionally more closely related. The ability of an ADL to support this first type of system family largely depends on its separation of architecture from implementation, i.e., allowing multiple implementations of a given architectural model. Not all ADLs allow this.

Certain ADLs, e.g., C2, Wright, and Rapide do not assume or prescribe a particular relationship between an architectural description and its implementation(s). We refer to these

9. UniCon does allow differing levels of completeness of an architecture, depending upon the task. For example, schedulability analysis does not require source code, but only a specification of the appropriate real-time information.

languages as *implementation independent*. On the other hand, several ADLs, e.g., Weaves, UniCon, and MetaH, require a much higher degree of fidelity of an architecture to its implementation. Components modeled in these languages are directly related to their implementations. These are *implementation constraining* languages. Darwin has elements of both implementation constraining and independent languages: it ties each primitive component to its implementation, but also enables modeling of composite components.

Implementation independence alone does not completely support system families. It does not address the creation of families by modifying architectures across family members. Component and connector inheritance, subtyping, or other evolution mechanisms are also inadequate: for example, interchanging two components that are related by subtyping does not guarantee that the resulting two architectures belong to the same logical family. Therefore, additional techniques are needed.

One such technique may be to exploit hierarchical composition and apply subtyping or inheritance to composite components. Another possible solution, adopted by ACME, takes advantage of an ADL's support for non-functional attributes: in addition to components and connectors used in a configuration, ACME also specifies the application family to which the architecture belongs. ACME's supports for architectural families goes beyond this simple notational addition: application families are first-class constructs that can also evolve using the *extends* feature. The component and connector types declared in a specific family provide a design vocabulary for all systems that are declared as members of that family.

A more complete summary of this section is given in Table 3-3. The table has been adapted from [46].

3.9.5 Tool Support for ADLs

A major impetus behind developing formal languages for architectural description is that their formality renders them suitable to manipulation by software tools to support architectural design, evolution, analysis, and executable system generation. The need for tool support in architectures is well recognized. However, there is a definite gap between what is identified as desirable by the research community and the state of the practice. While every surveyed ADL provides some tool support, with the exception of C2 and Rapide, they tend to focus on a single area of interest, such as analysis (e.g., Wright), refinement (e.g., SADL), or dynamism (e.g., Weaves). Furthermore, within these areas, ADLs tend to direct their attention to a particular technique (e.g., Wright's analysis for deadlocks), leaving other facets unexplored. This is the very reason ACME has been proposed as an architecture interchange language: to enable interaction and cooperation among different ADLs' toolsets and thus fill in these gaps. This section surveys the tools provided by the different languages, attempting to highlight the biggest shortcomings.

3.9.5.1 Active Specification

Active specification support can significantly reduce the cognitive load on software architects. Only a handful of existing ADLs provide tools that actively support specification of architectures. In general, such tools can be proactive or reactive. Proactive specification tools act in a proscriptive manner, similar to syntax-directed editors for programming languages: they limit the available design decisions based on the current state of architectural design. For example, such

Table 3-3: ADL Support for Modeling Architectural Configurations

Features / ADL	Understandability	Heterogeneity	Scalability	Evolution	System Families
ACME	explicit, concise textual specification	open property lists; required explicit mappings across ADLs	aided by explicit configurations; hampered by fixed number of roles	aided by explicit configurations	first-class architectural families
Aesop	explicit, concise graphical specification; parallel type hierarchy for visualization	allows multiple languages for modeling semantics; supports development in C	aided by explicit configurations; hampered by fixed number of roles	no support for partial architectures; aided by explicit configurations	implementation independent
C2	explicit, concise textual and graphical specification	enabled by internal component architecture; supports development in C++, Java, and Ada	aided by explicit configurations and variable number of connector ports; used in the construction of its own tool suite	allows partial architectures; aided by explicit configurations; minimal component interdependencies; heterogeneous connectors	implementation independent
Darwin	implicit textual specification with many connector details; provides graphical notation	allows multiple languages for modeling semantics of primitive components; supports development in C++	hampered by in-line configurations	no support for partial architectures; hampered by in-line configurations	implementation independent
MetaH	implicit textual specification with many connector details; provides graphical notation	supports development in Ada; requires all components to contain a process dispatch loop	hampered by in-line configurations	no support for partial architectures; hampered by in-line configurations	implementation constraining
Rapide	implicit textual specification with many connector details; provides graphical notation	supports development in VHDL, C/C++, Ada, and Rapide	hampered by in-line configurations; used in large-scale projects	no support for partial architectures; hampered by in-line configurations;	implementation independent
SADL	explicit, concise textual specification	supports both fine- and coarse-grain elements	aided by explicit configurations; used in large-scale project	no support for partial architectures; aided by explicit configurations;	implementation independent
UniCon	explicit textual and graphical specification; configuration description may be distributed	supports only predefined component and connector types; supports component wrappers	aided by explicit configurations and variable number of connector roles	some support for partial architectures; aided by explicit configurations;	implementation constraining
Weaves	explicit, concise graphical specification	development in C, C++, Objective C, and Fortran; requires all tool fragments to provide a set of methods	aided by explicit configurations, sockets, and variable number of socket pads; used in large-scale project	allows partial architectures; aided by explicit configurations	implementation constraining; support for application families via socket populated *frameworks*
Wright	explicit, concise textual specification	supports both fine- and coarse-grain elements	aided by explicit configurations and variable number of roles; used in large-scale project	suited for partial specification; aided by explicit configurations	implementation independent

tools may prevent selection of components whose interfaces do not match those currently in the architecture or disallow invocation of analysis tools on incomplete architectures.

UniCon’s graphical editor operates in this manner. It invokes UniCon’s language processing facilities to *prevent* errors during design, rather than correct them after the fact. Furthermore, the editor limits the kinds of players and roles that can be assigned to different types of components and connectors, respectively. Similarly, C2’s DRADEL development environment proactively guides the “architecting” process by disallowing certain operations (e.g., architectural type checking) before others are completed (e.g., topological constraint checking).

Aesop provides a syntax-directed editor for specifying computational behavior of *filters*. Although no other types of components are currently supported, integration with external editors for such components is allowed in principle. Aesop also provides a type hierarchy for visualizations of its architectural elements, where each component and connector class has an associated visualization class. For example, the *pipe* subclass of *connector* refers to the *arrow* visualization class, which is a subclass of the more general *connector_line* class. These classes can refer to external editors, so that, e.g., a visualization class in the pipe-and-filter style invokes an editor on filter code.

Darwin's *Software Architect's Assistant* [59] is another example of a proactive specification tool. The *Assistant* automatically adds services of appropriate types to components that are bound together. It also maintains the consistency of data types of connected ports: changing one port's type is automatically propagated to all ports which are bound to it. Finally, the choice of component properties during specification is constrained via dialogs.

Reactive specification tools detect *existing* errors. They may either only inform the architect of the error (*non-intrusive*) or also force him to correct it before moving on (*intrusive*). In the former case, once an inconsistency is detected, the tool informs the architect, but allows him to remedy the problem as he sees fit or ignore it altogether. The type checker in C2's DRADEL environment provides non-intrusive active specification support: the architect can proceed to the implementation generation phase even in the presence of type mismatches. In the latter case, the architect is forced to remedy the current problem before moving on. Certain features of MetaH's graphical editor can be characterized as intrusive: the MetaH editor gives the architect full freedom to manipulate the architecture until the *Apply* button is depressed, after which any errors must be rectified before the architect may continue with the design.

3.9.5.2 Analysis

Architectural descriptions are often intended to model large, distributed, concurrent systems. The ability to evaluate the properties of such systems upstream, at the architectural level, can substantially lessen the cost of any errors. Given that many unnecessary details are abstracted away in architectures, this task may also be easier than at source code level. Analysis of architectures has thus been the primary focus of ADL toolset developers.

The types of analyses for which an ADL is well suited depend on its underlying semantic model and, to a lesser extent, its specification features. For example, Wright, which is based on CSP, analyzes individual connectors and components attached to them for deadlocks. Aesop currently provides facilities for checking for type consistency, cycles, resource conflicts, and scheduling feasibility in its architectures. It also uses Wright's tools to analyze connectors. Darwin enables analysis of architectures by instantiating parameters and dynamic components to enact "what if" scenarios. Similarly, Rapide's, C2's, and Weaves' event monitoring and filtering tools facilitate analysis of architectures through simulation. Another analysis technique commonly employed in Weaves, and enabled by its data-flow nature, is the insertion of tool fragments whose only task is to analyze the data they receive from adjacent fragments in a weave. MetaH and UniCon both currently support schedulability analysis by specifying non-functional properties, such as criticality and priority. Given two architectures, SADL can establish their relative correctness with respect to a refinement map. Finally, C2's DRADEL environment ensures that the topological constraints imposed by the style are enforced; it also uses the specification of provided and required component services to establish type conformance among components in an architecture or a type hierarchy.

Language parsers and compilers are other kinds of analysis tools. Parsers analyze architectures for syntactic correctness, while compilers establish semantic correctness. All of the surveyed languages have parsers. Darwin, MetaH, Rapide, and UniCon also have "compilers," enabling them to generate executable systems from architectural descriptions, provided the implementations of individual components already exist. Aesop must provide style-specific compilers that can process the style-specific formal notations used in modeling components. For example, Aesop currently provides a compiler for the pipe-and-filter style and its substyles, such as pipeline. C2 provides a tool that generates executable implementation skeletons from an architectural model; the skeletons are completed either by developing new functionality or by reusing OTS components.

Another aspect of analysis is enforcement of constraints. Parsers and compilers enforce constraints implicit in type information, non-functional attributes, component and connector interfaces, and semantic models. Rapide also supports explicit specification of other types of constraints, and provides means for their checking and enforcement. Its *Constraint Checker* analyzes the conformance of a Rapide simulation to the formal constraints defined in the architecture. C2's constraint checker currently focuses only on the rules of the style; an initial integration with the architecture constraint checking tool, Armani [55], allows specification and enforcement of arbitrary constraints.

3.9.5.3 Implementation Generation

An elegant architectural model that exhibits desirable properties and is amenable to sophisticated analysis is of little value unless it can be refined into an implementation in a consistent and systematic manner. A large number of ADLs, but not all, provide such support.

Darwin and UniCon require preexisting component implementations in C++ and C, respectively, in order to generate applications. Rapide can construct executable systems in the same manner in C, C++, Ada, and VHDL, or it can use its executable sublanguage. Weaves generates implementations by providing dynamic linking support for tool fragments already implemented in C, C++, Objective C, and Fortran.

There are several problems with this approach. Primarily, there is an assumption that the relationship between elements of an architectural description and those of the resulting executable system will be 1-to-1. This is not always necessary, and may also be unreasonable, as architectures are intended to describe systems at a higher level of abstraction than source code modules. Secondly, there is no guarantee that the specified source modules will correctly implement the desired behavior; even if the specified modules currently implement the needed behavior correctly, this approach provides no means of ensuring that any future changes to those modules are traced back to the architecture and vice versa. Finally, this approach assumes certain homogeneity among OTS components that are composed into a system.

Aesop provides a C++ class hierarchy for its concepts and operations, such as components, connectors, ports, roles, connecting a port to a role, and so on. This hierarchy forms a basis from which an implementation of an architecture may be produced; the hierarchy is in essence a domain-specific language for implementing Aesop architectures. Aesop currently only generates C code for architectures in the pipe-and-filter style.

A similar approach is used in C2: we developed a framework of abstract classes for C2 constructs (discussed in Chapter 5). The framework implements interconnection and message passing protocols and enables generation of top-level (“main”) application routines, a concept required by the implementation languages. The framework has been implemented in C++ and Java; its subset is also available in Ada. Using this framework, the DRADEL environment automatically generates an application’s skeleton from an architecture. As discussed in Section 3.7, this approach is influenced by the DSSA work: we restrict the software development space to a specific architectural style and implementation infrastructure in order to transfer architectural decisions to running systems.

Several ADLs—SADL, ACME, and Wright—are currently used strictly as modeling notations and provide no implementation generation support. It is interesting to note that, while SADL focuses on refining architectures across levels of abstraction, it does not take the final step from architectural descriptions to source code.

A more complete summary of this section is given in Table 3-4. The table has been adapted from [46].

Table 3-4: ADL Tool Support

Features / *ADL*	Active Specification	Analysis	Implementation Generation
ACME	none	parser	none
Aesop	syntax-directed editor for components; visualization classes invoke specialized external editors	parser; style-specific compiler; type checker; cycle checker; checker for resource conflicts and scheduling feasibility	*build* tool constructs system glue code in C for pipe-and-filter style
C2	proactive “architecting” process in DRADEL; reactive, non-intrusive type checker	parser; style rule checker; type checker	class framework enables generation of C/C++, Ada, and Java code; DRADEL generates application skeletons
Darwin	automated addition of ports to communicating components; propagation of changes across bound ports; dialogs to specify component properties;	parser; compiler; “what if” scenarios by instantiating parameters and dynamic components	compiler generates C++ code
MetaH	graphical editor requires error correction once architecture changes are *applied*; constrains the choice of component properties via menus	parser; compiler; schedulability, reliability, and security analysis	DSSA approach; compiler generates Ada code
Rapide	none	parser; compiler; analysis via event filtering and animation; constraint checker to ensure valid mappings	executable system construction in C/C++, Ada, VHDL, and Rapide
SADL	none	parser; analysis of relative correctness of architectures with respect to a refinement map	none
UniCon	graphical editor prevents errors during design by invoking language checker	parser; compiler; schedulability analysis	compiler generates C code
Weaves	none	parser; real-time execution animation; low overhead observers; analysis/debugging components in a weave	dynamic linking of components in C, C++, Objective C, and Fortran; no code generation
Wright	none	parser; model checker for type conformance of ports to roles; analysis of individual connectors for deadlock	none

3.9.6 Discussion

Software architecture research provides a wide spectrum of specification and tool support. The support for certain areas, e.g., formalism in architectures and analyses it enables, has been pervasive. This has, inevitably, resulted in neglect of other important areas. The support for architecture-based evolution, in particular, is sparse. ADLs that do address evolution typically treat an architectural description as a conventional program, relying on a chosen implementation language to enforce a single form of subtyping/subclassing. Our experience indicates that architectures require more flexible and heterogeneous evolution methods. This is supported by the prevalent view of architectures as independent of the programming language(s) in which they may be implemented.

The separation of an architecture from its implementation also calls into question some aspects of existing approaches to implementing an architecture. The implementation generation approaches have largely embraced one of two positions:

- provide programming language-level constructs in the ADL and use a *compiler* to generate the executable system from the “architecture”; or
- assume that every architectural component corresponds to an existing implemented module and employ a *linker* to join those modules into a system.

Both classes of approaches fail to fully separate the architecture from its implementation(s). They can also hamper heterogeneity: the former is likely to exclude third-party components, while the latter has typically assumed that all components are implemented in a single programming language (see Table 3-4). Explicitly linking the modules prior to the system’s execution also results in a static implementation architecture. To remedy these shortcomings, we have developed an approach that combines reuse, arbitrary distribution, explicit connectors in the implementation, and heterogeneous implementation substrates.

CHAPTER 4: Evolution of Components, Connectors, and Configurations

This chapter presents our solution to the problem of specification-time, architecture-based evolution of software systems. The solution is comprehensive in that it provides a methodology for evolving all three top-level architectural constructs: components, connectors, and architectural configurations. Each is discussed in detail in Sections 4.1, 4.2, and 4.3, respectively. An ADL that embodies the evolution concepts discussed in this chapter is introduced in Section 4.4.

4.1 Component Evolution

This dissertation's support for component evolution stems from the recognition that architecture-level components share certain traits with OO classes, and the resulting expectation that, in particular, techniques for evolving the latter can be adapted to support the evolution of the former. An architectural component is similar to an OO class: the services a component provides are equivalent to a class specification; the services it requires roughly correspond to OO messages.[1] The specific aspect of OOPLs we adapt for supporting component evolution is the OO type theory. Garlan has argued 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 [21]. We take this notion further: 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 type hierarchies is a domain-independent approach that structures relationships between software components and enables us to verify those relationships via type checking. Furthermore, an existing software module can evolve in a controlled manner via subtyping.

Our approach to component evolution is indeed based on a type theory. We treat each component specification in an architecture as a type and support its evolution via subtyping. However, while programming languages (and several existing ADLs [23], [25], [42]) support a single subtyping mechanism, architectures may require multiple subtyping mechanisms, many of which are not commonly supported in programming languages. Therefore, existing programming language type theories are inadequate for use in software architectures.

Beyond evolution, types are also useful in establishing certain correctness criteria about a program or an architecture. As discussed in Chapter 3, several existing ADLs support type checking (e.g., Aesop [23], Darwin [43], Rapide [42], and UniCon [82]). However, as with most all of the existing programming languages, these ADLs essentially establish simple syntactic matches among interacting components. Our approach also establishes semantic conformance of components.

1. Note that some component properties have no OO equivalent. For example, 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. This is generally not allowed in an OOPL. Another difference is the granularity, since a component may encapsulate a number of objects (class instances). These differences are not critical in the case of component evolution [47]. Architectures address several other issues not found in OOPLs, including connectors as first-class entities and style-imposed topological constraints on the composition of component instances. However, these differences have no bearing on individual components and classes.

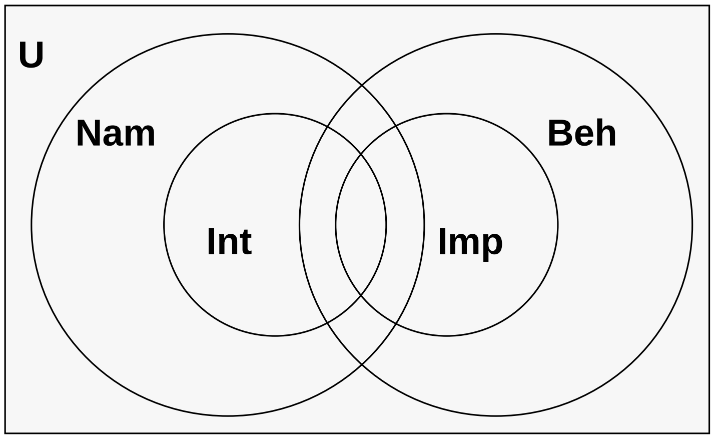


Figure 4-1. A framework for understanding OO subtyping relationships as regions in a space of type systems.

Furthermore, all existing type checking mechanisms regard types as either compatible or incompatible. Although it is beneficial to characterize component compatibility in this way, determining the *degree* of compatibility, and thus the potential for component interoperability, is more useful. One of the goals of the software architecture and component-based-development communities is to provide more extensive support for building systems out of existing parts. Those parts will typically not perfectly conform to each other. This dissertation will demonstrate that partially mismatched components can in certain cases still be effectively combined in an architecture (see Chapter 6). Establishing the degree of compatibility can also help determine the amount of work necessary to retrofit a component for use in a system.

The contributions of our method for supporting component evolution are threefold:

- a taxonomy that divides the space of potentially complex subtyping relationships into a small set of well defined, manageable subspaces;
- a flexible type theory for software architectures that is domain-, style-, and ADL-independent. By adopting a richer notion of typing, this theory is applicable to a broad class of design, evolution, and reuse circumstances; and
- an approach to establishing type conformance between interoperating components in an architecture. This approach is better suited to support large-scale development and OTS reuse facets of architecture research than other existing techniques.

4.1.1 General Principles of the Type Theory

When treating collections of architectural components as type hierarchies, architectural modeling involves

- identifying the types needed in an architecture (abstract components);
- selecting and evolving suitable existing types via subtyping (existing component specifications) to achieve the desired functionality; and
- creating new types (custom-designed components).

As discussed in above, OOPL subtyping alone is not sufficient to fulfill the evolution needs of software architectures. Instead, architectures need to expand upon the lessons learned from OOPLs and provide facilities for *heterogeneous* subtyping. To this end, we have developed a type theory for software architectures, represented by a framework for understanding component

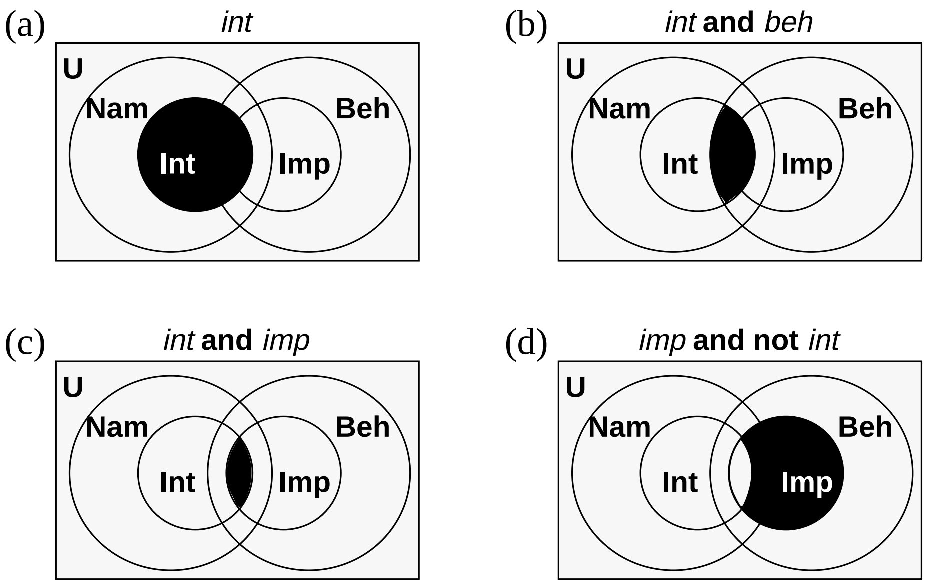


Figure 4-2. Examples of component subtyping relationships encountered in practice.

subtyping relationships as regions in a space of type systems, shown in Figure 4-1. The entire space of type systems is labeled *U*. The regions labeled *Int* and *Beh* contain systems that demand that two conforming types share interface and behavior, respectively. The *Imp* region contains systems that demand that a type share particular implementations of all supertype methods, which also implies that types preserve the behavior of their supertypes. The *Nam* region demands only shared method names, and thus includes every system that demands interface conformance.

Each subtyping relationship described by the Palsberg and Schwartzbach taxonomy [65] and summarized in Chapter 3 can be denoted via set operations on these regions. For example, *behavioral conformance*, which requires that both interface and behavior of a type be preserved, corresponds to the intersection of the Int and Beh regions and is expressed as *int* and *beh* (Figure 4-2b). Each region in Figure 4-1 encompasses a set of variations of a given subtyping relationship, rather than a single relationship. Thus, for example, the different flavors of the behavioral conformance relationship, described by Zaremski and Wing [99], represent different points in the *int* **and** *beh* subspace. The architectural type system we propose in the next section also represents a selection of individual points within the different subspaces.

This type theory is motivated by our specific experience with C2-style architectures, where we have encountered numerous situations in which new components are created by preserving one or more aspects of one or more existing components [48], [50]. Several examples are shown in Figure 4-2. We relate them to specific scenarios from the KLAX architecture described in Chapter 2:

- *interface conformance* (*int*) is useful when interchanging components without affecting dependent components. The *SpellingLogic* component in the Spelling KLAX architecture (Figure 2-6) used interface subtyping to provide a new implementation for the *TileMatchingLogic* component of the original architecture (Figure 2-5);
- *behavioral conformance* (*int* **and** *beh*) 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++. Behavioral subtyping results in sets of substitutable components, potentially 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 evolve a component with functionality that enables it to respond to queries from a debugger;
- *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 OO design pattern [19];
- *multiple conformance mechanisms* allow creation of a new type by subtyping from several types, potentially using different subtyping mechanisms. In the KLAX architecture, for example, *SpellingLogic* from Figure 2-6 was created by monotone subclassing of the *TileMatchLogic* component from the original architecture in Figure 2-5, and strictly monotone subclassing of an OTS spell checker.

These examples demonstrate that no single type conformance mechanism is adequate in describing all the subtyping relationships in a component hierarchy or an architecture. Note that we referred to the first three examples (Figure 4-2a-c) using the terminology from the Palsberg-Schwartzbach taxonomy. However, while in OOPLs the three subtyping mechanisms would be provided by three separate languages, in architectures they all may need to be supported by the same ADL and may actually be applied to components in a single architecture. Also, the example in Figure 4-2d does not have a corresponding OOPL mechanism, further motivating the need for a flexible type theory for software architectures. Relaxing the rules of a particular method to support our needs would sacrifice type checking precision. In order to describe typing relationships accurately while preserving type checking quality, we have opted to use multiple type conformance mechanisms to describe and evolve architecture-level components.

At the same time, by giving a software architect more latitude in choosing the direction in which to evolve a component, we allow some potentially undesirable side effects. For example, by preserving a component’s interface, but not its behavior, the component and its resulting subtype may not be interchangeable in a given architecture. However, it is up to the architect to decide whether to preserve architectural type correctness, in a manner similar to America [4], Liskov and Wing [40], Leavens et al. [16], [39], and others (depicted in Figure 4-2b), or simply to enlarge the palette of design elements in a controlled manner, in order to use them in the future.

4.1.2 Architectural Type System

In this section we present a type system for software architectures that instantiates the type theory. The two possible applications of an architectural type theory—evolution of existing components by software architects, and type checking of architectural descriptions—are discussed below in Sections 4.1.2.2 and 4.1.2.3, respectively. All definitions are specified in Z, a language for modeling mathematical objects based on first order logic and set theory [84]. Z uses standard logical connectives ($\vee$, $\wedge$, $\Rightarrow$, etc.) and set-theoretic operations ($\mathbb{P}$ to denote sets, $\in$, $\cup$, $\cap$, etc.). For a brief overview of Z, see Appendix A.

4.1.2.1 Components

Every component specification at the architectural level is an *architectural type*. We distinguish architectural types from *basic types* (e.g., integers, strings, arrays, records, etc.). Unlike OOPLs, in which objects communicate by passing around other objects, in software architectures components are distinguished from the data they exchange during communication. In other words, a “component” in the sense in which we use it here is never passed from one component in an architecture to another.

A component has a name, a set of interface elements, an associated behavior, and (possibly) an implementation. Each interface element has a direction indicator (*pro*vided or *req*uired), a name, a set of parameters, and (possibly) a result. Each parameter, in turn, has a name and a type.

A component’s behavior consists of an invariant and a set of operations. The invariant is used to specify properties that must be true of all component states. Each operation has preconditions, postconditions, and (possibly) a result. An operation is defined for a set of input states. Given an input state, the operation produces an output state. The sets of input and output states may or may not be disjoint. Since operations are decoupled from interface elements, they also provide a set of variables. Operation variables are used to express preconditions and postconditions; the values of operation variables and (relevant) component state variables at a given time represent the operation’s current state. Like interface elements, operations can be *pro*vided or *req*uired. Only provided operations will have an implementation in a given component. The preconditions and postconditions of required operations express the *expected* semantics for those operations. The relationship of component invariants to operation pre- and postconditions can be summarized as follows. Given a component C and operation O provided by C, for all valid input states of O that satisfy C’s invariant and O’s precondition, there exists a valid output state that satisfies both O’s postcondition and C’s invariant. Formal specification of an architectural type (component) is shown in Figure 4-3.[2]

Since we separate the interface from the behavior, we define a function, *int_op_map*, which maps every interface element to an operation of the behavior. This function is a total surjection: each interface element is mapped to a single operation, while each operation implements at least one interface. An interface element can be mapped to an operation only if the types of its parameters are subtypes of the corresponding variable types in the operation, while the type of its result is a supertype of operation’s result type. This property directly enables a single operation to export multiple interfaces.

4.1.2.2 Architectural Type Conformance

Informally, a subtyping relation, $\leq$, between two components, C_1 and C_2, is defined as the disjunction of the *nam*, *int*, *beh*, and *imp* relations shown in Figure 4-1:

$$(\forall C_1, C_2 : \mathtt{Component})(C_2 \leq C_1 \Leftrightarrow$$
$$C_2 \leq_{nam} C_1 \vee C_2 \leq_{int} C_1 \vee C_2 \leq_{beh} C_1 \vee C_2 \leq_{imp} C_1)$$

We consider these four relations in more detail below.

Name Conformance. Name conformance requires that a subtype share its supertype’s interface element names and all interface parameter names. The subtype may introduce additional interface

2. Capitalized identifiers are the basic (unelaborated) types in a Z specification. Trivial schemas and schemas whose meanings are obvious are omitted for simplicity.

$$\begin{array}{|l}
\hline \mathit{Variable} \\
\mathit{name} : \mathit{STRING} \\
\mathit{type} : \mathit{BASIC\_TYPE} \\
\mathit{value} : \mathit{BASIC\_TYPE\_INSTANCE} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \mathit{Int\_Element} \\
\mathit{dir} : \mathit{DIRECTION} \\
\mathit{name} : \mathit{STRING} \\
\mathit{params} : \mathbb{P}\ \mathit{Variable} \\
\mathit{result} : \mathit{BASIC\_TYPE} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \mathit{Operation} \\
\mathit{vars} : \mathbb{P}\ \mathit{Variable} \\
\mathit{precond} : \mathit{Logic\_Pred} \\
\mathit{postcond} : \mathit{Logic\_Pred} \\
\mathit{result} : \mathit{BASIC\_TYPE} \\
\mathit{dir} : \mathit{DIRECTION} \\
\mathit{implementation} : \operatorname{seq} \mathit{STATEMENT} \\
\mathit{in\_states} : \mathbb{P}\ \mathit{STATE} \\
\mathit{out\_states} : \mathbb{P}\ \mathit{STATE} \\
\mathit{current\_state} : \mathit{STATE} \\
\hline
\mathit{dir} = \mathit{REQ} \Rightarrow \mathit{implementation} = \varnothing \\
\mathit{current\_state} \in \mathit{in\_states} \lor \mathit{current\_state} \in \mathit{out\_states} \\
\mathit{vars} \subseteq \mathit{current\_state} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \mathit{Oper\_Computation} \\
\Delta \mathit{Operation} \\
\mathit{Computation} : \mathit{STATE} \leftrightarrow \mathit{STATE} \\
\hline
\#\mathit{vars}' = \#\mathit{vars} \\
\mathit{dir}' = \mathit{dir} \\
\mathit{implementation}' = \mathit{implementation} \\
\mathit{in\_states}' = \mathit{in\_states} \\
\mathit{out\_states}' = \mathit{out\_states} \\
\\
\operatorname{dom} \mathit{Computation} = \mathit{in\_states} \\
\operatorname{ran} \mathit{Computation} = \mathit{out\_states} \\
\mathit{current\_state} \in \mathit{in\_states} \\
\mathit{current\_state}' \in \mathit{out\_states} \\
\\
\forall\, \mathit{in}, \mathit{out} : \mathit{STATE} \mid \mathit{in} \in \mathit{in\_states} \land \mathit{out} \in \mathit{out\_states} \bullet \\
\qquad (\mathit{in}, \mathit{out}) \in \mathit{Computation} \\
\qquad\qquad \Leftrightarrow \\
\qquad \mathit{in} = \mathit{current\_state} \land \\
\qquad \mathit{out} = \mathit{current\_state}' \land \\
\qquad (\mathit{vars} \cup \mathit{precond}.\mathit{var\_operands}) \subseteq \mathit{in} \land \\
\qquad (\mathit{vars}' \cup \mathit{postcond}.\mathit{var\_operands}) \subseteq \mathit{out} \\
\hline
\end{array}$$

Figure 3-3 continues on the next page

$$\begin{array}{l}
\textit{Component} \\
\hline
\mathit{Basic\_Type\_Conformance} \\
\mathit{name} : \mathit{STRING} \\
\mathit{state\_vars} : \mathbb{P}\ \mathit{Variable} \\
\mathit{interface} : \mathbb{P}\ \mathit{Int\_Element} \\
\mathit{invariant} : \mathit{Logic\_Pred} \\
\mathit{operations} : \mathbb{P}\ \mathit{Operation} \\
\mathit{int\_op\_map} : \mathit{Int\_Element} \twoheadrightarrow \mathit{Operation} \\
\mathit{states} : \mathbb{P}\ \mathit{STATE} \\
\mathit{current\_states} : \mathbb{P}\ \mathit{STATE} \\
\hline
\operatorname{dom} \mathit{int\_op\_map} = \mathit{interface} \\
\operatorname{ran} \mathit{int\_op\_map} = \mathit{operations} \\
\mathit{current\_states} \subseteq \mathit{states} \\
\\
\forall\, o : \mathit{Operation} \mid o \in \mathit{operations} \bullet \\
\quad o.\mathit{vars} \cap \mathit{state\_vars} = \emptyset \land \\
\quad (o.\mathit{in\_states} \cup o.\mathit{out\_states}) \subseteq \mathit{states} \land \\
\quad o.\mathit{current\_state} \in \mathit{current\_states} \land \\
\quad o.\mathit{precond}.\mathit{var\_operands} \subseteq (o.\mathit{vars} \cup \mathit{state\_vars}) \land \\
\quad o.\mathit{postcond}.\mathit{var\_operands} \subseteq (o.\mathit{vars} \cup \mathit{state\_vars}) \\
\\
\forall\, \mathit{ie} : \mathit{Int\_Element};\ o : \mathit{Operation} \mid \\
\ \ \mathit{ie} \in \mathit{interface} \land o \in \mathit{operations} \bullet \\
\quad (\mathit{ie}, o) \in \mathit{int\_op\_map} \\
\qquad \Leftrightarrow \\
\quad \mathit{ie}.\mathit{dir} = o.\mathit{dir} \land \\
\quad (\mathit{ie}.\mathit{result}, o.\mathit{result}) \in \mathit{Basic\_Conf} \land \\
\quad (\forall\, \mathit{iv} : \mathit{Variable} \mid \mathit{iv} \in \mathit{ie}.\mathit{params} \bullet \\
\quad\ \ \exists\, \mathit{ov} : \mathit{Variable} \mid \mathit{ov} \in o.\mathit{vars} \bullet \\
\qquad\quad (\mathit{ov}.\mathit{type}, \mathit{iv}.\mathit{type}) \in \mathit{Basic\_Conf})
\end{array}$$

$$\begin{array}{l}
\textit{State\_Transition} \\
\hline
\Delta \mathit{Component} \\
\mathit{Oper\_Computation} \\
\hline
\mathit{name}' = \mathit{name} \\
\#\mathit{state\_vars}' = \#\mathit{state\_vars} \\
\mathit{interface}' = \mathit{interface} \\
\mathit{int\_op\_map}' = \mathit{int\_op\_map} \\
\mathit{states}' = \mathit{states} \\
\mathit{current\_states}' \neq \mathit{current\_states} \\
\\
\forall\, s : \mathit{STATE};\ o : \mathit{Operation} \mid s \in \mathit{current\_states} \land o \in \mathit{operations} \bullet \\
\quad s = o.\mathit{current\_state} \land s \in o.\mathit{in\_states} \\
\qquad \Rightarrow \\
\quad \mathit{state\_vars} \cup o.\mathit{vars} \subseteq s \land \\
\quad o.\mathit{precond}.\mathit{Value}(\mathit{state\_vars} \cup o.\mathit{vars}) = \mathit{TRUE} \land \\
\quad \mathit{invariant}.\mathit{Value}(\mathit{state\_vars} \cup o.\mathit{vars}) = \mathit{TRUE} \land \\
\quad (\exists\, s2 : \mathit{STATE} \mid s2 = \mathit{Computation}(s) \bullet \\
\quad s2 \in \mathit{current\_states}' \land \\
\qquad o \in \mathit{operations}' \land \\
\qquad \mathit{state\_vars}' \cup o.\mathit{vars} \subseteq s2 \land \\
\qquad o.\mathit{postcond}.\mathit{Value}(\mathit{state\_vars}' \cup o.\mathit{vars}) = \mathit{TRUE} \land \\
\qquad \mathit{invariant}.\mathit{Value}(\mathit{state\_vars}' \cup o.\mathit{vars}) = \mathit{TRUE})
\end{array}$$

Figure 4-3. Z specification of architectural types (components).

Logic_Pred is a schema that denotes a logical predicate, which is either a *Logical_Expression* or a *Boolean_Constant* (both evaluate to either *TRUE* or *FALSE*). A *Logical_Expression* consists of operands and operators. *STATE* is defined as a set of *Variable*s. Relation *Basic_Conf* is defined in the schema *Basic_Type_Conformance* and relates two basic types, the first of which is a supertype of the second.

$$\begin{array}{l}
\textit{Param\_Name\_Conformance} \\
\hline
\textit{Prm\_Nam\_Conf} : \textit{Int\_Element} \leftrightarrow \textit{Int\_Element} \\
\hline
\forall\, ie1, ie2 : \textit{Int\_Element} \bullet \\
\quad (ie1, ie2) \in \textit{Prm\_Nam\_Conf} \\
\qquad \Leftrightarrow \\
\quad (\forall\, p1 : \textit{Variable} \mid p1 \in ie1.params \bullet \\
\quad\ \exists_1 p2 : \textit{Variable} \mid p2 \in ie2.params \bullet \\
\qquad p1.name = p2.name \land \\
\qquad (\forall\, p3, p4 : \textit{Variable} \mid \\
\qquad\ p3 \in ie1.params \land p4 \in ie2.params \bullet \\
\qquad\quad p3.name = p4.name \\
\qquad\qquad \Leftrightarrow \\
\qquad\quad (p1 = p3 \land p2 = p4) \lor \\
\qquad\quad (p1 \neq p3 \land p2 \neq p4)))
\end{array}$$

$$\begin{array}{l}
\textit{Name\_Conformance} \\
\hline
\textit{Param\_Name\_Conformance} \\
\textit{Nam\_Conf} : \textit{Component} \leftrightarrow \textit{Component} \\
\hline
\forall\, c1, c2 : \textit{Component} \bullet \\
\quad (c1, c2) \in \textit{Nam\_Conf} \\
\qquad \Leftrightarrow \\
\quad (\forall\, ie1 : \textit{Int\_Element} \mid ie1 \in c1.interface \land ie1.dir = PROV \bullet \\
\quad\ \exists\, ie2 : \textit{Int\_Element} \mid ie2 \in c2.interface \land ie2.dir = PROV \bullet \\
\qquad ie1.name = ie2.name \land \\
\qquad (ie1, ie2) \in \textit{Prm\_Nam\_Conf}) \land \\
\quad (\forall\, ie2 : \textit{Int\_Element} \mid ie2 \in c2.interface \land ie2.dir = REQ \bullet \\
\quad\ \exists\, ie1 : \textit{Int\_Element} \mid ie1 \in c1.interface \land ie1.dir = REQ \bullet \\
\qquad ie1.name = ie2.name \land \\
\qquad (ie2, ie1) \in \textit{Prm\_Nam\_Conf})
\end{array}$$

Figure 4-4. Name Conformance.

elements and additional parameters to existing interface elements. Two interface elements in a single component can have identical names, but then their sets of parameter names must differ. Name conformance rules are formally specified in Figure 4-4.

Note that the possibility of introducing additional parameters to existing interface elements is different from method overloading and is typically not allowed in a programming language. However, software architectures are at a level of abstraction that is above source code and this feature may be supported by the architecture implementation infrastructure. For example, the C2 implementation infrastructure discussed in the next chapter allows the sender of the communication message to include parameters the receiver component does not expect; those parameters are simply ignored by the receiver. It is up to the architect to decide whether such a situation should be permitted in a given architecture.

Interface Conformance. Name conformance is a rather weak conformance requirement and we have encountered it in practice only as part of the stronger interface conformance relationship. Component C_2 is an interface subtype of C_1 if and only if it provides at least (but not necessarily only) the interface elements provided by C_1, and *matching* parameters and results for each interface element. Two parameters belonging to the two components' interface elements match if and only if they have identical names (*Param_Name_Conformance* schema in Figure 4-4) and

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, Param\_Conformance \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
Basic\_Type\_Conformance \\
Param\_Name\_Conformance \\
Prm\_Conf : Int\_Element \leftrightarrow Int\_Element \\
\hline
\forall\, ie1, ie2 : Int\_Element \bullet \\
\quad (ie1, ie2) \in Prm\_Conf \\
\qquad \Leftrightarrow \\
\quad (ie1, ie2) \in Prm\_Nam\_Conf \wedge \\
\quad (\forall\, p1, p2 : Variable \mid \\
\quad\; p1 \in ie1.params \wedge p2 \in ie2.params \wedge p1.name = p2.name \bullet \\
\qquad (p2.type, p1.type) \in Basic\_Conf) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, Interface\_Conformance \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
Name\_Conformance \\
Param\_Conformance \\
Int\_Conf : Component \leftrightarrow Component \\
\hline
\forall\, c1, c2 : Component \bullet \\
\quad (c1, c2) \in Int\_Conf \\
\qquad \Leftrightarrow \\
\quad (c1, c2) \in Nam\_Conf \wedge \\
\quad (\forall\, ie1, ie2 : Int\_Element \mid \\
\quad\; ie1 \in c1.interface \wedge ie2 \in c2.interface \wedge ie1.name = ie2.name \bullet \\
\qquad ((ie1.dir = ie2.dir = PROV \wedge (ie1, ie2) \in Prm\_Nam\_Conf) \Rightarrow \\
\qquad\quad ((ie1, ie2) \in Prm\_Conf \wedge (ie1.result, ie2.result) \in Basic\_Conf)) \wedge \\
\qquad ((ie1.dir = ie2.dir = REQ \wedge (ie2, ie1) \in Prm\_Nam\_Conf) \Rightarrow \\
\qquad\quad ((ie2, ie1) \in Prm\_Conf \wedge (ie2.result, ie1.result) \in Basic\_Conf))) \\
\hline
\end{array}
$$

Figure 4-5. Interface Conformance.

each parameter type of C_1 is a subtype of the corresponding parameter type of C_2 (*contravariance of parameters*, defined in the *Param_Conformance* schema in Figure 4-5). The results of two corresponding interface elements match if the result type in C_1 is a supertype of the result type in C_2 (*covariance of result*). For each interface element, the subtype must provide at least (but not necessarily only) the parameters that match the supertype's parameters.

The relationship among provided interface elements ensures that the subtype's functionality corresponding to a given interface element can always be accessed in the same manner as the supertype's functionality corresponding to the matching interface element. This means that the subtype can always be used in the place of the supertype.[3] To ensure that the same is true in the case of *required* interface elements, this relationship is reversed: the subtype can require *at most* the interface elements required by the supertype (with the appropriate relationship among parameters). If this were not the case, i.e., if the subtype required more than the supertype, the subtype could not be used in the place of the supertype. Interface conformance rules are formally specified in Figure 4-5.

Behavior Conformance. Behavior conformance requires that the invariant of the supertype be ensured by that of the subtype. Furthermore, each provided operation of the supertype must have a corresponding provided operation in the subtype (the subtype can also introduce additional

3. Note that this relationship only addresses interfaces and that it is possible for the subtype to provide behavior that is entirely different from the supertype's. This issue is addressed below.

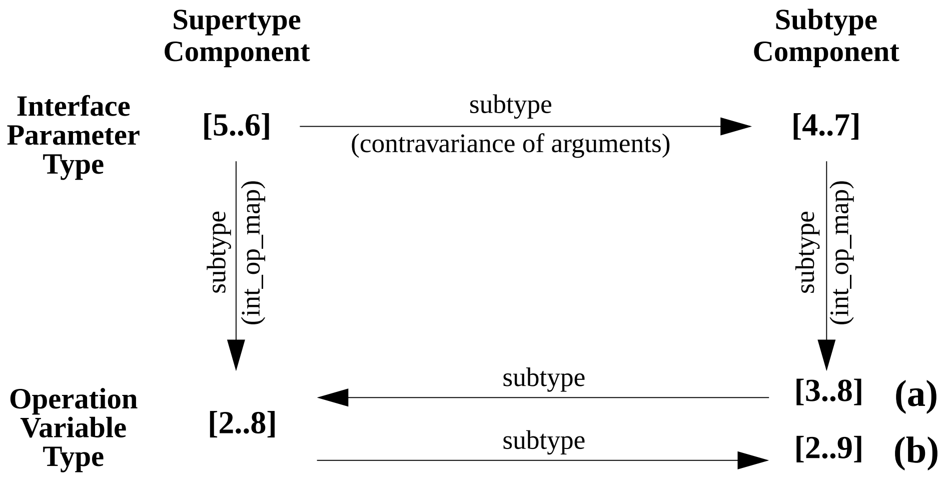


Figure 4-6. Relationship between supertype's and subtype's operation variable types. Contravariance of arguments and the *int_op_map* function do not guarantee a particular relationship between supertype's and subtype's operation variable types (illustrated using integer subranges): (a) supertype component's variable type is a supertype of subtype component's; (b) supertype component's variable type is a subtype of subtype component's.

operations), where the subtype's operation has the same direction indicator as the supertype's, the same or weaker preconditions, same or stronger postconditions, and preserves result covariance. This relationship is reversed for required operations; the argument for doing so is analogous to the one used above for interface elements.

No constraints are placed on the relationship between the types of the supertype's and subtype's corresponding operation variables. This relationship can vary, but is always an instance of one of the two cases depicted in Figure 4-6. Either relationship between the variable types is allowed so long as the proper relationships between operation pre- and postconditions are maintained. The rules for behavior conformance are specified in Figure 4-7.

The subtyping relationship that results from the combination of the *Behavior_Conformance* and *Interface_Conformance* schemas (in particular, the *Beh_Conf* and *Int_Conf* relations they define), and the mapping function, *int_op_map*, represents a point in the region depicted in Figure 4-2b. This relationship is similar to other notions of behavioral subtyping in that it guarantees substitutability between a supertype and a subtype in an architecture.

Implementation Conformance. Although useful in practice for evolving components, implementation conformance is not a particularly interesting relationship from a type-theoretic point of view. Implementation conformance may be established with a simple syntactic check if the operations of the subtype have identical implementations (both syntactically and semantically) as the corresponding operations of the supertype. Implementation conformance between two types thus also requires a behavioral equivalence between their shared operations, as shown in Figure 4-8. Note that establishing semantic equivalence between syntactically different implementations is undecidable in general. Techniques for making this problem tractable are outside the scope of this dissertation.

$$\begin{array}{l}
\textit{Oper\_Conformance} \\
\hline
\textit{Basic\_Type\_Conformance} \\
\textit{Logical\_Implication} \\
\textit{Oper\_Conf} : \textit{Operation} \leftrightarrow \textit{Operation} \\
\hline
\forall\, o1, o2 : \textit{Operation} \bullet \\
\quad (o1, o2) \in \textit{Oper\_Conf} \\
\qquad \Leftrightarrow \\
\quad (\forall\, v1 : \textit{Variable} \mid v1 \in o1.vars \bullet \\
\quad\ \exists\, v2 : \textit{Variable} \mid v2 \in o2.vars \bullet \\
\qquad (v1.type, v2.type) \in \textit{Basic\_Conf} \lor \\
\qquad (v2.type, v1.type) \in \textit{Basic\_Conf}) \land \\
\quad (o1.precond, o2.precond) \in \textit{Logic\_Impl} \land \\
\quad (o2.postcond, o1.postcond) \in \textit{Logic\_Impl} \land \\
\quad (o1.result, o2.result) \in \textit{Basic\_Conf}
\end{array}$$

$$\begin{array}{l}
\textit{Behavior\_Conformance} \\
\hline
\textit{Oper\_Conformance} \\
\textit{Logical\_Implication} \\
\textit{Beh\_Conf} : \textit{Component} \leftrightarrow \textit{Component} \\
\hline
\forall\, c1, c2 : \textit{Component} \bullet \\
\quad (c1, c2) \in \textit{Beh\_Conf} \\
\qquad \Leftrightarrow \\
\quad (c2.invariant, c1.invariant) \in \textit{Logic\_Impl} \land \\
\quad (\forall\, o1 : \textit{Operation} \mid o1 \in c1.operations \land o1.dir = PROV \bullet \\
\quad\ \exists\, o2 : \textit{Operation} \mid o2 \in c2.operations \land o2.dir = PROV \bullet \\
\qquad (o1, o2) \in \textit{Oper\_Conf}) \land \\
\quad (\forall\, o2 : \textit{Operation} \mid o2 \in c2.operations \land o2.dir = REQ \bullet \\
\quad\ \exists\, o1 : \textit{Operation} \mid o1 \in c1.operations \land o1.dir = REQ \bullet \\
\qquad (o2, o1) \in \textit{Oper\_Conf})
\end{array}$$

Figure 4-7. Behavior conformance.

Logic_Impl is a relation that denotes that the first element in the relation implies the second.

$$\begin{array}{l}
\textit{Implementation\_Conformance} \\
\hline
\textit{Behavior\_Conformance} \\
\textit{Imp\_Conf} : \textit{Component} \leftrightarrow \textit{Component} \\
\hline
\forall\, c1, c2 : \textit{Component} \bullet \\
\quad (c1, c2) \in \textit{Imp\_Conf} \\
\qquad \Leftrightarrow \\
\quad (c1, c2) \in \textit{Beh\_Conf} \land \\
\quad (c1.invariant, c2.invariant) \in \textit{Logic\_Impl} \land \\
\quad (\forall\, o1, o2 : \textit{Operation} \mid o1 \in c1.operations \land o2 \in c2.operations \bullet \\
\qquad (o1, o2) \in \textit{Oper\_Conf} \Rightarrow (o2, o1) \in \textit{Oper\_Conf} \land \\
\qquad (o2, o1) \in \textit{Oper\_Conf} \Rightarrow (o1, o2) \in \textit{Oper\_Conf} \land \\
\qquad o1.dir = PROV \Rightarrow o1.implementation = o2.implementation)
\end{array}$$

Figure 4-8. Implementation Conformance.

4.1.2.3 Type Checking a Software Architecture

In order to discuss type conformance of interoperating components, we must define an architecture that includes those components. There is no single, universally accepted set of guidelines for composing architectural elements. Instead, architectural topology depends on the

$$
\begin{array}{l}
\textit{Architecture} \\
\hline
components : \mathbb{P}\ Component \\
connectors : \mathbb{P}\ Connector \\
comp\_conn : Component \nrightarrow Connector \\
conn\_comp : Connector \leftrightarrow Component \\
conn\_conn : Connector \leftrightarrow Connector \\
Comm\_Link : Component \leftrightarrow Component \\
\hline
\operatorname{dom} comp\_conn = components \\
\operatorname{ran} comp\_conn = connectors \\
\operatorname{dom} conn\_comp = connectors \\
\operatorname{ran} conn\_comp = components \\
\operatorname{dom} conn\_conn = connectors \\
\operatorname{ran} conn\_conn = connectors \\
\operatorname{dom} Comm\_Link = components \\
\operatorname{ran} Comm\_Link = components \\
\\
\forall\, c : Component;\ b : Connector \mid \\
\quad c \in components \land b \in connectors \bullet \\
\qquad (c, b) \in comp\_conn \Rightarrow (b, c) \notin conn\_comp \land \\
\qquad (b, c) \in conn\_comp \Rightarrow (c, b) \notin comp\_conn \\
\\
\forall\, b1, b2 : Connector \mid b1 \in connectors \land b2 \in connectors \bullet \\
\qquad (b1, b2) \in conn\_conn \Rightarrow (b1 \neq b2 \land (b2, b1) \notin conn\_conn) \\
\\
\forall\, c1, c2 : Component \mid c1 \in components \land c2 \in components \bullet \\
\qquad (c1, c2) \in Comm\_Link \\
\qquad\qquad \Leftrightarrow \\
\qquad c1 \neq c2 \land \\
\qquad (\exists\, b1, b2 : Connector \mid b1 \in connectors \land b2 \in connectors \bullet \\
\qquad\qquad ((c1, b1) \in comp\_conn \land \\
\qquad\qquad (b2, c2) \in conn\_comp \land \\
\qquad\qquad (b1, b2) \in conn\_conn^{*}) \\
\qquad\qquad\qquad \lor \\
\qquad\qquad ((c2, b1) \in comp\_conn \land \\
\qquad\qquad (b2, c1) \in conn\_comp \land \\
\qquad\qquad (b1, b2) \in conn\_conn^{*})) \\
\hline
\end{array}
$$

Figure 4-9. Formal definition of architecture.

ADL in which the architecture is modeled, characteristics of the application domain, and/or the rules of the chosen architectural style. We therefore had to make certain choices in specifying properties of an architecture, partly influenced by our experience with C2:

- we model connectors explicitly, unlike, e.g., Darwin [43] and Rapide [42];
- we allow direct connector-to-connector links, unlike, e.g., Wright [3];
- finally, we assume certain topological constraints that are derived from the rules of the C2 style discussed in Chapter 2: a component is attached to single connectors on its top and bottom sides, while a connector can be attached to multiple components and connectors on its top and bottom.

None of the above choices is required by our type theory. It is indeed possible to provide a definition of architecture that reflects any other compositional guidelines. However, these kinds of decisions were necessary in order to formally specify and check type conformance criteria.

The formal definition of architecture is given in Figure 4-9. Connectors are treated simply as communication routing devices (as in C2); therefore, their definitions are omitted. Two components can interoperate if there is a communication link between them. This means that they

$$\begin{array}{l}
\textit{Minimal\_Type\_Conformance} \\
\hline
\textit{Interface\_Conformance} \\
\textit{Behavior\_Conformance} \\
\textit{Architecture} \\
\hline
\forall\, c1 : \textit{Component} \mid c1 \in \textit{components} \bullet \\
\exists\, c2 : \textit{Component} \mid c2 \in \textit{components} \land (c1, c2) \in \textit{Comm\_Link} \bullet \\
\quad (\exists\, ie1, ie2 : \textit{Int\_Element} \mid \\
\quad\; ie1 \in c1.\textit{interface} \land ie2 \in c2.\textit{interface} \bullet \\
\qquad ie1.\textit{name} = ie2.\textit{name} \land \\
\qquad ie1.\textit{dir} = \textit{REQ} \land ie2.\textit{dir} = \textit{PROV} \land \\
\qquad (ie1, ie2) \in \textit{Prm\_Conf} \land \\
\qquad (c1.\textit{int\_op\_map}(ie1), \\
\qquad\; c2.\textit{int\_op\_map}(ie2)) \in \textit{Oper\_Conf})
\end{array}$$

$$\begin{array}{l}
\textit{Full\_Type\_Conformance} \\
\hline
\textit{Interface\_Conformance} \\
\textit{Behavior\_Conformance} \\
\textit{Architecture} \\
\hline
\forall\, c1 : \textit{Component};\ ie1 : \textit{Int\_Element} \mid \\
\;\; c1 \in \textit{components} \land ie1 \in c1.\textit{interface} \land ie1.\textit{dir} = \textit{REQ} \bullet \\
\exists\, c2 : \textit{Component};\ ie2 : \textit{Int\_Element} \mid \\
\;\; c2 \in \textit{components} \land (c1, c2) \in \textit{Comm\_Link} \land \\
\;\; ie2 \in c2.\textit{interface} \land ie2.\textit{dir} = \textit{PROV} \bullet \\
\quad ie1.\textit{name} = ie2.\textit{name} \land \\
\quad (ie1, ie2) \in \textit{Prm\_Conf} \land \\
\quad (c1.\textit{int\_op\_map}(ie1),\ c2.\textit{int\_op\_map}(ie2)) \in \textit{Oper\_Conf}
\end{array}$$

Figure 4-10. Type conformance predicates.

are either on the opposite sides of the same connector or one can be reached from the other by following one or more connector-to-connector links (defined by the *Comm_Link* relation).

For example, in the KLAX architecture from Figure 2-5, there is a communication link between *StatusADT* and *StatusLogic* components (via a single connector-to-connector link). There is also a link between *StatusADT* and *TileMatchLogic* components (different sides of the same connector). On the other hand, there is no communication link between *StatusADT* and *ClockLogic* components: they are attached above the same (top-most) connector; however, the *Comm_Link* relation mandates that they be on different sides of a connector, which reflects C2's communication rules.

Given this definition of architecture, it is possible to specify type checking predicates. As already discussed, components need not be able to fully interoperate in an architecture. The two extreme points on the spectrum of type conformance are:

- *minimal type conformance*, where at least one service (interface and corresponding operation) required by each component is provided by some other component along its communication links; and
- *full type conformance*, where every service required by every component is provided by some component along its communication links.

They are defined in Figure 4-10. The predicates expressing the degree of utilization of a component's provided services in an architecture can be specified in a similar manner (see Appendix A).

Depending on the requirements of a given project (reliability, safety, budget, deadlines, and so forth), type conformance corresponding to different points along the spectrum may be adequate. What would be classified as a “type error” in one architecture may be acceptable in another. Therefore, architectural type correctness is expressible in terms of a percentage corresponding to the degree of conformance (per component or for the architecture as a whole).

Type Conformance and Off-the-Shelf Reuse. Establishing type conformance brings up the question of how much a component may know about other components with which it will interoperate. Although magnified by our separation of provided from required component services, this issue is not unique to our type theory. Rather, it is pertinent to all approaches that model behavior of a type and enforce behavioral conformance.

To demonstrate behavioral conformance between two interoperating components, by definition one must show that a specific relationship holds between their respective behaviors. This relationship is one of several flavors of equivalence or implication, summarized in [99].

Establishing whether two components can interoperate includes matching the specification of what is expected by a required operation of one component against what another component’s provided operation supplies. Behavior of an operation is modeled in terms of its interface parameters (in our approach, operation variables) and component state variables. A component may thus need to refer to state variables that belong to another component in order to specify a *required* operation’s expected behavior. However, doing so would be a violation of the “provider” component’s abstraction. It would also violate some basic principles of component-based development:

- the designer may not know in advance which, if any, components will contain a matching specification for the required operation and, thus, what the appropriate (types of) state variables are. This is particularly the case when using behavior matching to aid component discovery and retrieval;
- large-scale, component-based development treats an off-the-shelf component as a black box, thereby intentionally hiding the details of its internal state. Having to explicitly refer to those details would require them to be exposed.

Existing approaches to behavior modeling and conformance checking have not addressed this problem. The problem does not apply to component subtyping: the designer must know all of existing component’s details in order to effectively evolve it. Thus, those approaches that focus on behavioral subtyping (e.g., America [4], Liskov and Wing [40], and Leavens et al. [16], [39]) do not encounter this problem. Zaremski and Wing [99] do address component retrieval and interoperability. However, their approach makes the very assumption that the designer will have access to a “provider” component’s state (via a shared Larch trait [30]). Fischer and colleagues [18], [79] model components at the level of a single procedure. In order to be able to properly specify pre- and postconditions, they include all the necessary variables as procedure parameters. Thus, for example, the stack itself is passed as a parameter to the *push* procedure.

The solution to this problem we propose is based on two requirements arising from a more realistic assessment of component-based development:

- we do not have access to a “provider” component’s internal state (unlike Zaremski and Wing’s approach), and
- we cannot change the way many software components, especially in the OO world, are modeled (unlike Fischer et al.).

These two requirements result in an obvious third requirement:

- we must somehow refer to a "provider" component's state when modeling operations, even though we do not know what that state is.

This seeming paradox actually suggests our solution. We model a required operation as if we have access to a "provider" component's state. However, since we do not know the actual "provider" state variables or their types, we introduce a generic type, STATE_VARIABLE, which is a supertype of all basic types. Thus, variables of this type are essentially placeholders in logical predicates. When matching, e.g., a required and provided precondition, we attempt to unify (instantiate) each variable of the STATE_VARIABLE type in the required precondition with a corresponding state variable in the provided precondition. If the unification is possible and the implication (with all instances of STATE_VARIABLE placeholders replaced with actual variables) holds, then the two preconditions conform.

4.1.3 Summary

This section defined the major elements of our type theory: multiple subtyping relationships (Section 4.1.2.2) and type conformance (Section 4.1.2.3). The type theory couples the rigor necessary to ensure the substitutability of a subtype in the place of its supertype (by combining interface and behavior conformance) with the unprecedented flexibility of evolving a component via heterogeneous subtyping. Certain characteristics of our type theory are unique (e.g., separation of interface from behavior) and give rise to seemingly anomalous relationships when considered in isolation (e.g., supertype and subtype operation variable types depicted in Figure 4-6). However, the type theory as a whole supplies mechanisms that prevent any such anomalies. For example, the *int_op_map* function constrains the actual use of operation variables with the types of interface parameters through which the variables are accessed. The desired relationship between a supertype's and subtype's operation variables is thus ensured.

4.2 Connector Evolution

One of the greatest benefits of architectures is their *separation of computation from interaction* in a system. Software connectors are elevated to a first-class status, and their benefits have been demonstrated by a number of existing approaches [3], [82], [89]. Software connectors are a key enabler of this dissertation's support for architectural configuration evolution. The specific role of connectors is to remove from components the responsibility of knowing how they are interconnected and isolate all decisions regarding communication, mediation, and coordination in a system. At the same time, connectors also introduce a layer of indirection between components. Any potential penalties paid due to this indirection should be outweighed by other benefits of connectors, such as their role as facilitators of evolution. Connectors with a rigid structure and static interfaces hamper evolution. One of the hypotheses of our work is that flexible, i.e., evolvable connectors result in increased software adaptability by more easily accommodating changes to their attached components.

Existing architectural approaches have by and large sacrificed the potential flexibility introduced by connectors in order to support more powerful architectural analyses. For example, Wright [3] and UniCon [82] require the architect to specify the types of component *ports* and *players*, respectively, that can be attached to a given connector *role*. Furthermore, although some variability is allowed in specifying the number of components that a given connector will be able

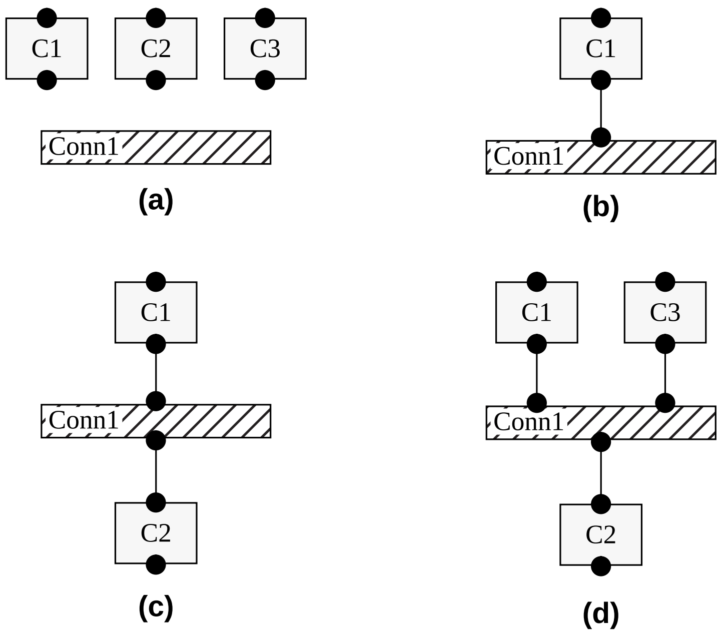


Figure 4-11. C2 connectors have *context reflective interfaces*.
Each C2 connector is capable of supporting any number of C2 components.
(a) Software architect selects a set of components and a connector from a design palette. The connector has no communication ports, since no components are attached to it.
(b-d) As components are attached to the connector to form an architecture, the connector creates new communication ports to support component intercommunication.

to support (parameterized number of roles in Wright; potentially unbounded number of players with which each role may be associated in UniCon), once these variables are set at architecture specification time, neither approach allows their modification.

Evolvability was a guiding principle in the design of the connectors introduced and employed in this dissertation. We achieve connector evolvability by assigning two properties to connectors: context-reflective interfaces and varying degrees of information exchange. Each is discussed below. Using the two techniques in tandem gives us unparalleled flexibility in evolving connectors and, in turn, supporting the evolution of architectures. The connectors discussed in this dissertation are also part of the C2 architectural style; for simplicity we will refer to them as “C2 connectors” below.

4.2.1 Context-Reflective Interfaces

A unique aspect of C2 connectors, and a direct facilitator of architectural evolution, are their *context-reflective interfaces*. This is a very simple concept, but one that has powerful consequences. A connector does not export a specific interface. Instead, it acts as a communication conduit which, in principle, supports communication among any set of components. The number of connector ports is not predetermined, but changes as components are attached or detached. This allows any C2 connector to support arbitrary addition, removal, replacement, and reconnection of components or other connectors, as shown in Figure 4-11. In other words, C2 connectors are inherently evolvable. The “interface” exported by a C2 connector is thus a function of its attached components’ interfaces.

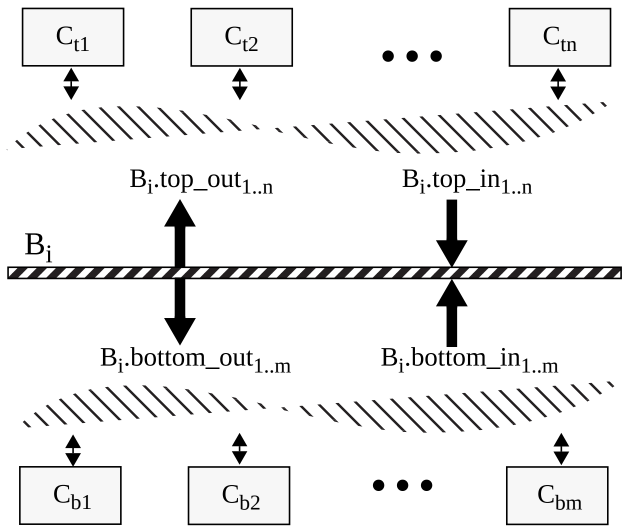


Figure 4-12. The interface of a C2 connector is a function of the interfaces of its attached components.

In order to elaborate more formally on this relationship between the interface of a C2 connector and the interfaces of its attached components, we consider a static, “snapshot” view of a connector. Figure 4-12 shows a C2 connector B_i, with the components C_{tj} and C_{bk} attached to its top and bottom respectively. The connector’s upper and lower domains of discourse (i.e., the connector’s interface) are completely specified in terms of these components’ interfaces.

Consider the notifications that come in from the components C_{tj} above the connector:

$$B_i.top\_in = \bigcup_j C_{tj}.bottom\_out$$

Since connectors have the ability to filter messages, as discussed below, the notifications that are emitted out of the bottom of a connector are a subset of the notifications that the connector receives from above. Thus, for each connector B_i, it is possible to identify the function *Filter_TB*, such that

$$B_i.bottom\_out = Filter\_TB(B_i.top\_in)$$

Similarly, consider the requests that come in from the components C_{bk} below the connector:

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

If the connector also filters requests, the requests that come out of its top are a subset of those that come in from below, so the function *Filter_BT* is defined as follows

$$B_i.top\_out = Filter\_BT(B_i.bottom\_in)$$

Therefore, a C2 connector’s interface is defined by the unions of the interfaces of the components above and below it, along with any filtering that the connector does to those interfaces. The interface will evolve dynamically as components are added, removed, and/or replaced.

4.2.2 Degree of Information Exchange

Another direction in which connectors may evolve deals with the amount of information that is shared between two components that communicate through a given connector. Our practical experience has indicated that, under certain circumstances, architectures can describe meaningful functionality even if their components cannot fully interoperate. In other cases, it is preferable to limit the degree of information exchange among components, e.g., to eliminate undesired behavior or improve performance. Enabling the architect to make the relevant decisions and change them during the lifetime of an architecture is thus critical. Isolating those changes to an appropriate construct will minimize the amount of modification to the architecture necessary to enact the changes. Since connectors are responsible for controlling all interaction among components, they are the natural construct to manage this aspect of interaction.

Information filtering constitutes a spectrum along which different amounts of information may be filtered out (or, conversely, propagated) by a connector. The two extremes are *no interaction*, meaning that the connector allows no information to be exchanged between two or more components, and *partial communication*, meaning that the connector essentially broadcasts all the information it receives to all of the attached components, regardless of whether they need or even understand the information.

Different points along this spectrum correspond to the current support for message exchange and filtering in C2 connectors, discussed in Chapter 2:

- A connector supports *no interaction* among its attached components if it employs the *message sink* filtering mechanism, whereby it filters out all messages it receives.
- *Full communication* corresponds to point-to-point information exchange in an architecture, achieved by using connectors that employ the *message filtering* mechanism. This is the optimal information exchange policy, since every component receives only those messages it needs and emits only the messages needed by other components in the architecture.
- As already discussed, a C2 connector that broadcasts messages to all of its attached components supports *partial communication*. The messages not understood by a given component are simply ignored (hence the communication is partial). Partial communication actually represents a range, rather than a single value, along the spectrum of information exchange, depending on whether only certain or all messages are broadcast by the connector. A variant of partial communication may occur in connectors that employ *notification filtering*, whereby a component receives only those notifications for which it explicitly registers. Since C2 components are substrate independent and thus are unable to register for requests they receive, partial communication occurs if the connector broadcasts the component's requests to all components above it.

The notions of no interaction and partial and full communication are formally defined as part of the formal specification of the C2 style, given in Appendix A.

A connector evolves by supporting different filtering mechanisms. Connectors do not provide any application-level functionality. However, their evolution may alter the behavior of an application. For example, a broadcast connector can evolve into a point-to-point connector without affecting the functionality of the encompassing architecture (although this is likely to improve the application's performance, as no unnecessary message traffic occurs). On the other hand, if a broadcast connector is evolved into a message sink, it would eliminate the portion of the behavior resulting from the interactions among its attached components.

4.3 Architectural Configuration Evolution

In addition to the evolution of individual components and connectors, architecture-based evolution can also occur at the level of their configurations. This section discusses the issues in configuration evolution and techniques employed by this dissertation to support it. The specific aspects of configuration evolution on which we have focused are addition, removal, replacement, and reconnection of components. Our support for evolving architectural configurations is based on

- employing evolvable connectors,
- providing heterogeneous connectors, and
- minimizing interdependencies among components in an architecture.

Evolvable connectors were presented in the preceding section. The remaining two categories are discussed below. Although this dissertation focuses on configuration evolution at specification-time, various facets of our approach have been critical in supporting the run-time evolution of C2-style architectures [61].

4.3.1 Minimal Component Interdependencies

Several C2 properties serve to provide added degrees of freedom in composing components in an architecture by minimizing component interdependencies: substrate independence, implicit invocation, and asynchronous communication. Each individual property aids the addition, removal, and interchange of components in a given architectural configuration. The three properties are related and together form a powerful tool for evolving C2-style architectures. They have been discussed in some depth in Chapter 2. We revisit them here specifically in the context of configuration evolution.

4.3.1.1 Substrate independence

A component in a C2-style architecture is not aware of the components below it. In particular, all notifications of component operation results, changes in the component's internal object, or of the component's entire current state are generated without knowing whether any component(s) will receive them and respond. Thus, the effects of adding, removing, or replacing a component in an architecture depend upon the component's position in the architectural configuration. For example, one *GraphicsBinding* component in Figure 2-5 may be substituted for another without any effect on the rest of the architecture, as no components have any explicit knowledge of the *GraphicsBinding*. Analogously, replacing components "higher" in an architecture affects all components below them. This effect is minimized by C2's use of domain translation, discussed in Chapter 2, and asynchronous communication via implicit invocation, discussed below.

4.3.1.2 Implicit Invocation

In the C2 style, implicit invocation occurs when a component reacts to a notification sent down an architectural configuration by invoking certain functionality. The internal architecture of a C2 component directly enables implicit invocation: the component that issues a notification does not know whether the notification will cause any reaction, while the notification itself does not explicitly name entry points into the objects of any components below. Even when a component issues a request, it does not explicitly name entry points into a component above it. Instead, the connectors above the component may route the request message to multiple possible recipients. If, due to component removal, no one is able to respond to the request, the issuing component may be affected (see Chapter 2); however, due to the asynchronous nature of communication (discussed below), the rest of the application can still continue with operation.

In the case of both notification and request messages, a recipient component's dialog attempts to respond to the message by invoking appropriate internal object operations. If it is unable to interpret the message, i.e., if the message is not in its domain of discourse (interface), the dialog simply ignores it. Implicit invocation thus greatly simplifies the process of adding a component to an architecture.

The internal component architecture also enables easy component replacement. A single internal object can export multiple message interfaces (via different dialogs). Conversely, multiple internal objects can export the same message interface. Interchanging two components that export *identical* interfaces is straightforward. Doing so for components with *similar* interfaces may result in partial communication and/or component service utilization, but, in general, the application will continue functioning at least in a degraded mode, as already discussed.[4]

4.3.1.3 Asynchronous Communication

All communication between C2 components is asynchronous and is solely achieved by exchanging messages. Message-based communication is extensively used in distributed applications, for which C2 is intended. While the style does not forbid synchronous communication, the responsibility for implementing synchronous message passing resides with individual components.

Asynchronous, message-based communication has direct ramifications on the evolvability of a given architectural configuration. It enables the resulting system to perform in a degraded mode if components fail and/or are being removed or replaced: any components that issue requests to a failed or removed component may continue functioning until those requests are answered at a later time; substrate independence mandates that no components whose notifications may reach the failed component will have any dependencies on it. Components may be added to a configuration at arbitrary times. They can immediately start responding to notifications (by issuing requests and their own notifications). If requests are broadcast, newly added components can also start responding to those requests they understand and thus improve the system's performance and accuracy.

4.3.2 Heterogeneous Connectors

The evolution of architectural configurations can also be facilitated by providing heterogeneous connectors, such that the behavior modeled in the architecture is preserved or modified, as desired. The encapsulation provided by a C2 connector allows variation of the low-level interaction mechanism it employs in a manner that is transparent to the architecture that includes the connector, the connector's attached components, or software architects who may want to employ the connector. This variation can happen along three different dimensions, shown in Figure 4-13:

- by interchanging connectors that support different types of interaction,
- by interchanging connectors that support different degrees of concurrency, and
- by interchanging connectors that support different degrees of information exchange.

4. This discussion has not addressed the effects of the difference in the two components' functionalities on the architecture. Our assumption is that the architect is aware of this difference and its impact.

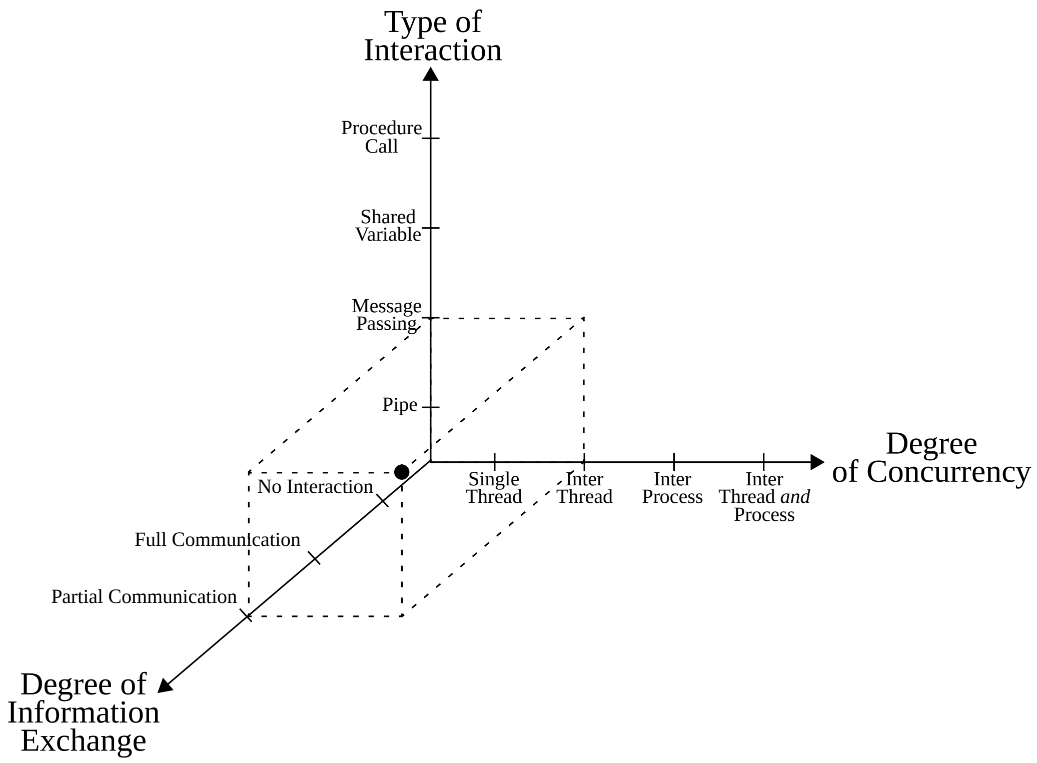


Figure 4-13. Dimensions of connector variation.
The *Type of Interaction* axis represents a set of discrete values with a nominal ordering.
The *Degree of Concurrency* axis is a set of discrete values with an ordinal ordering.
The *Degree of Information Exchange* axis is a continuum with an ordinal ordering of values.
Example values are shown along each axis. A single point in the three-dimensional space represents a particular kind of connector. One such connector is highlighted: a message broadcasting, inter-thread connector.

A point in this three-dimensional space represents a specific kind of connector. The *degree of information exchange* dimension represents a direction in which a C2 connector itself may evolve and was discussed in the preceding section. Different points along the remaining two dimensions deal with implementation-level issues and result in *different* connectors. Therefore, we do not consider them examples of *connector* evolution. They are discussed in more detail below.

4.3.2.1 Type of Interaction

In architectures, connectors may, e.g., be separately compilable message routing devices, shared variables, table entries, buffers, instructions to a linker, dynamic data structures, procedure calls, initialization parameters, client-server protocols, pipes, SQL links between a database and an application, and so forth [25], [82]. The corresponding axis in the space of Figure 4-13 has a nominal ordering of values.

Since a connector provides a distinct abstraction that isolates architecture-level communication, mediation, and coordination, it may be possible to evolve an architecture by replacing an existing connector with a connector that supports a different type of interaction. For example, a message passing connector may be replaced with an RPC connector. It should be noted that connectors that employ certain types of interaction will not be interchangeable without additional modifications to the components that communicate through them. For example, procedure calls are typically synchronous, whereas shared variables may be used asynchronously in an application; to effectively substitute one for the other, the application itself may need to be (partially) redesigned.

In this dissertation, we have employed connectors that support two types of interaction: asynchronous message passing and RPC. We have used message passing connectors very extensively. We employed RPC to a lesser degree, in the context of a specific OTS middleware technology, discussed in Chapter 6. A C2 architecture is easily evolved to use one or the other type of connector or it can employ both simultaneously. Note that certain types of connectors cannot be employed in a C2 architecture because of the mismatch between their interaction mechanisms (e.g., shared variables) and the properties of the C2 style (e.g., no assumption of shared address space across components).[5]

4.3.2.2 Degree of Concurrency

Another distinction is among connectors that enable interaction between components executing in the same thread of control (no concurrency) or in different threads of control. The granularity of concurrency supported by a given connector ranges from fine-grained, where all components execute in different threads but in the same operating system process, to coarse-grained, where every component executes in a different process. An architecture whose implementation employs single-thread connectors can be easily evolved into a distributed architecture by replacing those connectors with inter-process connectors.

Effective support for inter-thread and inter-process communication is of particular importance to us: it is essential for distributed applications, for which C2 is specifically intended. We have built support for single-thread, inter-thread, and inter-process connectors. This has enabled us to easily distribute C2-style architectures across process and machine boundaries. For example, the KLAX architecture discussed in Chapter 2 was easily distributed across three processes by using an inter-process C2 connector, as shown in Figure 4-14. In creating inter-thread or inter-process C2 connectors, all decisions that deal with concurrency support are encapsulated inside a connector. The benefits of this approach are twofold:

- the architect is not required to have any knowledge of the actual communication mechanism employed by the connector, but can instead use connectors that support any degree of concurrency in exactly the same manner;
- additional inter-thread or inter-process C2 connectors can be created simply by replacing the communication-mechanism-specific portion of an existing C2 connector.

When possible, we exploit the threading mechanisms provided in the underlying programming languages to support inter-thread connectors. However, if a programming language does not provide threading support, and in order to enable interconnection of parts of a C2 application executing in multiple processes and possibly on multiple machines, this dissertation provides three additional techniques. These techniques are independent of the choice of

5. Recall from Chapter 2 that such restrictions only apply to a conceptual architecture and do not constrain its implementation. Hence, it is indeed possible to implement C2 components to interact via shared variables. However, the responsibility of ensuring that an application implemented in such a manner adheres to its architecture resides entirely with developers. Furthermore, specific implementation-level techniques and tools must be provided to support the evolution of components implemented in this manner, as well as their configurations.

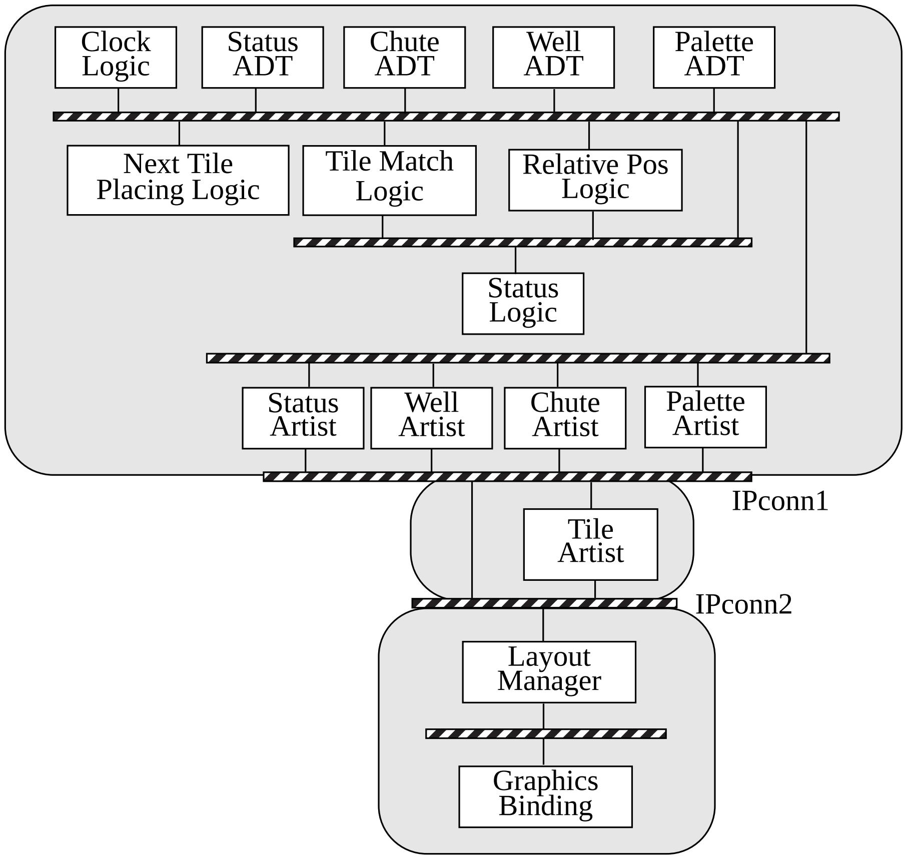


Figure 4-14. Multi-process implementation of the KLAX architecture.
Shaded ovals represent process boundaries. The application exhibits the same behavior as "original" KLAX.

underlying inter-process communication mechanism.[6] One divides an architecture along a single communication port, while the other two do so along a connector (as in Figure 4-14).

Linking Ports across Process Boundaries. The first method we implemented involves linking two C2 ports across a process or a machine boundary, using an interprocess communication mechanism to bridge those boundaries. All accesses to the middleware technology would be entirely encapsulated within the port entity and would not be visible to architects or developers. The single-process implementation of a C2 connector links two ports together by having each port contain a reference to the other one. In this way, the ports can call methods on each other, sending communication messages as method parameters.

One potential solution is to simply exchange port references across process boundaries and use the existing, single-process technique for message passing. This strategy is infeasible for several reasons, however. Most importantly, ports are complex objects and are not easily serializable. Typically, any objects sent across a process or network boundary must first be serialized into a byte stream. C2 ports contain references to complex C2 objects to which they are attached (connectors, components, and entire architectures), which would, in turn, also have to be

6. The same arguments apply to both inter-thread and inter-process connectors in the remainder of the section. For simplicity, we refer only to inter-process connectors.

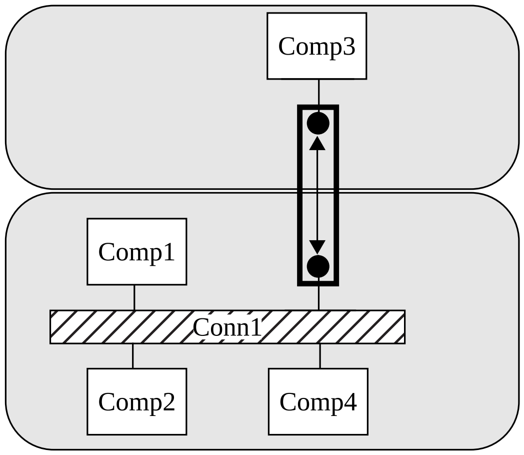


Figure 4-15. Linking ports across process boundaries.
Two communication ports in separate processes comprise a single “virtual port.” For clarity, we do not highlight component and connector ports. Shaded ovals represent process boundaries.

serialized, rendering this approach impractical. Secondly, references to objects are typically not preserved across process boundaries since all network data is passed by copy instead of by reference. Thus, even if we could overcome the serialization issue and pass port objects through the network, a connection made by using the references to them would not be preserved over the network.

Instead of passing complex objects across a process or network boundary, we refine the approach to simply send messages. Messages consist only of data and are easily marshaled. This method results in the creation of two ports, one per process, to simulate a single “virtual port,” as shown in Figure 4-15. Rather than sending a reference to itself to the other port, each port simply sends messages.

Linking Connectors across Process Boundaries. Sharing communication ports across process boundaries gives us fine-grained control over implementing an architecture as a multi-process application. However, it fails to isolate the change to the appropriate abstraction: the connector. In order to remedy this, we provide two connector-based methods, both of which consist of implementing a single conceptual connector using two or more actual connectors that are linked across process or network boundaries. Each actual connector thus becomes a segment of a single “virtual connector.” All access to the underlying inter-process communication mechanisms is encapsulated entirely within the abstraction of a connector, i.e., it is hidden from both architects and developers.

We call the first method “lateral welding,” depicted in Figure 4-16a. Messages sent to any segment of the multi-process connector are broadcast to all other segments. Upon receiving a message, each segment has the responsibility of filtering and forwarding it to components in its process as appropriate. Only messages are sent across process boundaries.

While the lateral welding approach allowed us to “vertically slice” a C2 architecture, we also provide an approach to “horizontally slice” an application, as shown in Figure 4-16b. This approach is similar to the idea of lateral welding: a conceptual connector is broken up into top and bottom segments, each of which exhibits the same properties as a single-process connector to the

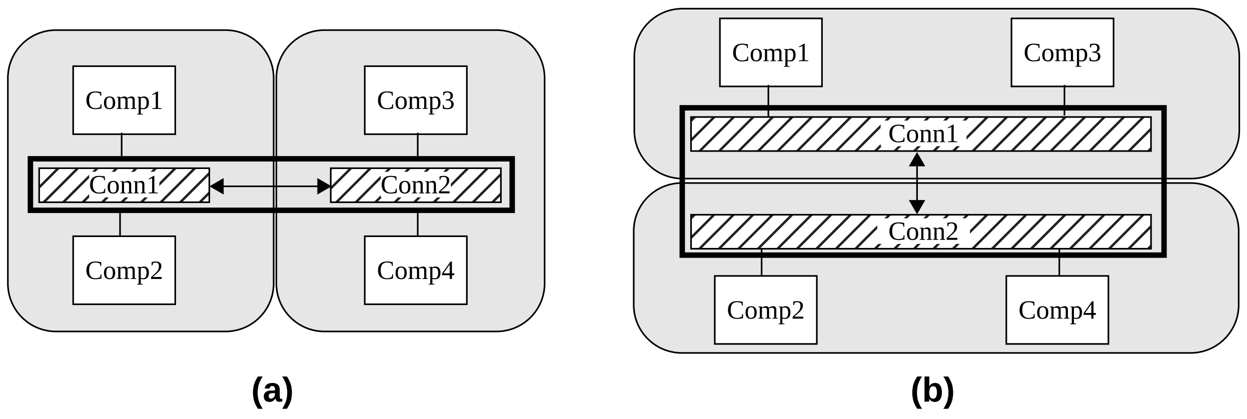


Figure 4-16. Linking connectors across process boundaries.
Connectors are the primary vehicle for interprocess communication. A single conceptual connector can be "broken up" (a) vertically or (b) horizontally for this purpose. Shaded ovals represent process boundaries.

components attached above and below it, respectively. The segments themselves are joined across process boundaries using the appropriate middleware.

4.3.3 Discussion

It is important to note that, strictly speaking, the three-dimensional space shown in Figure 4-13 is not defined at every point. The three concerns (type of interaction, level of concurrency, and degree of information exchange) are not fully orthogonal. Instead, some types of connectors imply a particular execution thread/process structure and degree of information exchange. For example, procedure calls are typically employed inside a single thread of control. Furthermore, procedure calls require full communication: in order to compile an application, every procedure call must correspond to a procedure definition, it is never broadcast, and cannot be ignored. Thus, it would not make sense to talk about procedure call connectors that support partial communication.

Although this is not a particular focus of this dissertation, it may be possible in general to modify a given connector type to fully separate the three concerns. For example, *RPC* is an inter-thread/process counterpart to *procedure calls*. Similarly, shared variable connectors assume a single address space, which is not the case with components that execute in multiple processes. Replacing *shared variable* connectors with connectors that enable communication by reading from and writing to specific blocks of memory accessible to all of the interacting components (*shared memory*) can remedy this problem. Finally, connectors that implement certain types of interaction can be used to simulate other types of interaction if the appropriate mechanisms are unavailable. For example, we have demonstrated the ability to simulate asynchronous message broadcast via RPC (see Chapter 6).

4.4 An Architecture Description and Evolution Language

In order to render the evolution concepts introduced in this dissertation and discussed in this chapter usable in practice, we have designed an ADL that incorporates them. This section introduces C2SADEL, a *S*oftware *A*rchitecture *D*escription and *E*volution *L*anguage for C2-style architectures. The complete specification of C2SADEL's syntax is given in Appendix A. C2SADEL supports component evolution via heterogeneous subtyping and facilitates architectural descriptions that allow establishment of type-theoretic notions of architectural soundness. It also supports modeling of connectors with context-reflective interfaces and different data filtering capabilities, as well as configurations that adhere to the topological rules of the C2 style.

We encountered a tension between formality and practicality in designing C2SADEL. Our goal was a language that was simple enough to be usable in practice, yet formal enough to adequately support analysis and evolution. For this reason, we kept the syntax simple and reduced formalism to a minimum.

A C2SADEL specification consists of either a set of component types or of an architecture. An architecture contains a specification of component types, connector types, and topology. To properly specify an architecture's topology, component and connector types are instantiated and connected.

4.4.1 Component Types

A component specification is a type that can be defined in-line or externally (using the keyword *extern*). The specification of an external component type is given in a file different from the file in which the rest of the architecture is specified. For example,

```
component WellADT is extern {WellADT.c2;}
```

specifies that the *WellADT* component used in the KLAX architecture shown in Figure 4-14 on page 62 is specified in the file *WellADT.c2*. This feature allows for components to be treated as reusable design elements, independent of an architecture. A component type consists of the following:

- state variables,
- component invariant,
- interface,
- behavior, and
- the map from interface elements to the operations of the behavior. This map is a surjective function, as discussed in Section 4.1.

A component type may be a *subtype* of another type. The exact subtyping relationship must be specified. Keywords *nam*, *int*, *beh*, and *imp* are used to denote name, interface, behavior, and implementation conformance, respectively. Different combinations of these relationships, corresponding to the different areas in the space of type systems shown in Figure 4-1, are specified using the keywords *and* and *not*. For example,

```
component WellADT is subtype Matrix (beh)
```

specifies that the KLAX component *WellADT* preserves (and possibly extends) the behavior of a component *Matrix*, but may change its interface. This relationship can be made stricter by specifying that *WellADT must* alter *Matrix*'s interface as follows:

```
component WellADT is subtype Matrix (beh \and \not int)
```

The rules defined in Section 4.1 are used to ensure that the specified subtyping relationship between two components holds.

As in a programming language, variables are specified as <name, type> pairs, as in

```
capacity : Integer;
```

Additionally, a component's state variable may also be specified as a function:

```
well_at : Integer -> Color;
```

The *well_at* function maps a set of *Integer* locations in the well to a set of *Color* tiles at each location.

Variable types in C2SADEL, such as *Integer* or *Color*, are *basic* types and are distinguished from components, which are *architectural* types. We do not explicitly model the semantics of basic types; however, C2SADEL does allow the architect to specify that one basic type is a subtype of another:

```
Natural is basic_subtype Integer;
```

Such relationships allow the creation of basic type hierarchies. This, in turn, enables proper mapping of interface elements to operations and checking of contra- and covariance rules in architectural type conformance checking, specified in Section 4.1.

A component's invariant is a conjunction of predicates specified in first-order logic. The invariant defines a set of conditions that must be satisfied throughout the component's execution. It is specified with component state variables as operands and logical operators (*\and*, *\or*, *\not*, *\implies*, and *\equivalent*), comparison operators (*\greater*, *\less*, *\eqgreater*, *\eqless*, =, and <>)[7], set operators (*\union*, *\intersection*, *\in*, *\not_in*, and *#*)[8], and arithmetic operators (+, -, *, /, and ^)[9]. This set of operators is intended to be extensible as needed; the currently supported operators, described above, have been sufficiently expressive to describe the properties of C2-style applications to date (see Chapters 2 and 6). Operator precedence in C2SADEL is defined as shown in Table 4-1.

Table 4-1: Operator Precedence in C2SADEL, Given in Descending Order

Operator Precedence
#, \not
^
*, /
+, -
\union, \intersection
\greater, \eqgreater, \less, \eqless, =, <>, \in, \not_in
\and, \or
\implies, \equivalent

For example, the invariant for the *WellADT* component can be specified as follows.

```
invariant {
   (num_tiles \eqgreater 0) \and (num_tiles \eqless capacity);
}
```

7. <> denotes inequality.
8. *\in* and *\not_in* denote set membership, while # denotes set cardinality.
9. ^ denotes exponentiation.

A component's interface consists of a set of interface elements. An interface element is declared with a direction indicator (*prov* or *req*), name, set of parameters, and possibly a result type. The parameter specification syntax is identical to that used in variable specification. Since interface elements may have identical names, a unique label may be assigned to each as a notational convenience. For example, in

```
prov gt1: GetTile (location : Integer) : Color;
prov gt2: GetTile (i : Natural) : GSColor;
```

both interface elements are intended to be used with operations that remove and return a tile at the given location in the KLAX *well*. The first interface element accesses a color tile at the *Integer* location *location*; the second accesses a gray-scale tile at the *Natural* location *i*. The labels, *at1* and *at2*, uniquely identify the two.

A component's behavior consists of a set of operations. Each operation is declared as either *prov*ided or *req*uired and with a unique label, used to refer to the operation. Additionally, each operation may define a set of preconditions that must be true *prior* to the operation's execution, and a set of postconditions that must be true *after* its execution. Since operations are separated from the interface elements through which they are accessed, operations also define local variables, which, along with component state variables, are used in specifying the pre- and postcondition predicates. The pre- and postconditions are specified in the same manner as component invariants. An operation's postcondition may contain the keyword *\result*, to denote the operation's return value. Additionally, a postcondition may specify the value of a variable after the operation has executed, denoted with a ~, followed by the variable name.

An example operation can be specified as follows.

```
prov tileget: {
   let pos : Integer;
   pre (pos \greater 0) \and (pos \eqless num_tiles;)
   post \result = well_at(pos) \and ~num_tiles = num_tiles - 1;
}
```

The local variable *pos* denotes the position in the well. *num_tiles* and *well_at* are component state variables. Recall that *well_at* is a function that returns the color value of the well at the given position. The postcondition specifies that the number of tiles in the well decreases after the tile is removed.

The *tileget* operation can export multiple interfaces. For example, both *GetTile* interface elements can be mapped to the operation based on the mapping rules specified in Section 4.1, provided that *GSColor* is a basic subtype of *Color*:

```
map {
   gt1 -> tileget (location -> pos);
   gt2 -> tileget (i -> pos);
}
```

These elements are composed into a complete component specification in the following manner:[10]

10. For illustration, the specification of *WellADT* only includes the aspects of this component previously discussed.

```
component WellADT is subtype Matrix (beh) {
   state {
      capacity : Integer;
      num_tiles : Integer;
      well_at : Integer -> GSColor;
   }
   invariant {
      (num_tiles \eqgreater 0) \and (num_tiles \eqless capacity);
   }
   interface {
      prov gt1: GetTile (location : Integer) : Color;
      prov gt2: GetTile (i : Natural) : GSColor;
   }
   operations {
      prov tileget: {
         let pos : Integer;
         pre (pos \greater 0) \and (pos \eqless num_tiles);
         post \result = well_at(pos) \and ~num_tiles = num_tiles - 1;
      }
   }
   map {
      gt1 -> tileget (location -> pos);
      gt2 -> tileget (i -> pos);
   }
}
```

Finally, a component type may be specified as a *virtual* type: it can be used in the definition of the topology, but it does not have a specification and does not affect type checking of the architecture; furthermore, a virtual type cannot be evolved via subtyping. The concept of virtual types is useful in the case of components for which implementations are known to already exist, but which are not specified in C2SADEL.

4.4.2 Connector Types

Since the connectors in this dissertation do not export a particular interface, but are context-reflective, the only aspect of connector types modeled in C2SADEL is their filtering mechanism, denoted with the *message_filter* keyword. The different filtering mechanisms are *no_filtering*, *notification_filtering*, *message_filtering*, *prioritized*, or *message_sink*. We consider type of interaction and concurrency, discussed in Section , to be implementation-level issues, and do not model them as part of a connector. An example broadcast connector is specified as follows.

```
connector BroadcastConn is {
   message_filter no_filtering;
}
```

4.4.3 Topology

To model the topology of an architecture, component and connector types are instantiated and interconnected. Each type may be instantiated multiple times. C2SADEL requires that a component be attached to at most one connector on its top and one on its bottom; it allows multiple components and connectors to be attached to the top and bottom sides of a connector. The part of the KLAX topology that concerns the well (see Figure 2-5 on page 13) is specified as follows.

```
architectural_topology {
   component_instances {
      Well : WellADT;
      WellArt : WellArtist;
      MatchLogic : TileMatchLogic;
   }
   connector_instances {
      ADTConn : BroadcastConn;
      ArtConn : BroadcastConn;
   }
   connections {
      connector ADTConn {
         top Well;
         bottom MatchLogic, ArtConn;
      }
      connector ArtConn {
         top ADTConn;
         bottom WellArt;
      }
   }
}
```

4.4.4 Composite Components

C2SADEL supports *hierarchical composition* of components, where an entire architecture is used as a single component in another architecture, as shown in Figure 4-17. We supply no additional constructs in the language to support hierarchical composition. Instead, we use the existing C2SADEL constructs to specify composite components (as architectures) and provide a technique for determining their sets of provided and required services, needed for architectural type checking.

Direct interaction with a composite component is restricted to its externally-visible constituent components. A component in an architecture is externally visible if it is a top-most or bottom-most component in the architecture (e.g., C1 and C7, respectively, in Figure 4-17), or if it is accessible via connector-to-connector links (e.g., C3 in Figure 4-17).[11]

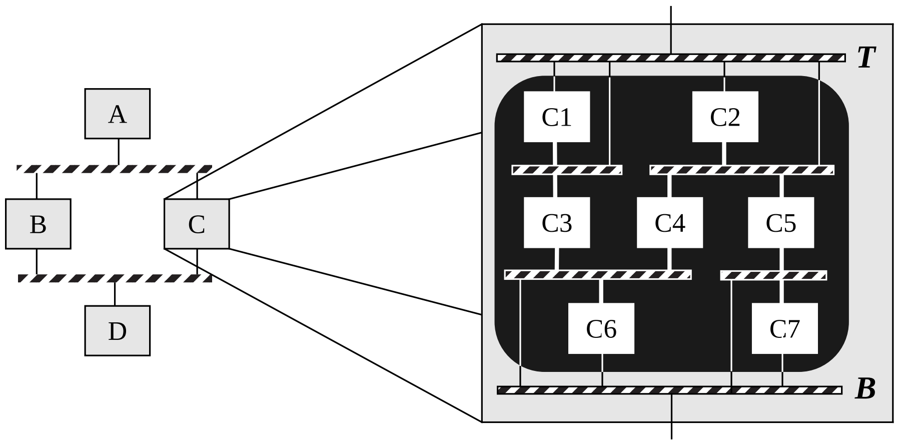


Figure 4-17. Hierarchical composition.
A component, *C*, in an architecture is itself an architecture (highlighted).

We employ the following technique for achieving hierarchical composition; the technique preserves the principles of the C2 architectural style:

- add connectors, *T* and *B*, at the top and bottom of the composite component;
- attach the top sides of the architecture's top-most components and connectors to the bottom of *T*;
- attach the bottom sides of the architecture's bottom-most components and connectors to the top of *B*;
- finally, attach the top of T to the bottom of the connector above it and the bottom of B to the top of the connector below it.

The invariant and sets of provided and required services of the composite component are a function of its constituent components' invariants and services. Recall that C2 mandates that a component only request services from components above it and respond to requests from components below it. Thus the composite component's provided services are a union of the bottom-visible components' provided services. Similarly, the composite component's required services are a union of the top-visible components' required services. Finally, although the type theory defined in Section 4.1 currently does not support the evolution of composite components, we do employ a mechanism for determining a composite component's invariant: since only the externally-visible components directly interact with the rest of the architecture, only their invariants are relevant for subtyping; thus, a composite component's invariant is a conjunction of the invariants of its externally-visible components.[12]

11. For simplicity, in the remainder of the section we refer to components externally visible from a composite component's top and bottom sides as "top-visible" and "bottom-visible," respectively.
12. We discuss our intent to expand component evolution to include composite components in Chapter 7.

CHAPTER 5: Mapping Architecture to Implementation

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

At the same time, it is unreasonable to expect that architectures can be used as a basis for a *general* solution to the problem of automated system generation. That would essentially reduce architecture-based software development to a variant of transformational programming [67], thus inheriting all of the latter's problems and limitations, with the added problem of scale. Instead, architecture can render the problem more tractable by focusing on application properties within specific domains and OTS reuse of components and connectors. Understanding the properties of an application domain (or set of domains) enables the identification of canonical implementation constructs, which can be exploited during implementation generation. Reuse of large-grain software components offers the potential for significant savings in application development cost and time. Reuse of interconnection mechanisms, typically found in OTS middleware technologies, enables architects to achieve the desired properties during system composition: heterogeneity, efficiency, reliability, distribution, and so forth.

Successful reuse and substitutability of components and connectors depends both on qualities of the components and connectors reused as well as the software context in which the reuse is attempted. Architectural styles are disciplined approaches to the structure and design of software applications and offer the potential of providing a hospitable setting for such reuse. However, all styles are not equally well equipped to support reuse. If a style is too restrictive, it will exclude the world of legacy components. On the other hand, if the set of style rules is too permissive, developers may be faced with all of the well documented problems of reuse in general [7], [8], [24], [38], [81]. Therefore, achieving a balance, where the rules are strong enough to make reuse tractable but broad enough to enable integration of OTS components, is a key issue in formulating and adopting architectural styles.

In this chapter we argue that the principles introduced in this dissertation and embodied in the C2 architectural style offer significant potential for automating the generation of implementations from architectures, OTS reuse and, consequently, the development of application families. By providing support to encapsulate legacy systems and new software in C2 components and bind these components together with custom-built or middleware-integrated software connectors, the costs and difficulties of building new software systems are diminished. By embodying in software tools the principles discussed in the preceding chapter, this dissertation enables the evolution of architectures to be reflected in their implementations. Finally, by enabling evolution and multiple implementations of an architecture, the dissertation directly facilitates application families.

This chapter discusses the development tools we have constructed to aid the mapping of C2 architectures to their implementation(s) and reuse of OTS components. The chapter also discusses

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

Figure 5-1. C2 implementation framework.

the issues in reusing OTS components and middleware technologies, highlighting those characteristics of our approach and tools that facilitate their reuse. Finally, we discuss an environment for modeling, analyzing, evolving, and implementing architectures. The environment supports automated implementation generation aided by OTS reuse.

5.1 C2 Implementation Infrastructure

5.1.1 C2 Class Framework

To support implementation of C2 architectures, we developed an extensible framework of abstract classes for C2 concepts such as architectures, components, connectors, communication ports, and messages, shown in Figure 5-1. This framework is the basis of development and OTS component reuse in C2. It implements component interconnection and message passing protocols. Components and connectors used in C2 applications are subclassed from the appropriate abstract classes in the framework.[1] The components subclassed from the framework's "component" abstract classes contain application-specific functionality. The connectors are application-independent, i.e., they are reusable across C2 architectures, and can differ in their interaction mechanisms, support for concurrency, and degree of message filtering, as discussed in Chapter 4. The connectors are implemented to export dynamically changing (i.e., context-reflective) interfaces.

The framework guarantees interoperability among components and connectors, eliminates many repetitive programming tasks, and allows developers of C2-style applications to focus on application-level issues. The different implementations of component and connector classes in the framework enable a variety of implementation configurations for a given architecture: the entire resulting system may execute in a single thread of control, a set of components and/or connectors

1. Note that implementation framework concepts, such as "abstract class," refer to the programming language used to implement an architecture, and are in no way related to the architectural type theory presented in Chapter 4.

may run in its own thread of control or OS process, or each component and connector may run in its own thread of control or process.

The framework has been fully implemented in C++ and Java; its subset is also available in Ada. The framework is compact, as reflected in the fact that both of its full implementations consists of approximately 3500 commented source lines of code. The framework is also easily extensible. For example, we have been able to add a multi-process and multi-lingual connector to the framework by reusing the Q system [44]. This connectors enables communication between C2 components implemented in C++ and Ada. The infrastructure for extending this support to Java C2 components has been integrated into the framework in the form of the ILU distributed object system [97]. The integrations of Q and ILU are discussed in more detail in Chapter 6.

The Java implementation of the framework is particularly significant in that it represents the first step in our endeavor to (partially) automate domain translation. A common form of interface mismatch between communicating components is different ordering of message parameters, as discussed in Chapter 2. The Java implementation of C2 messages eliminates this problem by allowing components to access message parameters by name, rather than by position. Other domain translation techniques can be incrementally incorporated into the framework. The Java implementation of the framework has also been used as the basis of ArchShell, a tool that supports interactive construction, execution, and runtime modification of C2-style architectures [61].

5.1.2 C2 Graphics Binding

As already discussed, C2's particular focus is on applications with a significant GUI aspect. However, commercial user interface toolkits have interface conventions that do not match up with C2's notifications and requests. In a C2 architecture, the toolkit is always at or near the bottom, since it performs functions conceptually closest to the user (see Chapter 2). As such, it must be able to receive notifications from components above it and issue requests in response. Typically, however, toolkits will generate events of the form "this window has been selected" or "the user has typed the 'x' key." These events need to be converted into C2 requests before they can be sent up the architectures. Conversely, notifications from a C2 architecture have to be converted to the type of invocations a toolkit expects.

In order for these translations to occur and be meaningful, careful thought has to go into the design of the bindings to the toolkits such that they contain the required functionality and are reusable across architectures and applications. Our experience with reusing OTS components in C2 architectures and with building bindings for Motif and OpenLook in Chiron-1 [90] suggested the approach depicted in Figure 5-2: a C2 component is created such that the graphics toolkit becomes its internal object, while the C2 message traffic is handled by its dialog. *GraphicsBinding*'s dialog accepts notifications from C2 components above it and reifies them as calls to toolkit methods. It also transforms user events, generated in the graphics toolkit, into C2 requests. A C2 component's internal architecture, its reliance on asynchronous, message-based communication, and no assumption of shared address space or thread of control eliminate the need to internally modify the toolkits in any way.

We have built C2 bindings for two graphics toolkits: Xlib [77] and Java's AWT [14]. The Xlib and AWT bindings are subclasses of C++ and Java frameworks' *component* classes, respectively. This enables their easy integration into C2-style architectures built using the two frameworks. The

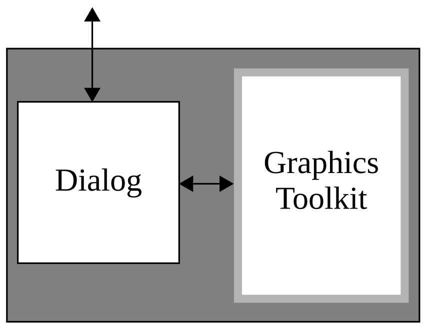


Figure 5-2. A C2 *GraphicsBinding* component.
An OTS windowing toolkit is wrapped inside a C2 component. Since a graphics binding is always at the very bottom of a C2-style architecture, the Dialog only receives notifications and emits requests.

dialog portions of the toolkit binding components are extensible wrappers that enable rendering of user interface widgets and graphical objects on the screen by exchanging C2 messages. The components export standard interfaces and are reusable across architectures implemented using each respective framework. The standardized interfaces also enable the interchange of the two bindings in the same architecture if an appropriate connector that supports composition of multi-lingual components is employed.[2]

5.2 OTS Component Reuse

The two C2 *GraphicsBinding* components represent examples of OTS component reuse. This section discusses the characteristics of our approach, reflected in the C2 style, that make it a good platform for OTS reuse. We then propose a set of heuristics that should be employed when attempting to reuse OTS components in C2-style architectures.

5.2.1 C2's Suitability for OTS Component Reuse

Several characteristics of the C2 style render it well suited to supporting reuse. Although most of these characteristics are not unique to C2, this dissertation's approach of combining and exploiting them for reuse is. We believe the style rules are restrictive enough to make reuse easier while flexible enough to integrate components built outside the style:

- *component heterogeneity* - the style does not place restrictions on the implementation language or granularity of the components.
- *substrate independence* - a component is not aware of components below it, and therefore does not depend on their existence.
- *internal component architecture* - the internal architecture of a C2 component (recall Figure 2-2 on page 6) separates communication from processing. The dialog receives all incoming notifications and requests and maps them to internal object operations (*implicit invocation*). By localizing this mapping, the dialog isolates the internal object (i.e., the OTS component) from changes in the rest of the architecture.
- *asynchronous message passing via connectors* - since all communication between components is achieved by exchanging asynchronous messages through connectors, control integration

2. Note that identical interfaces are not a requirement; two bindings with different interfaces could be substituted for one another by using a domain translator.

issues are greatly simplified. This remedies some of the problems associated with integrating components which assume that they are the application's main thread of control [24].

- *no assumption of shared address space* - components cannot assume that they will execute in the same address space as other components. This eliminates complex dependencies, such as components sharing global variables, that hamper reuse.
- *no assumption of single thread of control* - conceptually, components execute in their own thread(s) of control. This allows multithreaded OTS components, with potentially different threading models, to be integrated into a single application.
- *separation of architecture from implementation* - many potential performance issues can be remedied by separating the conceptual architecture from actual implementation techniques. For example, C2 disallows direct procedure calls and any assumptions of shared threads of control or address spaces in a conceptual architecture. However, substantial performance gains may be made in a particular implementation of that architecture by placing multiple components in a single process and address space where appropriate. Such implementation decisions are isolated in the C2 implementation framework, discussed above. For example, if two components are placed in the same address space, a connector between them may use direct procedure calls to implement message passing.[3]

5.2.2 OTS Component Reuse Heuristics

A study of the properties of the C2 style and the experience of integrating OTS components, such as graphics toolkits, have enabled us to devise a set of simple heuristics for OTS component integration in C2. We have refined these heuristics and established their utility and applicability in the process of incorporating OTS components, discussed in Chapter 6. The only assumption we make is that the functionality of OTS components will be accessible (at least) via application programmable interfaces (APIs):

- If the OTS component does not contain all of the needed functionality, its source code must be altered. In general, this is a difficult task, whose complexity is well recognized [24], [38], [56].[4]
- If the OTS component does not communicate via messages, a C2 wrapper must be built for it. This was the case with the two *GraphicsBinding* components described above.
- If the OTS component is implemented in a programming language different from that of other components in the architecture, an IPC connector must be employed to enable their communication. An example such connector that uses the Q system [44] and enables the interaction of C2 components implemented in C++ and Ada was briefly discussed above. We have also incorporated existing support for software packaging [72]. The potential benefit of software packaging technologies is to decouple a component's functionality from its interfacing requirements and automate a significant portion of the work associated with adapting the component for use in new environments. The details of this integration are discussed in Chapter 6.
- If the OTS component must execute in its own thread of control, an inter-thread connector must be employed. This was accomplished in the case of the Java AWT graphics toolkit.

3. We do not currently provide procedure call connectors as part of the framework. This was a deliberate decision intended to explore the issues in employing less conventional types of interaction, in particular, implicit invocation via asynchronous messages.
4. Note that the component can still be reused "as is" if the developers are willing to risk degraded or incorrect performance, due to partial communication and partial component service utilization in the architecture.

- If the OTS component executes in its own process, an IPC connector must be employed. The issues in incorporating IPC connectors into C2 were discussed in Chapter 4.
- If the OTS component communicates via messages, but its interface does not match interfaces of components with which it is to communicate, a domain translator must be built for it. Although not a specific focus of our work to date, preliminary support for domain translation exists in the Java implementation of the C2 class framework. The architectural type theory, presented in Chapter 4, has the potential to further aid this task.

The information above is summarized in Table 5-1. As a whole, the heuristics in the table are conceptually very simple. This simplicity is an indicator of the inherent support for OTS component reuse provided by this dissertation. Practical support for the heuristics requires specific, but achievable, techniques, such as component wrapping and domain translation, and technologies, such as inter-thread or IPC connectors.

Table 5-1: OTS Component Integration Heuristics for C2

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

5.3 Constructing Connectors by Reusing OTS Middleware

As argued in Chapter 4 and indicated in Table 5-1, connectors are a natural abstraction that isolates many issues regarding interactions among components in an architecture. Connectors have an externally-visible structure, while internally they utilize low-level communication mechanisms (e.g., message exchange between ports in a single process or RPC for inter-process communication). The exact nature and source of these low-level mechanisms is irrelevant, so long as the connector maintains the same external appearance. Middleware (e.g., Enterprise JavaBeans [63], CORBA [62], COM/DCOM [80]) is a potentially useful tool when building software connectors in that it can supply these communication mechanisms in a reliable manner. First, it often specializes in bridging across thread, process and network boundaries. Secondly, it can provide pre-built protocols for exchanging data among software components or connectors. Finally, some middleware packages already implement features of software connectors such as filtering, routing, and broadcast of messages or other data.

The techniques for distributing connectors across process boundaries, discussed in Chapter 4 and depicted in Figure 4-16, are used as a basis of OTS connector reuse in this dissertation: a C2 connector is “sliced” horizontally or vertically across processes and an OTS middleware technology is used to ensure the proper communication among the connector segments. By combining the two techniques, a single conceptual connector can potentially be implemented using multiple middleware mechanisms.

5.3.1 Middleware Evaluation Criteria

Unlike software components, which are reused to satisfy the needs of specific applications, connectors are intended to be used across applications. Still, the choice of a specific connector will depend upon the characteristics and needs of an application. For this reason, we provide a framework for evaluating OTS middleware before investing in their integration with our infrastructure. In doing so, we focus on several factors:

- *inter- and intra-process communication support* — a distributed application is likely to contain a mix of components that execute in a *single thread* of control, in *different threads* of control (but in the same process), and in *different processes*, some of which will reside on different machines. If a given middleware technology effectively supports only interprocess communication, its utility is limited and additional types of middleware may need to be employed. Note that multiple types of middleware in an application may indeed be preferable, as each may optimize a particular type of communication.
- *features of software connectors* — a middleware technology may only provide the ability for two processes to exchange data. The needs of software connectors are broader: event routing (e.g., broadcast, multicast, point-to-point), filtering, registration, and so forth [69]. If such features are not supported, additional infrastructure must be provided before such a technology may be used in a distributed architecture, such as a typical C2-style architecture.
- *platform and language support* — software architectures, and C2 architectures in particular, are intended to support the development of distributed systems, built out of components which are potentially implemented in different programming languages and executing on multiple platforms. An interconnection technology that supports multi-lingual and multi-platform applications is thus a better candidate for integration than one that does not. The penalties (e.g., adoption costs, performance) accrued by using a technology that only supports a single language and/or platform may outweigh any benefits of using it.
- *communication method* — similarly to the different types of connectors at the architectural level (see Section 4.3), methods of communication across middleware technologies vary and can include RPC, message passing, passing object references, shared memory, and so forth. A middleware technology that is not suited to an architectural style may cause implementation difficulties when used in the context of that style. The degree of difficulty will vary depending on the middleware and the style. For example, we have been able to implement connectors that translate from the RPC communication method to message-passing to fit the constraints of the C2 style with relatively little effort.
- *ease of integration and use* — if integrating an OTS technology into the implementation infrastructure and/or its use in an application requires a substantial amount of effort, its effectiveness and power may be rendered irrelevant. One possible source of problems was discussed above (communication method mismatch). Another is a mismatch in assumptions made by the OTS tool and the environment into which it is being incorporated. For example, if an interconnection tool assumes that it is the application's main thread of control, it is not well suited for use with C2, since C2 mandates that all components execute independently of each other.
- *multiple instances in an application* — one benefit of distributed systems is that they do not have to depend on a single set of resources, thus avoiding performance bottlenecks. Analogously, it may be useful to physically distribute the very tool used to interconnect a

distributed system. Centralized OTS middleware tools that use a single point of communication form potential bottlenecks and single points of failure.

- *support for dynamic change* — for an important class of safety- and mission-critical software systems, such as air traffic control or telephone switching systems, shutting down and restarting the system for upgrades incurs unacceptable delays, increased cost, and risk. Support for run-time modification is thus a key aspect of these systems. A middleware technology that does not support dynamic change is not an adequate candidate for them.
- *performance* — performance is a key issue in systems with real-time requirements. For example, in the KLAX application described in Chapter 2, several hundred messages may be generated every second. The ability to efficiently ferry these messages among the components and across process boundaries is paramount.

Because software connectors provide a uniform interface to other connectors and components within an architecture, architects need not be concerned with the properties of different middleware technologies as long as the technology can be encapsulated within a software connector. Internally, however, connectors based on different middleware technologies have different abilities. Implementors of a given architecture can use this knowledge to determine which middleware solutions are appropriate in a given implementation of an architecture. In this way, encapsulating middleware functionality within software connectors maintains the integrity of an architectural style by keeping it separate from implementation-dependent factors such as how to bridge process boundaries within a given architecture.

5.3.2 Multiple Mappings from Architecture to Implementation

Different connector implementations enable architects to select a mapping from an architecture to its implementation that is best suited to the system's current requirements. Different middleware technologies used in implementing a connector can have unique benefits. By combining multiple such technologies in a single application, the application can potentially obtain the benefits of all of them. For instance, a middleware technology that supports multiple platforms but only a single language could be combined with one that supports multiple languages but a single platform, to create an application that supports both multiple languages and multiple platforms.

The advantages of combining multiple middleware technologies within a single software connector are manifold. In the absence of a single panacea solution which supports all required platforms, languages, and network protocols, the ability to leverage the capabilities of several different middleware technologies significantly widens the range of applications that can be implemented within an architectural style such as C2.

As an example, we consider two connectors that encapsulate different OTS middleware technologies. The two are implemented using different strategies for supporting component interaction across process boundaries, described in Chapter 4. The two connectors can be used to implement a single conceptual connector, as shown in Figure 5-3. This is accomplished with no modification to the C2 implementation framework or the connectors themselves, by combining the lateral welding technique from Figure 4-16a with the horizontal slicing technique from Figure 4-16b. This approach creates a three-process "virtual connector" using two in-process C2 connectors to laterally bind two multi-process connectors. The approach works for any

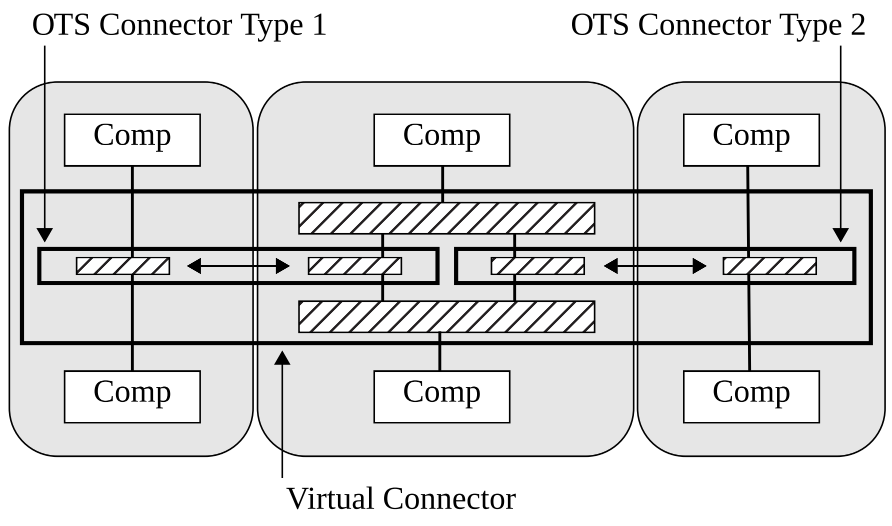


Figure 5-3. A three-process C2 application that employs different OTS middleware mechanisms. A single virtual connector is implemented with two in-process and two multi-process connectors. The in-process connectors facilitate message passing between the multi-process connectors. Shaded ovals represent process boundaries.

combination of OTS connectors that use the lateral welding technique. An integration of OTS middleware technologies using the technique depicted in Figure 5-3 is discussed in Section 6.2.

An alternative approach would be to explicitly create a single connector implementation that supports both OTS middleware technologies, but this would require changes to the framework. The technique shown in Figure 5-3 avoids this difficulty with an expected slight efficiency cost due to the addition of in-process connectors to bind the multi-process connectors.

5.4 An Environment for Architecture-Based Development and Evolution

Chapter 4 discussed techniques for evolving architectures and individual architectural elements, as well as an ADL for modeling them. This chapter has thus far presented a set of mechanisms for providing implementations for architectures and their individual elements. Many of the concepts from Chapter 4, such as minimal component interdependencies and heterogeneous, inherently-evolvable connectors, are directly supported in the mechanisms introduced in this chapter, such as the implementation framework and middleware-integrated connectors. Other concepts, e.g., subtyping and analysis via type checking, are not supported by the mechanisms described thus far and can only be used as conceptual tools in making decisions about evolving existing architectures and reusing OTS components and connectors. The evolution and reuse must be performed manually, which is potentially expensive and error-prone. Furthermore, the implementation framework alone cannot guarantee that an architecture described in C2SADEL will be mapped to its implementation in the intended manner. Therefore, to render our architecture-based development and evolution methodology more useful in practice, we provide an environment that implements subtyping, type checking, and transferring of the properties of an architecture to its implementation. This section describes the environment, called DRADEL (*D*evelopment of *R*obust *A*rchitectures using a *D*escription and *E*volution *L*anguage).

DRADEL is a culmination of several years of our research in software architectures and ADLs, evolution, and OTS reuse. It allows

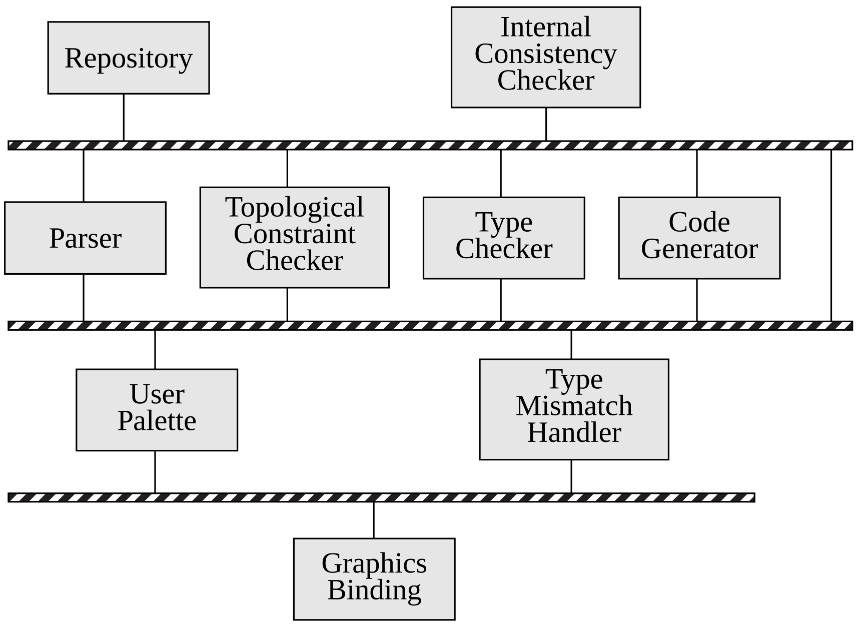


Figure 5-4. Architecture of the DRADEL environment.
The architecture fully adheres to the rules of the C2 style.

- architecture modeling in C2SADEL,
- evolution via heterogeneous subtyping,
- analysis of internal architectural consistency, topological constraints, and type conformance among interacting component instances in a given configuration, and
- generation of application skeletons using our implementation infrastructure and, thereby, support for OTS component and connector reuse.

DRADEL also guides a high-level “architecting” process and supports the concept of *architect’s discretion*: architecture-level analyses serve as guides to the architect; the architect has the prerogative to override an analysis tool if (s)he believes the errors reported by the tool not to be critical and/or their correction to be unreasonably expensive.

In addition to supporting the concepts introduced in this dissertation, DRADEL was also constructed to serve as a demonstration of those concepts. The conceptual architecture of the DRADEL environment is shown in Figure 5-4. Just like the application architectures it is built to support, the architecture of DRADEL itself adheres to C2 style rules. The environment is built using the C2 implementation infrastructure, discussed in Section 5.1. Specifically, we have used the Java version of the C2 framework and the binding to Java’s AWT toolkit in the implementation of DRADEL. The current implementation consists of 13,000 source lines of code, in addition to the 7,000 lines of code in the base framework and C2 *GraphicsBinding*. In the current implementation, each DRADEL component executes in a separate thread of control within the same process.

The remainder of this section presents a more detailed view of DRADEL’s functionality.

5.4.1 DRADEL's Architecture

5.4.1.1 Repository

The *Repository* component from Figure 5-4 stores architectures modeled in C2SADEL. Upon request, the component broadcasts architectural descriptions via notifications. The *Repository* is currently ASCII-file based, resulting in its simplicity and cross-platform portability. This implementation of the *Repository* also allows architects to edit C2SADEL descriptions using standard text editors. On the other hand, the flat-file organization of the *Repository* is not well suited to maintaining “design palettes”, i.e., hierarchies of reusable architectural elements. For this reason, another underlying implementation, such as a relational database, may be preferable. Two implementations of the *Repository* component will be fully interchangeable in DRADEL, so long as they export identical interfaces.

5.4.1.2 Parser

The *Parser* component receives via C2 messages specifications of architectures or sets of components described in C2SADEL and parses each specification. If the specification is syntactically correct, the *Parser* requests that the *InternalConsistency-Checker* component check the consistency of the specification; otherwise, the *Parser* notifies the *UserPalette* component of the syntax errors and aborts parsing. The architect must correct the errors and again invoke the *Parser.*

5.4.1.3 InternalConsistencyChecker

The *InternalConsistencyChecker* builds its own representation of the architecture and ensures that components and connectors are properly specified (e.g., two interface elements in a component cannot be identical), instantiated, and connected; that component interface elements are correctly mapped to operations (as specified in Section 4.1); that variables referenced in an operation are defined either as local variables for that operation or as component state variables; and that operation pre- and postcondition expressions are type correct (e.g., a `set` variable is never used in an arithmetic expression, although its cardinality may be). Furthermore, the *InternalConsistencyChecker* computes communication links for every component in an architecture: two components can interoperate if and only if they are on the opposite sides of the same connector (e.g., *Repository* and *Parser* in Figure 5-4) or are on the opposite sides of two connectors which are, in turn, connected by one or more connector-to-connector links (e.g., *Repository* and *UserPalette* in Figure 5-4).

Once the entire specification is parsed and its consistency ensured, its internal representation is broadcast by the *InternalConsistencyChecker* to the *TopologicalConstraintChecker*, *TypeChecker*, and *CodeGenerator* components.

5.4.1.4 TopologicalConstraintChecker

The *TopologicalConstraintChecker* component receives a notification from the *InternalConsistencyChecker* containing either (a representation of) an architecture or a set of components. If the notification contains an architecture, the *TopologicalConstraint-Checker* ensures that the topological rules of the C2 style are satisfied (see Chapter 2); if any topological constraints are violated, the component notifies the *UserPalette* of the errors. If the notification received from the *InternalConsistencyChecker* contains a set of components, no topological constraints are enforced.

5.4.1.5 TypeChecker

The *TypeChecker* performs two kinds of analysis:

- given an architectural description, it analyzes each component instance to establish whether its requirements are satisfied by the component instances along its communication links;
- given a set of component specifications, the *TypeChecker* ensures that their specified subtyping relationships hold.

The *TypeChecker* performs these functions by establishing the relationships, discussed in Section 4.1, among component interface elements, invariants, and operations. It attempts to find the appropriate matches among component state variables, operation variables, and interface element parameters, as well as operation and interface element result types, such that the required (implication) relationships hold. As required by the architectural type theory, in order to establish the conformance of a required operation to a provided operation, the *TypeChecker* instantiates variables of type STATE_VARIABLE in required operations with provided operations' variables. This is done repeatedly, until either a match is found that satisfies the conformance rules or no further matches are possible.

Since our approach does not explicitly model *basic types* (see Section 4.1), the *TypeChecker* essentially performs symbolic evaluation of logical expressions. For this reason, there are two categories of cases in which the *TypeChecker* is unable to correctly determine whether the implication indeed holds. The first category involves basic types that are conceptually related (e.g., *Integer* and *Natural*), but have not been declared as such in C2SADEL. The *TypeChecker* treats such types as unrelated and does not attempt to establish a relationship between variables of those types during type checking.

The other category of cases occurs in expressions involving a single basic type. For example, assume that one component's operation precondition is

PRE1: `n \eqless 10`

and a candidate *beh*avior subtype's corresponding operation precondition is

PRE2: `n^2 \eqless 100`

where `n` in both expressions is declared to be of the basic type *Natural*. To establish that the *beh*avior subtyping relationship holds, PRE1 must imply PRE2. This is obviously true: if a natural number is less than or equal to `10`, its square will be less than or equal to `100`. Note that this would not be true of integers (e.g., if `n = -20`). The *TypeChecker* has no knowledge of the non-negative property of natural numbers and cannot establish that the relationship between the two preconditions is true. It is *pessimistically inaccurate*: it treats those cases for which it cannot determine type conformance as errors.

Finally, there is another category of expressions that the current implementation of the *TypeChecker* evaluates in a pessimistically inaccurate manner. However, unlike the previous two categories, which are a direct result of our decision not to model the properties of basic types, the *TypeChecker* can be evolved to support this class of expressions. These expressions involve logical operators. For example, if one component's operation precondition is

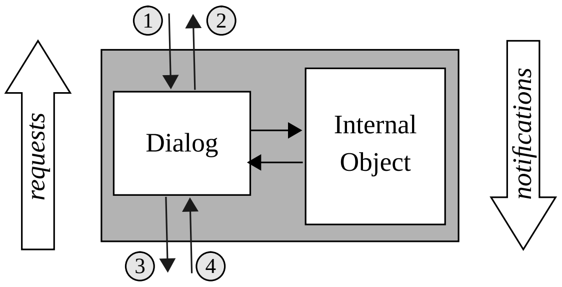


Figure 5-5. Internal architecture of a canonical C2 component.
This figure is a slightly modified version of Figure 2-2. Message pathways are shown from the component's perspective and are explicitly labeled: (1) incoming notifications; (2) outgoing requests; (3) outgoing notifications; and (4) incoming requests.

```
PRE1: a \and c
```

and a candidate *beh*avior subtype's corresponding operation precondition is

```
PRE2: b \implies c
```

the *beh*avior subtyping relationship will hold if the following implication is satisfied:

```
(a \and c) \implies (b \implies c)
```

Given the definitions of logical conjunction and implication, the above expression is true for all values of variables `a`, `b`, and `c`. However, truth tables for logical operators have not yet been implemented in the *TypeChecker*. The current version of the *TypeChecker* therefore cannot determine the truth value of this expression; instead, it informs the architect of the possible error and leaves the final decision up to the architect.

5.4.1.6 CodeGenerator

The *CodeGenerator* component generates application skeletons for the specified architecture or set of components. Like DRADEL itself, the application skeletons are built on top of the Java C2 implementation framework. The "main program," containing the configuration (component and connector instances and their interconnections), is automatically generated, as is the "make" file for all of the generated files. For each specified component, the *CodeGenerator* creates the corresponding C2 component with the canonical internal architecture, shown in Figure 5-5. The *InternalObject* of every generated component is a Java class corresponding to the C2SADEL specification of the component. For example, the generated internal object for the *WellADT* component from Section 4.4 is shown in Figure 5-6: the state variables, state variable access methods, and provided component service declarations are generated.

Each method corresponding to a component service (e.g., *GetTile* in Figure 5-6) is implemented as a null method in the generated class[5], and is preceded by a comment containing the method's precondition and followed by one containing its postcondition. In general, these individual, application-specific methods are the only parts of a component for which the developers will have to provide an implementation. In the case of entirely new functionality, the pre- and postcondition comments serve as an implementation guideline to the developer;

5. The exception are non-void functions: Java requires their results to be initialized and returned. DRADEL initializes the results to an arbitrary value.

```
package c2.KLAXSystem;
import java.lang.*;

public class WellADT extends Object {

   // COMPONENT INVARIANT: num_tiles \eqgreater 0.0 \and num_tiles \eqless capacity

/***** State Variables *****/
   private Integer num_tiles;
   private Integer capacity;
   private GSColor well_at_pos;

/***** Class Constructor *****/
   public WellADT() {
      num_tiles = null;      // or: new Integer(<init val>);
      capacity = null;       // or: new Integer(<init val>);
      well_at_pos = null;    // or: new GSColor(<init val>);
   }

/***** ADL Specified Methods *****/
   // PRECONDITION: pos \greater 0.0 \and pos \eqless num_tiles
   public GSColor GetTile(Natural pos) {
      /*** METHOD BODY ***/
      return well_at_pos;
   }
   // POSTCONDITION: \result = well_at_pos \and ~num_tiles = num_tiles - 1.0

   // PRECONDITION: pos \greater 0.0 \and pos \eqless num_tiles
   public Color GetTile(Integer pos) {
      /*** METHOD BODY ***/
      return well_at_pos;
   }
   // POSTCONDITION: \result = well_at_pos \and ~num_tiles = num_tiles - 1.0

/***** State Variable Access Methods *****/
   public void SET_num_tiles(Integer new_value) {
      num_tiles = new_value;
   }

   public Integer GET_num_tiles() {
      return num_tiles;
   }

   public void SET_capacity(Integer new_value) {
      capacity = new_value;
   }

   public Integer GET_capacity() {
      return capacity;
   }

   public void SET_well_at_pos(GSColor new_value) {
      well_at_pos = new_value;
   }

   public GSColor GET_well_at_pos() {
      return well_at_pos;
   }
}
```

Figure 5-6. Generated internal object class skeleton for the *WellADT* component. Vertical spacing of the generated code has been altered to fit on the page.

otherwise, they serve as an indicator of whether OTS functionality may be reused in the given context, by replacing internal object skeletons with OTS components implementing the specified methods.

Except for the Java defined types (e.g., *Integer* or *String*), the *CodeGenerator* also supplies class skeletons for all other basic types, which include constructors and access methods. The actual data structures must be specified in these classes by the developers. C2SADEL `set` types are currently implemented by subclassing from Java's *Vector* class, which provides a reasonable abstraction of a set.

The *Dialog* portion of a C2 component from Figure 5-5 is responsible for all of the component's message-based interaction. The *CodeGenerator* can generate a component's dialog almost completely from its C2SADEL specification: the provided services correspond to the notifications a component emits (message pathway `3` in Figure 5-5) and the requests to which it is capable of responding (pathway `4`), while the required services are used as a basis for specifying the requests the component issues (pathway `2`). The parameters of a notification generated by a component are determined from the specification of the corresponding operation's postcondition: any modified variables (marked with a ~) and the operation's result (in any) are reported; we are currently assuming that it is not necessary to report variables whose values remain unchanged.[6] The dialog class also contains a specification of the component's message interface in the form needed by the underlying implementation framework to support various protocols of communication, e.g., message broadcast, registration, or point-to-point.

The only portion of the dialog that cannot be generated based on the information currently modeled in a C2SADEL specification is what the dialog should do in response to the notifications it receives (message pathway `1` in Figure 5-5). This information could easily be specified in the ADL, as was shown in the prototype design language for C2-style architecture that preceded C2SADEL [53]; however, we have chosen to remove those constructs in the interest of language simplicity.[7]

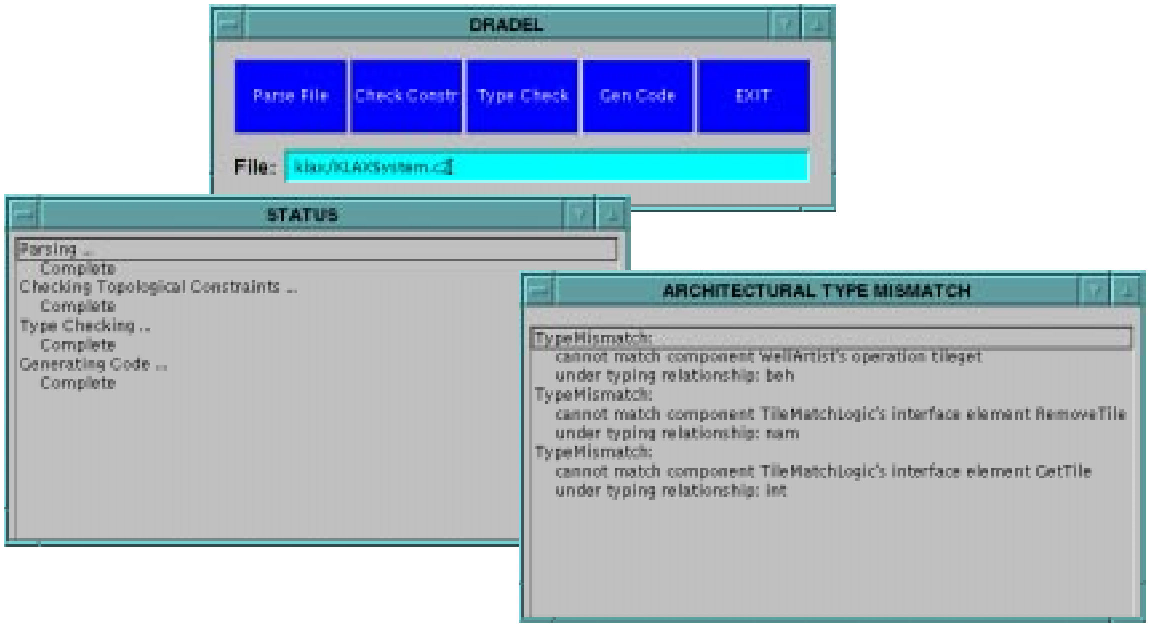
The dialog portion of the *WellADT* component is shown in Figure 5-7. Note that the modeled portion of *WellADT* does not have any required operations and, therefore, the dialog does not generate any requests. Furthermore, *WellADT* is at the top of the KLAX architecture, thus it receives no notifications from above.

Given that a component's dialog is generated almost entirely from its C2SADEL specification, the internal object may, in fact, be completely replaced by an OTS component that does not communicate via messages. Essentially, the OTS component is modeled in C2SADEL, so that DRADEL can be used to check its compliance with the rest of the architecture and/or other components in its type hierarchy, as well as to generate its C2 dialog, i.e., message wrapper. This is one of the techniques for reusing OTS components discussed in Section 5.2 above.[8]

6. The *CodeGenerator* could be easily extended to support any other policy for generating notification parameters.
7. Another reason this information has been removed from C2SADEL was discussed in Chapter 4: C2SADEL embodies the architectural type theory, which is intended to be domain-, style-, and ADL-independent. C2SADEL is thus designed to be largely a domain- and style-independent. DRADEL's CodeGenerator, on the other hand, is C2-specific, resulting in this (slight) mismatch.
8. Prior to DRADEL's development, OTS component dialogs had to be implemented manually.

```
package c2.KLAXSystem;
import c2.framework.*;
import java.lang.*;

public class WellADT_C2_Component extends ComponentThread {

   private WellADT state_var;

/***** Class Constructor *****/
   public WellADT_C2_Component(String name) {
      create(name);
   }

   public void create(String name) {
      super.create(name, FIFOPort.classType());
      recordMessageInterface();
      state_var = new WellADT();
   }

/***** Notification Handling *****/
   public void handle(Notification notif_msg) { }

/***** Request Handling *****/
   public void handle(Request req_msg) {
      if (req_msg.name().equals(“GetTile”)) {
         Natural i = (Natural)req_msg.getParameter(“i”);
         handleRequest_GetTile(i);
      } else if (req_msg.name().equals(“GetTile”)) {
         Integer location = (Integer)req_msg.getParameter(“location”);
         handleRequest_GetTile(location);
      }
   }

   private void handleRequest_GetTile(Natural i) {
      GSColor result = state_var.GetTile(i);
      notifyGetTile(state_var.GET_num_tiles(), result);
   }

   private void handleRequest_GetTile(Integer location) {
      Color result = state_var.GetTile(location);
      notifyGetTile(state_var.GET_num_tiles(), result);
   }

/***** Notification Generating Methods *****/
   private void notifyGetTile(Integer num_tiles, GSColor result) {
      Notification notif_msg = new Notification(“GetTileCompleted”);
      notif_msg.addParameter(“num_tiles”, num_tiles);
      notif_msg.addParameter(“result”, result);
      send(notif_msg);
   }

   private void notifyGetTile(Integer num_tiles, Color result) {
      Notification notif_msg = new Notification(“GetTileCompleted”);
      notif_msg.addParameter(“num_tiles”, num_tiles);
      notif_msg.addParameter(“result”, result);
      send(notif_msg);
   }

/***** Recording Interface <<< DO NOT MODIFY BELOW THIS LINE>>> *****/
   private void recordMessageInterface() {
      addMessageToInterface(“bottom”, “in”, “GetTile”);
      addMessageToInterface(“bottom”, “in”, “GetTile”);
      addMessageToInterface(“bottom”, “out”, “GetTileCompleted”);
      addMessageToInterface(“bottom”, “out”, “GetTileCompleted”);
   }
}
```

Figure 5-7. Generated dialog for the *WellADT* component.
Vertical spacing of the generated code has been altered to fit on the page.



```
klax/KLAXSystem.c2
```

```
Parsing ..
   Complete
Checking Topological Constraints ..
   Complete
Type Checking ..
   Complete
Generating Code ..
   Complete
```

```
TypeMismatch:
   cannot match component WellArtist's operation tileget
   under typing relationship: beh
TypeMismatch:
   cannot match component TileMatchLogic's interface element RemoveTile
   under typing relationship: nam
TypeMismatch:
   cannot match component TileMatchLogic's interface element GetTile
   under typing relationship: int
```

Figure 5-8. DRADEL environment's user interface.

5.4.1.7 User Interface Components

The remaining components in DRADEL's architecture, *UserPalette*, *TypeMismatch-Handler*, and *GraphicsBinding* (see Figure 5-4) handle the user's interaction with DRADEL. The *UserPalette* component drives the entire environment. It also displays the current execution status. The *TypeMismatchHandler* component informs the user of the results of all component subtype matching and architectural type checking. Finally, as with any C2-style application, the *GraphicsBinding* renders DRADEL's user interface on the screen. The user interface is shown in Figure 5-8, with examples of *nam*, *int*, and *beh* type conformance violations in the *KLAXSystem* architecture discussed and (partially) specified in Section 4.4.

Since each component in an architecture may have multiple other components along its communication links, the type mismatch information in Figure 5-8 only specifies the "end result" of type checking, rather than its pairwise breakdown. In other words, the *TypeChecker* searches for a match for a given required service only until one is found. Thus, even if a component is actually intended to interact with multiple other components in an architecture, the *TypeChecker* ensures only that it can interact with *at least* one other component. Another alternative, implemented in a previous version of DRADEL, is to attempt to match every required service to *every* component along the communication links, but this approach is much less efficient in terms of execution speed. Note that this problem does not exist in the case of component evolution as each component has an explicitly specified supertype component and need only be checked against it, regardless of the size of the component hierarchy.

5.4.2 The Architecture-Based Development and Evolution Process

The *UserPalette* component, rendered as the top and left panes of Figure 5-8, enforces the high-level, "architecting" process encoded in DRADEL. Prior to parsing a file, the *CheckConstr*, *TypeCheck*, and *GenCode* buttons are grayed out. Only once an architectural description or a set of components is successfully parsed and its internal consistency established can the user perform other functions. If the parsed file contains an architecture, the *CheckConstr* button is enabled, and the topological constraints must then be ensured. This must be done before either type checking the architecture or generating the application skeleton. If one is evolving design elements (i.e., ensuring the specified subtyping relationships among components), the *TypeCheck* and *GenCode* buttons are automatically enabled. This process is repeated every time the user decides to parse a new file.

If any errors are found during parsing, consistency checking, or topological constraint checking, a message will appear in the *Status* window informing the user of the exact error and its location. Errors encountered during any of these activities impact the subsequent operations: in the case of parsing errors, an architecture's internal model cannot be constructed and checked for consistency; in the case of internal inconsistencies, the architecture cannot be type checked; finally, in the case of violated topological constraints, an implementation of the architecture cannot be generated on top of the current implementation infrastructure. Therefore, the user is not allowed to proceed until all parsing, consistency, and topological errors are corrected.

This is not the case with architectural type checking: the user can still generate the application, even if there are type errors (*architect's discretion*), or the user may decide to skip the type checking stage altogether. The reasons for this are twofold.

Unlike a programming language, which requires complete type conformance, architectures may describe meaningful functionality even if there are interface or behavioral mismatches. This has certainly been the case with C2-style applications, in which components communicate via asynchronous messages and are substrate independent: C2 architectures are robust (hence the "R" in "DRADEL") in that a type mismatch may result in degraded functionality, or it may have no ill effects on the system, if it affects a part of the system that is not used in a given setting. It is the architect's responsibility to decide whether a given type mismatch is acceptable.

The other reason for allowing generation of type-mismatched components and architectures is the *TypeChecker* itself. To establish behavioral conformance between two components, the *TypeChecker* attempts to find mappings between their variables such that the correct (implication) relationships between their invariants, and pre- and postconditions hold. If the architect either judges the type mismatch not to be critical or discovers that the error is a result of *TypeChecker*'s pessimistic inaccuracy, the architect has the choice to override the *TypeChecker* and proceed with code generation.

5.4.3 Discussion

DRADEL has been designed to be easily evolvable. Its components can be replaced to satisfy new requirements. For example, as already discussed, the file system-based *Repository* can be replaced with a database to keep track of design elements and architectures more efficiently. Another *Parser* can be substituted to support a different ADL, while a new *TopologicalConstraintChecker*, e.g., Armani [55], can be used to ensure adherence to a different architectural style. The *TypeChecker* may be replaced with a component that supports a different notion of architectural evolution and analysis; also, additional analysis tools may be added, even at runtime, using techniques described in [61]. Finally, the existing *CodeGenerator* may be replaced or new generation components added to support different implementation infrastructures.

DRADEL itself can be used reflexively to model and ensure the consistency of its own evolution. As the diagrams in Figures 2-5 and 5-4 and their subsequent discussions indicate, there are *no* fundamental differences between DRADEL and an application modeled, analyzed, evolved, and implemented with its help. Indeed, DRADEL's architecture can be specified in C2SADEL, parsed, checked for internal consistency, type checked, and the environment itself partially generated, as discussed above, using DRADEL.

CHAPTER 6: Validation

Our approach to validating the claims of this dissertation is twofold: empirical *demonstration* of the utility of each claim and their analytical *evaluation*. To demonstrate our approach, we present a series of exercises intended to explore and confirm our hypotheses. To evaluate the results of this work, we discuss to what degree and in what manner the different exercises demonstrate our evolution techniques, extrapolate from the experience of conducting the exercises, and compare the techniques we have developed to their alternatives that represent the state of the practice.

We have already discussed two examples that provide demonstration of this dissertation's concepts: KLAX and several of its variations, in Chapters 2 and 4, and DRADEL, in Chapter 5. The next three sections describe in more detail additional exercises used to demonstrate our techniques and claims. Section 6.4 presents our evaluation of these specific exercises and our work as a whole.

6.1 OTS Component Reuse

Several examples of OTS component reuse have been discussed either in this dissertation or in our earlier work. These include GUI toolkit bindings (Chapter 5); two WWW browsers and a persistent object manager [48]; ArchShell, a tool for run-time manipulation of architectures, and Argo, a design environment [61]. We focus on the below series of exercises since it is representative of the employed technique, incorporates a medium-size and a large OTS component, and demonstrates additional issues, including partial communication, partial component service utilization, and application families.

6.1.1 Background

The KLAX architecture from Chapter 2 is used as the demonstration platform for this series of exercises. Specifically, we used the implementation of KLAX on top of the C++ version of the implementation framework. Two OTS UI constraint solvers were selected for integration with KLAX: SkyBlue, a medium-size solver [76], and Amulet, a large solver [45].[1] In its form as described in Chapter 2, KLAX does not necessarily need a constraint solver. Its constraint management needs would certainly not exploit the full power of a solver such as SkyBlue, e.g., handling constraint hierarchies. On the other hand, it should be possible to use a powerful constraint manager for maintaining a small number of simple constraints. Additionally, the main purpose of this effort was to explore the architectural issues in integrating OTS components into a C2-style architecture. We therefore opted not to unnecessarily expend resources to artificially create a situation where a number of complex constraints needed to be managed. Instead, we decided to integrate SkyBlue with KLAX to support its extant constraint management needs. If we were unable to do so, there would be at least four possible sources of problems:

1. the C2 style,

1. The sizes of the tow OTS components do not reflect their relative constraint-solving power. SkyBlue, the smaller of the two, provides more advanced constraint solving capabilities. Amulet, on the other hand, is a complete GUI builder. We only used its one-way formula constraint manager, but this also required the inclusion of Amulet's object system, substantially adding to the extracted component's size. For simplicity, we refer to the solver as "Amulet" in the remainder of this chapter. The entire Amulet system will be referred to as "the Amulet GUI builder."

2. our strategy for OTS integration (see Chapter 4),
3. the KLAX architecture, and
4. the OTS constraint solver.

In any case, we would learn a useful lesson.

We defined the following four constraints for management by a solver:

- *Palette Boundary:* The palette cannot move beyond the chute and well's left and right boundaries.
- *Palette Location:* Palette's coordinates are a function of its location and are updated every time the location changes.[2]
- *Tile Location:* The tiles which are on the palette move with the palette. In other words, the *x* coordinate of the center of the tile always equals the *x* coordinate of the center of the palette.
- *Resizing:* Each game element (chute, well, palette, and tiles), is maintained in an abstract coordinate system by its artist. This constraint transforms those abstract coordinate systems, resizing the game elements to have the relative dimensions depicted in Figure 2-4 on page 12 before they are rendered on the screen. This constraint would be essential in a case where the application is composed from preexisting components supplied by different vendors. A similar constraint could also be used to accommodate resizing of the game window, and hence of the game elements within it.

6.1.2 Integrating SkyBlue with KLAX

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

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

The constraints were defined in the "dialog and constraints" part of the *LayoutManager* component (see Figure 6-1), while SkyBlue became the component's internal object. As such, SkyBlue has no knowledge of the architecture of which it is now a part. It maintains the constraints, while all the request and notification traffic is handled by *LayoutManager*'s dialog, as shown in Figure 6-1. *LayoutManager* thus became a constraint management component in the C2 style that can be reused in other applications by only modifying its dialog to include new constraints.[3]

2. Location is an integer between 1 and 5.

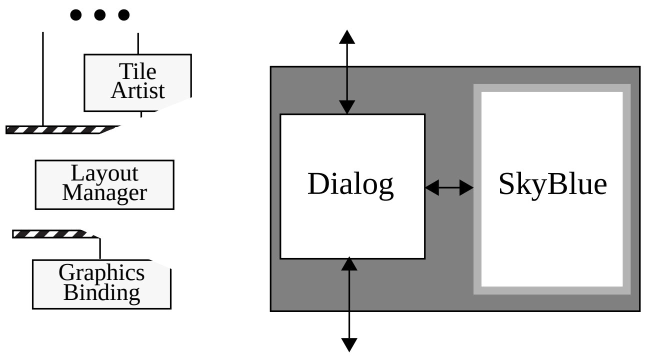


Figure 6-1. A C2-style UI constraint management component.
The SkyBlue constraint management system is incorporated into KLAX by placing it inside the *LayoutManager* component. *LayoutManager*'s dialog handles all the C2 message traffic.

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

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

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

6.1.3 Integrating Amulet with KLAX

C2 supports reuse through the internal component architecture, substrate independence, and asynchronous communication via connectors. These features also support component substitutability and localization of change. In general, two behaviorally equivalent components can always be substituted for one another; behavior preserving modifications to a component's implementation have no architecture-wide effects (see Section 4.1). In the example discussed in the previous section, this would mean that SkyBlue may be replaced with another constraint manager by only having to modify the "dialog and constraints" portion of *LayoutManager* to define constraints as required by the new solver. The set of messages in *LayoutManager*'s interface and the rest of the KLAX architecture would remain unchanged.

3. In the remainder of the paper, when we state that a constraint solver is "inside" or "internal to" a component, the internal architecture of the component will resemble that of *LayoutManager* from Figure 6-1.

To demonstrate this claim, we substituted SkyBlue with Amulet. This exercise required identifying, extracting, and recompiling the needed portion of the Amulet GUI builder, a task that was accomplished by a single developer in approximately 25 hours. This added effort was necessitated by our inability to locate implementations of any other UI constraint solvers. It resulted in a situation that is common when attempting software reuse: OTS systems may not contain components that can be clearly identified or easily isolated and extracted [8], [24], [38].

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

6.1.4 KLAX Component Library

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

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

The two integrations described above resulted in three versions of *LayoutManager*: the original, SkyBlue, and Amulet versions. These are listed as *LayoutManager* versions 1, 2, and 3 in Table 6-1. Two versions each of *PaletteADT*, *PaletteArtist*, *ChuteArtist*, and *WellArtist* were created as well: original components maintaining local constraints with in-line code (versions 1 of the four components in Table 6-1) and components whose constraints were managed elsewhere in the architecture (versions 2 of the four components in Table 6-1).[4]

The two initial integrations also suggested other variations of these components, such as replacing in-line constraint management code with SkyBlue and Amulet constraints in *PaletteADT* and *PaletteArtist* (see Footnote 3). Also, a version of *LayoutManager* was implemented that maintained only the *Resizing* constraint, in anticipation that other components will internally manage their local constraints (this scenario was briefly described at the beginning of Section 6.1.2). This resulted in a total of 18 implemented versions of the five components, as depicted in Table 6-1.

6.1.5 Building an Application Family

The four versions of *PaletteADT* and *PaletteArtist,* two versions of *ChuteArtist* and *WellArtist,* and six versions of *LayoutManager*, described in Table 6-1, could potentially be used to build 384 different variations of the KLAX architecture, i.e., members of the KLAX application family. Three such variations were described in Chapter 2 (using versions 1 of all five components),

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

Table 6-1: Implemented Versions of *PaletteADT, PaletteArtist, ChuteArtist, WellArtist,* and *LayoutManager* KLAX Components

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

Section 6.1.2 (using versions 2 of the five components), and Section 6.1.3 (replacing *LayoutManager*-2 with *LayoutManager*-3 in the architecture from Section 6.1.2). In this section, we discuss several additional implemented variations of the architecture that exhibit interesting properties.

6.1.5.1 Multiple Instances of a Constraint Manager

In the architecture depicted in Table 6-2, the *Palette Boundary*, *Palette Location*, and *Tile Location* constraints are defined and maintained in SkyBlue inside *PaletteADT* and *PaletteArtist*, while the *Resizing* constraints are maintained globally by *LayoutManager*. Therefore, multiple instances of SkyBlue maintain the constraints in different KLAX components. Since C2 separates architecture from implementation, we were able to implement the three components that contain their own logical copies of SkyBlue using a single physical instance of the constraint manager.

6.1.5.2 Partial Communication and Service Utilization

Particularly interesting are components that are used in an architecture for which they have not been specifically designed, i.e., they can do more or less than they are asked to do. This is an

Table 6-2: Multiple Instances of SkyBlue

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

Table 6-3: None of *LayoutManager*'s Constraint Management Functionality is Utilized

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

issue of reuse: if components are built a certain way, are their users (architects) always obliged to use them "fully"; furthermore, can meaningful work be done in an architecture if two components communicate only partially, i.e., certain messages are lost? The architectures described below represent a crossection of exercises conducted to better our understanding of partial communication and partial component service utilization.

- A variation of the original architecture was implemented by substituting *LayoutManager*-2 into the original architecture, as shown in Table 6-3. *LayoutManager*-2's functionality remains largely unused as no notifications are sent to it to maintain the constraints (see Section 6.1.2). The application still behaves as expected and there is no performance penalty. Note that this will not always be the case: if *LayoutManager*-2 was substantially larger than *LayoutManager*-1 or had much greater system resource needs (e.g., its own operating system process), the performance would be affected.
- Another variation of the architecture that was implemented is shown in Table 6-4. This exercise was intended to explore heterogeneous approaches to constraint maintenance in a single architecture: some components in the architecture maintain their constraints with in-line code (*WellArtist* and *ChuteArtist*), others maintain them internally using SkyBlue (*PaletteADT*), while *PaletteArtist*'s constraints are maintained by an external constraint manager. *LayoutManager*-2 is still partially utilized, but a larger subset of its services is used than in the preceding architecture.
- In the architecture shown in Table 6-5, *PaletteADT* expects that the *Palette Boundary* constraint will be maintained externally by some other component. However, in this case, *LayoutManager*-1 does not understand and therefore ignores the notifications sent by

Table 6-4: ***LayoutManager*****'s Constraint Management Functionality is Only Partially Utilized**

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

Table 6-5: ***Palette Boundary*** **Constraint is not Maintained**

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

PaletteADT (partial communication). Movement of the palette is thereby not constrained and the application behaves erroneously: the palette disappears when moved beyond its right boundary; the execution aborts when the palette moves beyond the left boundary and the *GraphicsBinding* component (see Section 5.1) attempts to render it at negative screen coordinates.

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

6.1.5.3 Multiple Constraint Managers in an Architecture

Combining multiple constraint solvers in a single application has recently been identified as a potentially useful approach to constraint management [45], [76]. We investigated this issue by using multiple constraint managers in different components in a single architecture. Such an architecture was implemented using components shown in Table 6-6. In this architecture, *Palette*

Table 6-6: *Palette Boundary* Constraint is not Maintained

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

Boundary and *Resizing* constraints are maintained by SkyBlue, and *Palette Location* and *Tile Location* by Amulet. The sets of constraint variables managed by the two solvers are disjoint, and there are no interdependencies between SkyBlue and Amulet that would have required us to account for their different type systems.[5] Architectures built according to the C2 style will always have this property: since C2 does not assume a single address space for its components, inter-component constraint variable sets will always be disjoint. Hence, this modification to the architecture was a simple one.

6.2 OTS Connector Reuse

In programming languages, connectors are primitive and implicit in, e.g., procedure calls and global variables. Since software components at the architectural level may contain complex functionality, it is reasonable to expect that their interactions will be complex as well. Modeling and implementing software connectors with potentially complex protocols thus becomes a key aspect of architecture-based development [3], [52], [82].

While practitioners are typically intimately familiar with “connecting” software modules via, e.g., procedure calls, their understanding of other interconnection mechanisms, e.g., client-server protocols and message routers, is often minimal. Several commercial and research off-the-shelf (OTS) middleware software systems that explicitly implement such interconnection mechanisms are available: Field [73], SoftBench [13], Tooltalk [34], Q [44], Polylith [72], DCE [78], CORBA [62], ILU [97], COM/DCOM [80], and ActiveX [15]. Also, several object-oriented (OO) programming languages provide remote procedure call (RPC) mechanisms. A representative example is Java’s Remote Method Invocation (RMI) system [87]. Unfortunately, the applicability of these mechanisms and tools to software architectures is not well understood [17]. They are rarely used by architecture researchers in practice. With the exception of UniCon [82], the focus of researchers has instead generally been on formal modeling of connector protocols with implementation support for simple connections only.

To explore the use of OTS middleware packages with software connectors, we chose four representative middleware technologies. These were Q, an RPC system [44], Polylith, a message bus [72], RMI, a connection mechanism for Java objects [87], and ILU, a distributed objects package [97]. Each middleware technology was integrated with the C2 implementation infrastructure to create an additional implementation of a C2 connector. Each integration

5. The difference in the two solvers’ type systems requires special treatment if both are used inside a single component, as discussed in [50].

employed one of the techniques discussed in Section 4.3. The middleware technologies were evaluated using the criteria specified in Section 5.3. The results of these integrations are discussed below.

6.2.1 Q

The Q system, developed at the University of Colorado, is intended to provide interoperability support for multilingual, heterogeneous component-based systems. Q presents a layer of functionality between software components communicating across process boundaries. It is based on remote procedure calls (RPC) and provides support for marshaling and unmarshaling of arbitrarily complex type structures. Q also supports placement of components executing in a single thread or in multiple threads of control inside a single process. It ensures the proper communication of multi-threaded components with other parts of a system. Q addresses the issue of performance by adding an asynchronous message interface on top of a standard RPC interface, so that processor time is used for interprocess communication only when it is known that data is pending.

Q uses a remote procedure call (RPC) mechanism for communication, which is dissimilar to C2's message-based style. Nonetheless, we easily emulated message passing using RPC by passing serialized messages as parameters in remote calls. Q supports systems built in several languages: C/C++, Ada, Java, Tcl, Lisp, and Prolog. It was originally built for the UNIX platform, although its Java interface presents the potential for moving to other platforms. We have made use of its support for C/C++ and Ada with the intent to exploit its support for Java in the near future.

Our approach to integrating Q with the C2 implementation infrastructure consisted of encapsulating Q inside a C2 connector (we refer to it as a "Q-C2 connector" below). Q is not a software bus, so it does not support typical connector-like features, such as event registration, filtering, and routing. However, this layer of support is added easily in a Q-C2 connector.

A Q-C2 connector exports the same interface as a regular C2 connector, so architects attach components to it in the usual manner. Internally, however, a Q-C2 connector provides a mechanism for communicating across process boundaries via Q. At each process boundary, a conceptual C2 connector is horizontally "broken up" into two or more Q-C2 connectors, one per process, as shown in Figure 4-16b on page 64. When using Q-C2 connectors, all processes containing C2 subarchitectures must register with a single "name server." All links across process boundaries are specified in the Q-C2 connector, by naming the attached connectors, and are maintained by Q at execution time. Clearly, care must be taken to ensure that there are no naming conflicts, i.e., that multiple Q-C2 connectors do not share a name.

Given that we can explicitly specify the connections among Q-C2 connectors in an architecture, a single instance of Q is sufficient to support the needs of an architecture. Since Q is UNIX-based, it supports addition and removal of processes at execution time. Any additional support for dynamism, such as transactions, state preservation during change, or component (i.e., process) replacement, must be built on top of Q.

We used Q to generate a multi-process version of KLAX, shown in Figure 4-14 on page 62. Connectors *IPconn1* and *IPconn2* were used at process boundaries. The rest of the application remained identical to single-process KLAX. This three-process configuration allowed us to explore issues in supporting multilingual applications in C2. For example, we were able to replace the "middle" process in KLAX, where the *TileArtist* component and both connectors were initially implemented in C++, with their Ada implementations. This can be done at specification- or run-time. If the change is made at run-time, a part of the game state is lost, as no one receives the notifications issued by components in the “top” process or requests issued by the “bottom” process components during the course of the change. The performance of this variation of KLAX easily exceeded human reaction time if the *ClockLogic* component used short time intervals.

6.2.2 Polylith

The Polylith software bus was developed at the University of Maryland. Polylith was built to allow several parts of an application to communicate across process boundaries using messages made up of arbitrarily complex type structures. Polylith uses messages for communication, which made it well-suited for implementing C2-style connectors. Polylith can transfer messages among processes running on a single machine or on multiple machines using the TCP/IP networking protocol. The Polylith toolkit is implemented in C and runs on several variants of UNIX. Polylith currently supports applications developed in C/C++.

Polylith is inherently built to communicate among UNIX processes. Although there is no support for multithreading in Polylith, multiple threads within a process are allowed in principle. Polylith has support for marshaling and unmarshaling of C basic types and structures. The Polylith bus itself runs in its own process and acts as a message queue for other processes, which are individually responsible for periodically sending and retrieving messages to and from the bus.

Like the Q-C2 connector, the “Polylith-C2” connector is an extension of the standard, in-process C2 connector: at each process boundary, a conceptual connector is “broken up” vertically into Polylith-C2 connectors, as shown in Figure 4-16a on page 64. All access to Polylith is done within the C2 connector, and is transparent. Components can attach themselves to a Polylith-C2 connector in the usual manner.

The process-level structure of a C2 application that uses Polylith is defined statically, i.e., at compile time, using a proprietary language called MIL. The MIL code can be generated automatically in a fairly straightforward manner. As a software bus, Polylith has the ability to route messages at the process level, but it is necessary to implement one’s own intra-process routing mechanisms. There is no support for message filtering in Polylith.

The current Polylith toolkit uses the UNIX process scheduler for all process scheduling. Polylith applications with specific scheduling needs must explicitly make system-level calls from within the application. Such performance limitations became problematic when Polylith-C2 was used in the implementation of the KLAX application. The implementation suffered from poor performance due to the UNIX process scheduler giving large time slices to each process, resulting in messages being handled in bursts rather than in a fluid manner. This may be unacceptable in a real-time application such as KLAX.[6]

6.2.3 RMI

Java’s Remote Method Invocation (RMI) is a technology developed by Sun Microsystems to allow Java objects to invoke methods of other objects across process and machine boundaries.

6. The authors of Polylith acknowledge this problem; an experimental, as yet unreleased version of Polylith alleviates this shortcoming.

RMI supports several standard distributed application concepts, namely registration, remote method calls, and distributed objects. Currently, RMI only supports Java applications.[7]

Each RMI object that is to be shared in an application defines a public interface (a set of methods) that can be called remotely. This is similar to the RPC mechanism of Q. These methods are the only means of communication across a process boundary via RMI. Because RMI is not a software bus, it has no concept of routing, filtering, or messages. However, Java's built-in serialization and deserialization capabilities handle marshaling of basic and moderately complex Java objects, including C2 messages.

RMI is fully compatible with the multithreading capabilities built into the Java language, and is therefore well suited for a multithreaded application. It allows communication among objects running in different processes, which may be on different machines. Communication occurs exclusively over the TCP/IP networking protocol.

We successfully applied to RMI all three strategies for providing distribution support to C2 architectures, discussed Section 4.3 and depicted in Figures 4-15 and 4-16. Like the Polylith- and Q-integrated connectors, the RMI-C2 connector we developed has all the capabilities of a single-process C2 connector. Additionally, it has the ability to register and deregister itself at run-time with the Java-RMI name server, and to be linked to other registered connectors. All access to RMI facilities is encapsulated within the connector and is transparent.

Minimal modification was required to convert the existing Java implementation of the C2 KLAX application into a multi-process application that uses RMI-C2 connectors. RMI supports application modification at run-time, a capability enabled by Java's dynamic class loading. The performance of the three-process implementation of KLAX using RMI-C2 was satisfactory. Another variation of the KLAX application built using RMI-C2 connectors was multiplayer KLAX. This variation allowed players to remotely join a game already in progress and compete against other participants.

RMI's properties make it ideal for use within a Java C2 application. Its native support in Java 1.1 makes it more easily available to architecture implementors than third party alternatives. Also, using software connectors that work with RMI does not preclude an application implemented partially or completely in Java from using another middleware technology, such as Q or ILU, as well.

6.2.4 ILU

Xerox PARC's ILU (Inter-Language Unification) was developed as a free CORBA-compliant object brokering system. Functionally, it is similar to RMI, allowing objects to call methods on other objects across process or network boundaries. ILU is different from RMI in that it has wide platform and language support: C, C++, Java, Python, LISP, Modula-3, Perl, and Scheme, on both Windows and UNIX platforms. The current ILU implementation can be thought of as a CORBA Object Request Broker (ORB), but ILU is not yet fully CORBA compliant.

Like RMI, each ILU object that is to be shared in an application defines a public set of methods that can be called remotely. There is no inherent concept of messages in ILU, but messages can be passed as parameters in remote method calls. Similarly to Q, ILU has the ability to serialize moderately complex objects across language boundaries. As with other distributed

7. A forthcoming link between RMI and CORBA will remedy RMI's exclusive support for Java.

object systems, object references are not preserved across the serialization boundary. ILU does not include a name server, but it facilitates object registration through a method called “simple binding” that is part of the ILU package. Our integration of ILU with C2 was done using the vertical “slicing” technique depicted in Figure 4-16a. The Java implementations of the C2 framework and the ILU package were used. The ILU-C2 connector thus created has all the capabilities of an in-process C2 connector, but it is also capable of lateral connection to ILU-C2 connectors in other processes. Again, all access to ILU is done entirely within the connector, in a manner that is transparent to architects and developers.

ILU takes full advantage of Java’s multithreading capabilities and works in multithreaded applications implemented in other languages, even if such threading is provided by the operating system rather than the language itself. This makes it well suited for real-time, asynchronous message passing architectures, such as C2-style architectures. Minimal modification was required when converting a single-process C2 application to a multi-process C2 application. ILU allows objects to be registered and deregistered at run-time, therefore enabling dynamic application construction at run-time. We utilized this feature to demonstrate a set of components and connectors joining a larger, already executing application at run-time.

6.2.5 Simultaneous Use of Multiple Middleware Technologies

We combined our implementations of ILU-C2 and RMI-C2 connectors in a single application. This was accomplished by combining the lateral welding technique with the horizontal slicing technique, as discussed in Section 5.3. As anticipated, no modification was required to the C2 framework or the connectors themselves.

6.3 Architectural Subtyping and Type Checking

We use a logistics system for routing incoming cargo to a set of warehouses to demonstrate component evolution via subtyping, architectural type checking, and implementation generation from an architectural description. This example is a variant of the application first introduced in [61]. Its architecture is shown in Figure 6-2. The *DeliveryPort*, *Vehicle*, and *Warehouse* component types are objects that keep track of the state of a port, a transportation vehicle, and a warehouse, respectively. Each of them may be instantiated multiple times in a system. The *DeliveryPortArtist*, *VehicleArtist*, and *WarehouseArtist* components are responsible for graphically depicting the states of their respective sets of objects to the end-user. Each organizes its depiction based on the actual number of its object instances. The *Layout Manager* ensures that artist depictions are correctly juxtaposed on the screen. *SystemClock* provides consistent time measurement to interested components, while the *Map* component informs vehicles of routes and distances. The *Router* component determines when cargo arrives at a port, keeps track of available transport vehicles at each port, and tracks the cargo during its delivery to a warehouse. *RouterArtist* allows entry of new cargo as it arrives at a port and informs the *Router* component when the end-user decides to route cargo. The *GraphicsBinding* component renders the drawing requests sent from the artists using a GUI toolkit.

A screenshot of the Cargo Router application is given in Figure 6-3. From top to bottom, the main window shows the current states of the delivery ports, vehicles, and warehouses, respectively. The depicted version of the application has three delivery ports: two airport runways and a train station. When cargo arrives at a port, an item is added to the port’s list box, containing

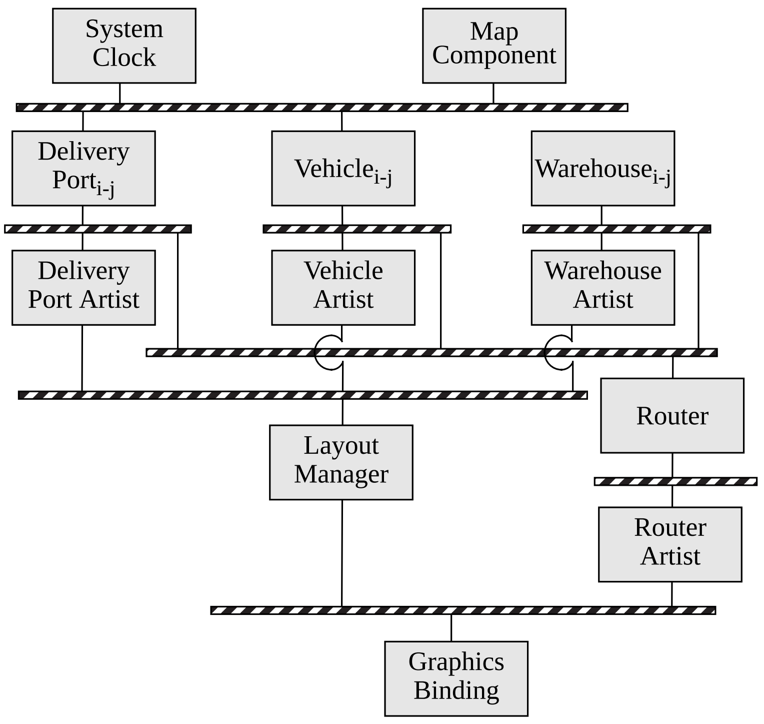
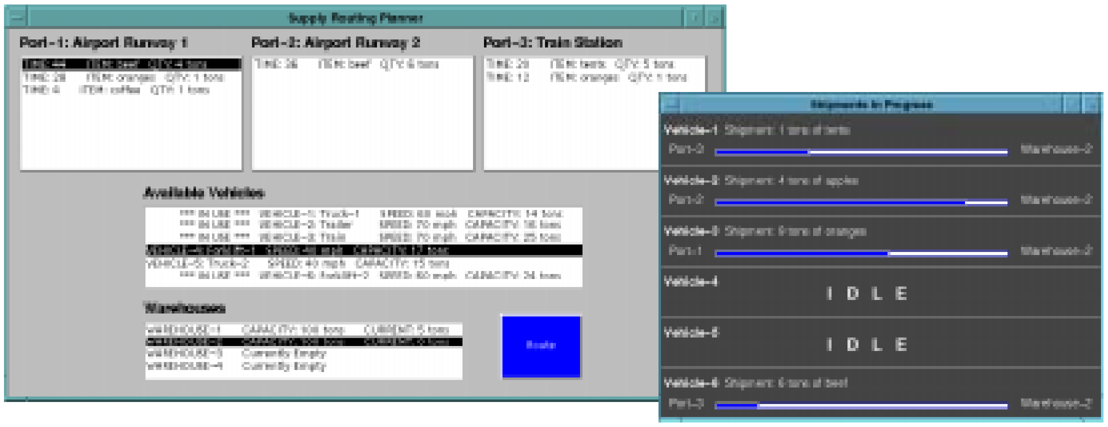


Figure 6-2. Architecture of the Cargo Router system in the C2 style.
The *DeliveryPort*, *Vehicle*, and *Warehouse* components are instantiated multiple times in the architecture to keep track of individual ports, vehicles, and warehouses, respectively. They are depicted by single artists (*DeliveryPortArtist*, *VehicleArtist*, and *WarehouseArtist*, respectively).

the item's name, weight, and the time elapsed since its arrival. The box in the center shows the status of vehicles used to transport cargo from delivery ports to warehouses. Each vehicle has a maximum speed and capacity. A vehicle is either idle or in transit. Finally, at the bottom of the main window is a text box that displays the current status of the available warehouses: the maximum capacity and the currently used portion of the warehouse. End-users route cargo by selecting an item from a delivery port, an available vehicle, and a destination warehouse, and clicking on the "Route" button.

Several extensions to this basic architecture were discussed in [61]. These extensions can be added at specification-time or at run-time. One extension involves adding an artist to provide more information about the exact status of vehicles in transit. The artist's depiction is shown in the right pane in Figure 6-3. In the architecture from Figure 6-2, this information is maintained by the *Router* component; an artist component can be easily added next to the *RouterArtist* in the architecture to display the information. Another extension of the original architecture, not modeled in the variation shown in Figure 6-2, involves adding an automatic planner component to the architecture. This component automatically selects an item at a delivery port, a vehicle, and a warehouse based on some set of heuristics (e.g., shipment with the longest wait time, the fastest idle vehicle, and the emptiest warehouse); the user can "Route" the suggested selection or override the planner. This functionality can be added to the shown version of the architecture either by evolving the *Router* component or by evolving the configuration to add a separate *Planner* component alongside the *Router* component.



Supply Routing Planner

Port-1: Airport Runway 1

```
[illegible]
TIME: 28     ITEM: oranges   QTY: 1 tons
TIME: 4      ITEM: coffee    QTY: 1 tons
```

Port-2: Airport Runway 2

```
TIME: 26     ITEM: beef   QTY: 6 tons
```

Port-3: Train Station

```
TIME: 20     ITEM: [illegible]   QTY: 5 tons
TIME: 12     ITEM: oranges   QTY: 1 tons
```

Available Vehicles

```
     *** IN USE ***  VEHICLE-1: Truck-1    SPEED: 60 mph   CAPACITY: 14 tons
     *** IN USE ***  VEHICLE-2: Trailer    SPEED: 70 mph   CAPACITY: 16 tons
     *** IN USE ***  VEHICLE-3: Train      SPEED: 70 mph   CAPACITY: 25 tons
VEHICLE-4: [illegible]   SPEED: 40 mph   CAPACITY: [illegible] tons
VEHICLE-5: Truck-2    SPEED: 40 mph   CAPACITY: 15 tons
     *** IN USE ***  VEHICLE-6: [illegible]   SPEED: 60 mph   CAPACITY: 24 tons
```

Warehouses

```
WAREHOUSE-1    CAPACITY: 100 tons    CURRENT: 5 tons
[illegible]
WAREHOUSE-3    Currently Empty
WAREHOUSE-4    Currently Empty
```

Route

Shipments In Progress

```
Vehicle-1  Shipment: 1 tons of [illegible]
  Port-2                                   Warehouse-2
Vehicle-2  Shipment: 4 tons of apples
  Port-2                                   Warehouse-2
Vehicle-3  Shipment: 6 tons of oranges
  Port-1                                   Warehouse-2
Vehicle-4
                 I D L E
Vehicle-5
                 I D L E
Vehicle-6  Shipment: 6 tons of beef
  Port-2                                   Warehouse-2
```

Figure 6-3. Cargo Router system's user interface.
Accompanying the main window is the a graphical depiction of the vehicles' current status (on the right).

6.3.1 C2SADEL Specification

A C2SADEL specification of the Cargo Router architecture is shown in Figure 6-4. The figure does not contain the descriptions of any of the components, all but one of which are specified externally, in the *Planner* directory. As already discussed in Chapter 4, the *GraphicsBinding* component is modeled as a virtual component. The configuration shown in the figure is defined with single instances of *DeliveryPort*, *Warehouse,* and *Vehicle* types, corresponding exactly to the diagram in Figure 6-2.

6.3.1.1 Component Evolution

The Cargo Router system's main entities—delivery ports, vehicles, and warehouses—share a number of traits. As indicated in the screenshot in Figure 6-3, the ports and vehicles also require access to a clock (modeled as a separate component in this application), while the warehouses do not. We therefore specify a general *TimedCargo-RouteEntity* type that contains the features of a Cargo Router entity, shown in Figure 6-5. The *TimedCargoRouteEntity* component has one required service, `clockTick`. *DeliveryPort* and *Vehicle* components are subtyped from *TimedCargoRouteEntity.* To ensure that the *Warehouse* component and any similar components that may be added to the application in the future are properly modeled, *TimedCargoRouteEntity* is subtyped to form another general design element, *CargoRouteEntity*, which does not require any services.[8] *Warehouse* is, in turn, subtyped from *CargoRouteEntity.* The *DeliveryPortArtist*, *VehicleArtist*, and *WarehouseArtist* components also have commonalities; they are subtyped from the more general *CargoRouteEntityArtist* type.

The C2SADEL specification of the *DeliveryPort* component is shown in Figure 6-6. *TimedCargoRouteEntity* is evolved such that both its interface and behavior are preserved. Note that another conjunct, expressing an invariant property of *DeliveryPort*'s internal timer, is added to the component's invariant expression. As in *TimedCargoRouteEntity*, the `clockTick` interface element is required; the operation to which `clockTick` is mapped models the

8. Recall from the specification of the type theory in Chapter 4 that a subtype must provide *at least* the appropriate elements (e.g., operations) of the supertype, but may require *at most* those required by the supertype.

anticipated *Clock* component's state variable as a generic STATE_VARIABLE type, as discussed in the preceding chapters. Currently implicit in the component specification is the fact that the value of *DeliveryPort*'s `internal_clock` will be synchronized with the external clock, using the *TimeIncrement* component service.[9] This information can be added as a comment in the current version of C2SADEL.[10]

6.3.1.2 Type Checking the Architecture

To demonstrate type checking of the Cargo Router architecture, we model the *Clock* component, shown in Figure 6-7. Its provided operation `op_1`'s postcondition matches with the required operation `or_clktck`'s postcondition from the *DeliveryPort* component (Figure 6-6) when `or_clktck`'s variable `t` of type STATE_VARIABLE is instantiated with *Clock*'s internal variable `time`. However, note that as they are currently specified, the two relevant interface element names do not match: *Clock* provides a `Tick`, while *DeliveryPort* requires a `clockTick`. This, and a number of other mismatches, were discovered by DRADEL in the process of specifying the Cargo Router architecture. In this case, the modification of the *DeliveryPort* component is trivial; in more complex cases, a domain translator may be needed.

As another example of type checking, we show the *DeliveryPortArtist* component in Figure 6-8. Its required services include `Select` and `Deselect`, provided by *DeliveryPort*. Note that *DeliveryPortArtist*'s `Deselect` has a single parameter of type `Number`, while *DeliveryPort* provides `Deselect` with a parameter of type `Integer`. Since `Integer` is declared to be a basic subtype of `Number` (see Figure 6-4), the provided and required interface elements match. *DeliveryPortArtist*'s required `or4` operation, which corresponds to the `Select` interface element, contains two placeholder variables (of type STATE_VARIABLE), `selection` and `cargo_size`. To establish `or4`'s behavior conformance to *DeliveryPort*'s `op_selshp` operation, `selection` must be instantiated with *DeliveryPort*'s internal variable `selected` and `cargo_size` with the cardinality of *DeliveryPort*'s `set` variable `cargo`.

Finally, *DeliveryPortArtist* is representative of artists currently modeled in C2SADEL. Although *GraphicsBinding* is treated as a virtual type and does not affect architectural type checking, we can model an artist's provided services such that its generated implementation will adhere to *GraphicsBinding*'s interface. Specifically, `InitVport`'s parameters in *DeliveryPortArtist* are required by the *GraphicsBinding* to display a window. In this particular architecture, only the *LayoutManager* component interacts directly with the *GraphicsBinding* (see Figure 6-2) and is responsible for managing the entire application's depiction. The *LayoutManager* provides a `CreateViewport` service. DRADEL's *CodeGenerator* will generate a `CreateViewportCompleted` notification resulting from this service, which is, in turn, expected by the *GraphicsBinding*.

9. The service refers to the interface element labeled `ip_timinc` and its corresponding operation labeled `op_timinc`.
10. As discussed in Chapter 5, a previous version of C2SADEL did model this kind of information.

```
PlannerSystem is {
   basic_types {
      Integer is basic_subtype Number;
      Natural is basic_subtype Integer;
   }
   component_types {
      component Map is extern { Planner/Map.c2; }
      component Clock is extern { Planner/Clock.c2; }
      component DeliveryPort is extern { Planner/InPort.c2; }
      component Vehicle is extern { Planner/Vehicle.c2; }
      component DeliveryPortArtist is extern { Planner/InPortArtist.c2; }
      component VehicleArtist is extern { Planner/VehicleArtist.c2; }
      component LayoutManager is extern { Planner/LayoutManager.c2; }
      component CargoRouter is extern { Planner/CargoRouter.c2; }
      component RouterArtist is extern { Planner/RouterArtist.c2; }
      component Warehouse is extern { Planner/Warehouse.c2; }
      component WarehouseArtist is extern { Planner/WarehouseArtist.c2; }
      component GraphicsBinding is virtual { }
   }
   connector_types {
      connector FilteringConn is { message_filter message_filtering; }
      connector RegularConn is { message_filter no_filtering; }
   }
   architectural_topology {
      component_instances {
         SimClock : Clock;                    Runway : DeliveryPort;
         RunwayArt : DeliveryPortArtist;      Whouse : Warehouse;
         WhouseArt : WarehouseArtist;         Truck : Vehicle;
         DistanceCalc : Map;                  VehicleArt : VehicleArtist;
         Router : CargoRouter;                RouteArt : RouterArtist;
         LayoutArtist : LayoutManager;        Binding : GraphicsBinding;
      }
      connector_instances {
         UtilityConn : FilteringConn;         RunwayConn : FilteringConn;
         TruckConn : FilteringConn;           WhouseConn : FilteringConn;
         RouterConn : FilteringConn;          RouterArtConn : FilteringConn;
         LayoutArtistConn : FilteringConn;    BindingConn : RegularConn;
      }
      connections {
         connector UtilityConn
            { top SimClock, DistanceCalc; bottom Runway, Truck; }
         connector RunwayConn { top Runway; bottom RunwayArt; }
         connector TruckConn { top Truck; bottom VehicleArt; }
         connector WhouseConn { top Whouse; bottom WhouseArt; }
         connector RouterConn
            { top TruckConn, RunwayConn, WhouseConn; bottom Router; }
         connector RouterArtConn { top Router; bottom RouteArt; }
         connector LayoutArtistConn
            { top RunwayArt, VehicleArt, WhouseArt; bottom LayoutArtist; }
         connector BindingConn
            { top LayoutArtist, RouteArt; bottom Binding; }
      }
   }
}
```

Figure 6-4. C2SADEL specification of the Cargo Router architecture.
All components are defined externally, except *GraphicsBinding*, which is virtual (see Section 4.4). All connectors but one are instances of a filtering (point-to-point) connector type; the lone exception is *BindingConn*, which is a broadcasting connector.

```
component TimedCargoRouteEntity is {
   state {
      cargo : \set Shipment;           cargo_val : \set String;
      name : String;                   max_capacity : Integer;
      capacity : Integer;              internal_clock : Integer;
   }
   invariant
      { (capacity \eqgreater 0) \and (capacity \eqless max_capacity); }
   interface {
      prov ip_setcap: SetCapacity(c : Integer);
      prov ip_getcap: GetCapacity() : Integer;
      prov ip_plcshp: PlaceShipment(name : String; shp : Shipment);
      prov ip_getshp: GetShipment(name : String; loc : Integer) : Shipment;
      prov ip_remshp: RemoveShipment(name : String; loc : Integer);
      prov ip_getcrg: GetContentInfo() : \set String;
      req  ir_clktck: clockTick();
   }
   operations {
      prov op_setcap: {
         let num : Integer;
         post ~capacity = num;
      }
      prov op_getcap: {
         post \result = capacity;
      }
      prov op_plcshp: {
         let shp : Shipment; n : String; shp_size : Integer;
         pre name = n;
         post (shp \in ~cargo) \and (~capacity = capacity + shp_size);
      }
      prov op_getshp: {
         let shp_loc : Integer; n : String; shp_by_loc : Shipment; //fn
         pre  name = n;
         post (\result = shp_by_loc);
      }
      prov op_remshp: {
         let shp_loc : Integer; n : String;
             shp_by_loc : Shipment; shp_size : Integer; // fn
         pre (name = n) \and (shp_by_loc \in cargo);
         post shp_by_loc \not_in ~cargo \and ~capacity = capacity - shp_size;
      }
      prov op_getcrg: {
         post \result = cargo_val;
      }
      req or_clktck: {
         let time : STATE_VARIABLE;
         post ~time = time + 1;
      }
   }
   map {
      ip_setcap -> op_setcap (c -> num);
      ip_getcap -> op_getcap ();
      ip_plcshp -> op_plcshp (name -> n, shp -> shp);
      ip_getshp -> op_getshp (name -> n, loc -> shp_loc);
      ip_remshp -> op_remshp (name -> n, loc -> shp_loc);
      ip_getcrg -> op_getcrg ();
      ir_clktck -> or_clktck ();
   }
}
```

Figure 6-5. C2SADEL specification of the *CargoRouteEntity* component type.
The portion of the specification that modifies and accesses the name and maximum capacity of the component has been elided for brevity.

```
component DeliveryPort is subtype TimedCargoRouteEntity (int \and beh) {
   state {
      cargo : \set Shipment;           cargo_val : \set String;
      name : String;                   max_capacity : Integer;
      capacity : Integer;              internal_clock : Integer;
      selected : Integer;
   }
   invariant {
      (capacity \eqgreater 0) \and (capacity \eqless max_capacity) \and
      (internal_clock \eqgreater 0);
   }
   interface {
      /*** several provided interface elements elided ***/
      prov ip_timinc: TimeIncrement();
      prov ip_selshp: Select(sel : Integer);
      prov ip_dslshp: Deselect(sel : Integer);
      req  ir_clktck: clockTick();
   }
   operations {
      /*** several provided operations elided ***/
      prov op_dslshp: {
         let num : Integer;
         pre num = selected;
         post ~selected = -1;
      }
      prov op_selshp: {
         let num : Integer;
         pre num \less #cargo;
         post ~selected = num;
      }
      prov op_timinc: {
         post ~internal_clock = internal_clock + 1;
      }
      req or_clktck: {
         let t : STATE_VARIABLE;
         post ~t = t + 1;
      }
   }
   map {
      /*** several maps elided ***/
      ip_dslshp -> op_dslshp (sel -> num);
      ip_selshp -> op_selshp (sel -> num);
      ip_timinc -> op_timinc ();
      ir_clktck -> or_clktck ();
   }
}
```

Figure 6-6. C2SADEL specification of the *DeliveryPort* component type.
The portions of the component that have remained unchanged from the specification of *CargoRouteEntity* in Figure 6-5 have been elided for brevity, as indicated by comments.

```
component Clock is {
   state {
      time : Integer;
      speed : Integer;
   }
   invariant {
      speed \eqgreater 0;
   }
   interface {
      prov ip1: Tick();
      prov ip2: setClockSpeed(rate : Integer);
   }
   operations {
      prov op1: {
         post ~time = time + 1;
      }
      prov op2: {
         let  r : Integer;
         post ~speed = r;
      }
   }
   map {
      ip1 -> op1 ();
      ip2 -> op2 (rate -> r);
   }
}
```

Figure 6-7. C2SADEL specification of the *Clock* component type.

6.3.2 Code Generation

DRADEL currently generates implementations of architectures on top of the Java implementation of our infrastructure. It generates class skeletons for all basic types. For example, the skeleton for Cargo Router's type `\set Shipment` is shown in Figure 6-9.

For each component type in the architecture, DRADEL generates an internal object skeleton and a dialog. For example, the generated internal object of the *Clock* component is shown in Figure 6-10, while its dialog is shown in Figure 6-11. Since it is at the top of the Cargo Router architecture, *Clock* does not respond to any notifications, nor does it generate requests. An example of requests generated from the *DeliveryPortArtist* component's specification is shown in Figure 6-12.

Given an architectural description, DRADEL also generates the "main" routine for the application. The "main" routine of the Cargo Router architecture is shown in Figure 6-13. Finally, DRADEL generates the "make" file that enables compilation of the generated skeleton. Although it provides no application-specific functionality, the skeleton can be executed once it is compiled: the implementation framework's scheduler will allot execution time to each component (stub) in the architecture.

```
component DeliveryPortArtist is subtype CargoRouteEntityArtist(int \and beh) {
   state {
      entity_name : String;            contents : \set String;
      vport_name : String;             vport_xpos : Integer;
      vport_ypos : Integer;            vport_fg : String;
      vport_bg : String;               width : Integer;
      height : Integer;                selection : Integer;
   }
   invariant {
      (width \eqgreater 0) \and (height \eqgreater 0) \and
      (vport_xpos \eqgreater 0) \and (vport_ypos \eqgreater 0);
   }
   interface {
      /*** several provided and required interface elements elided ***/
      prov ip1: InitVport(n : String; x : Integer; y : Integer; h : Integer;
                       w : Integer; fg : String; bg : String);
      prov ip6: SelectItem(entity : String; item : Integer);
      req ir4: Select(sel : Integer);
      req ir5: Deselect(sel : Number);
   }
   operations {
      /*** several provided and required operations elided ***/
      prov op1: {
         let  n : String; x : Integer; y : Integer; h : Integer;
              w : Integer; fg : String; bg : String;
         pre (x \eqgreater 0) \and (y \eqgreater 0) \and
            (h \eqgreater 0) \and (w \eqgreater 0);
         post (~vport_name = n) \and (~vport_xpos = x) \and
              (~vport_ypos = y) \and (~height = h) \and
              (~width = w) \and (~vport_fg = fg) \and (~vport_bg = bg);
      }
      prov op6: {
         let name : String; item_loc : Integer;
         pre (item_loc \eqless #contents) \and (name = entity_name);
         post (~selection = item_loc) \and (~entity_name = entity_name);
      }
      req or4: {
         let  num : Integer;
              selection : STATE_VARIABLE; cargo_size : STATE_VARIABLE;
         pre   num \eqless cargo_size;
         post ~selection = num;
      }
      req or5: {
         let num : Number; sel : STATE_VARIABLE;
         pre num = sel;
         post ~sel = -1;
      }
   }
   map {
      /*** several maps elided ***/
      ir4 -> or4 (sel -> num);
      ir5 -> or5 (sel -> num);
      ip1 -> op1 (n -> n, x -> x, y -> y, h -> h, w -> w, fg -> fg, bg -> bg);
      ip6 -> op6 (entity -> name, item -> item_loc);
   }
}
```

Figure 6-8. C2SADEL specification of the *DeliveryPortArtist* component type.
Several provided and required services (interface elements and their corresponding operations) have been elided for brevity. The two required operations correspond to the two *DeliveryPort* operations modeled in Figure 6-6.

```
package c2.PlannerSystem;
import java.lang.*;

public class Shipment_SET extends java.util.Vector {
   public Shipment_SET() { }

/*** Type-Specific Accessor Methods ***/

   public Shipment getElementAt(int i) {
      return (Shipment)super.elementAt(i);
   }

   public Shipment getFirstElement() {
      return (Shipment)super.firstElement();
   }

   public Shipment getLastElement() {
      return (Shipment)super.lastElement();
   }
}
```

Figure 6-9. A basic type skeleton generated by DRADEL.

```
package c2.PlannerSystem;
import java.lang.*;

public class Clock extends Object {

   // COMPONENT INVARIANT: speed \eqgreater 0.0

   private Integer time;
   private Integer speed;

   public Clock() {
      time = null; // or: new Integer(<init val>);
      speed = null; // or: new Integer(<init val>);
   }

/***** ADL Specified Methods *****/
   // PRECONDITION:
   public void setClockSpeed(Integer r) {
      /*** METHOD BODY ***/
   }
   // POSTCONDITION: ~speed = r

   // PRECONDITION:
   public void clockTick() {
      /*** METHOD BODY ***/
   }
   // POSTCONDITION: ~time = time + 1.0

/***** State Variable Access Methods *****/
   public void SET_time(Integer new_value) {
      time = new_value;
   }

   public Integer GET_time() {
      return time;
   }

   public void SET_speed(Integer new_value) {
   speed = new_value;
   }

   public Integer GET_speed() {
      return speed;
   }
}
```

Figure 6-10. Generated internal object class skeleton of Cargo Router's *Clock* component. Vertical spacing of the generated code has been altered to fit on the page.

```
package c2.PlannerSystem;
import c2.framework.*;
import java.lang.*;

public class Clock_C2_Component extends ComponentThread {

   private Clock state_var;

   public Clock_C2_Component(String name) {
      create(name);
   }

   public void create(String name) {
      super.create(name, FIFOPort.classType());
      recordMessageInterface();
      state_var = new Clock();
   }

/***** Notification Handling *****/
   public void handle(Notification notif_msg) { }

/***** Request Handling *****/
   public void handle(Request req_msg) {
      if (req_msg.name().equals(“setClockSpeed”)) {
         Integer rate = (Integer)req_msg.getParameter(“rate”);
         handleRequest_setClockSpeed(rate);
      }
      else if (req_msg.name().equals(“clockTick”)) {
         handleRequest_clockTick();
      }
   }

   private void handleRequest_setClockSpeed(Integer rate) {
      state_var.setClockSpeed(rate);
      notifysetClockSpeed(state_var.GET_speed());
      }

   private void handleRequest_clockTick() {
      state_var.clockTick();
      notifyclockTick(state_var.GET_time());
   }

/***** Notification Generating Methods *****/
   private void notifysetClockSpeed(Integer speed) {
      Notification notif_msg = new Notification(“setClockSpeedCompleted”);
      notif_msg.addParameter(“speed”, speed);
      send(notif_msg);
   }

   private void notifyclockTick(Integer time) {
      Notification notif_msg = new Notification(“clockTickCompleted”);
      notif_msg.addParameter(“time”, time);
      send(notif_msg);
   }

/***** Recording Interface <<< DO NOT MODIFY BELOW THIS LINE>>> *****/
   private void recordMessageInterface() {
      addMessageToInterface(“bottom”, “in”, “setClockSpeed”);
      addMessageToInterface(“bottom”, “in”, “clockTick”);
      addMessageToInterface(“bottom”, “out”, “setClockSpeedCompleted”);
      addMessageToInterface(“bottom”, “out”, “clockTickCompleted”);
   }
}
```

Figure 6-11. Generated dialog for the *Clock* component.
Vertical spacing of the generated code has been altered to fit on the page.

```
    private void requestDeselect(Integer sel) {
       Request req_msg = new Request("Deselect");
       req_msg.addParameter("sel", sel);
       send(req_msg);
    }

    private void requestSelect(Integer sel) {
       Request req_msg = new Request("Select");
       req_msg.addParameter("sel", sel);
       send(req_msg);
    }
```

Figure 6-12. A fragment of *DeliveryPortArtist*'s dialog depicts generated requests.

```
package c2.PlannerSystem;
import c2.framework.*;
import c2.comp.graphics.*;
import java.lang.*;

public class PlannerSystemArchitecture extends SimpleArchitecture {
   static public void main(String argv[]) {
      SimpleArchitecture PlannerSystem =
         new SimpleArchitecture("PlannerSystem");

      /*** several component declarations elided ***/
      Clock_C2_Component SimClock = new Clock_C2_Component("SimClock");
      DeliveryPort_C2_Component Runway =
         new DeliveryPort_C2_Component("Runway");
      DeliveryPortArtist_C2_Component RunwayArt =
         new DeliveryPortArtist_C2_Component("RunwayArt");

      /*** several connector declarations elided ***/
      FilteringConn RunwayConn = new FilteringConn("RunwayConn");
      FilteringConn UtilityConn = new FilteringConn("UtilityConn");

      /*** several component instantiations elided ***/
      PlannerSystem.addComponent(SimClock);
      PlannerSystem.addComponent(Runway);
      PlannerSystem.addComponent(RunwayArt);

      /*** several connector instantiations elided ***/
      PlannerSystem.addConnector(RunwayConn);
      PlannerSystem.addConnector(UtilityConn);

      /*** a part of the configuration specification elided ***/
      PlannerSystem.weld(Runway, RunwayConn);
      PlannerSystem.weld(RunwayConn, RunwayArt);
      PlannerSystem.weld(SimClock, UtilityConn);
      PlannerSystem.weld(UtilityConn, Runway);

      // SET UP CONNECTOR MESSAGE FILTERS
      /*** several connector message filters elided ***/
      RunwayConn.enableFiltering("message_filtering");
      UtilityConn.enableFiltering("message_filtering");

      PlannerSystem.start();
   }
}
```

Figure 6-13. "Main" routine for the Cargo Router system generated by DRADEL.
For brevity, only the declarations and instantiations of *DeliveryPort*, *DeliveryPortArtist*, and *Clock* components, and the relevant connectors are shown.

6.4 Evaluation

The examples discussed in the preceding sections of this chapter, as well as the C2 implementation infrastructure, discussed in Section 5.1, and the DRADEL environment, presented in Section 5.4, represent a body of work that, as a whole, validates the claims of this dissertation. Recall that the four main facets of our methodology are component evolution, connector evolution, architectural configuration evolution, and support for mapping an architecture to its implementation(s). We discuss below the manner and degree to which they are supported by our development infrastructure, its extensions (specifically, OTS middleware-integrated connectors), and example applications. This information is summarized in the matrix given in Table 6-7. Only the two representative applications—KLAX and Cargo Router—are shown in the table; DRADEL serves as both an example application and a development toolsuite.

Table 6-7: Coverage Matrix for the Different Aspects of Our Methodology

LEGEND:

☆ represents demonstration of a concept in an example
● represents implementation of or tool support for a concept
✪ represents both demonstration and implementation / tool support

*DRADEL is both a tool suite and an example application

		Development Infrastructure			OTS Middleware				Example Apps	
		C2 Class Framework	Graphics Binding	DRADEL*	Q	Polylith	RMI	ILU	KLAX	Cargo Router
Component Evolution	Heterogeneous Subtyping			●					☆	☆
	Type Checking			✪						☆
	Architect's Discretion			●						☆
Connector Evolution	Context-Reflective Interfaces	●		✪	●	●	●	●	☆	☆
	Information Filtering	●		●	●	●	●	●		☆
Configuration Evolution	Minimal Component Dependencies	●	☆	✪	●	●	●	●	☆	☆
	Heterogeneous Connectors	●		✪	☆	☆	☆	☆	☆	☆
	Application Family			✪					☆	☆
Implementation of Architectures	Implementation Infrastructure	●	●	✪	●	●	●	●	☆	☆
	Implementation Generation	●	●	●						☆
	Component Reuse		☆	✪					☆	☆
	Connector Reuse	●			☆	☆	☆	☆	☆	☆

6.4.1 Component Evolution

The specific technique we employ to support component evolution is heterogeneous subtyping of component specifications, i.e. architectural types. Treating components as types also enables analysis of architectures for type correctness. Finally, a key aspect of our approach is that, due to the separation of architecture from its implementation, the course of action, if any, in the case of a type mismatch is left to the architect's discretion. Together, these techniques present a substantive improvement over their existing alternatives.

6.4.1.1 Heterogeneous Subtyping

The motivation for heterogeneous subtyping in architectures arose from actual applications. Specifically, the variations of the KLAX architecture clearly demonstrate the need for flexible evolution of a component:

- Spelling KLAX, discussed in Chapter 2, contains examples of components that exported the same interface, but different behaviors (e.g., the *TileMatchLogic* and *SpellingLogic* components);
- Multi-lingual KLAX, discussed in Section 4.3, contains an example of a component (*TileArtist*) whose two different implementations (in C++ and Ada) have identical interfaces and behaviors;
- Finally, the integration of OTS constraint solvers, discussed in Section 6.1, represents examples of OTS components whose behaviors were preserved, but whose interfaces were changed to fit in an alternate domain of discourse. Similarly, C2's *Graphics-Binding* components are evolved OTS toolkits. The behaviors of the toolkits are preserved, while their interfaces are altered to enable their reuse across C2-style architectures.

These initial observations were formally elaborated in our architectural type theory, which then became the semantic basis of C2SADEL. Automated tool support for the type theory is provided in the DRADEL environment. The Cargo Router application was redesigned as another demonstration of the concepts of heterogeneous subtyping. Both DRADEL itself and architectures modeled, implemented, and evolved with its help can be type checked for correctness using DRADEL. DRADEL grants the architect discretion in dealing with type mismatches, as was demonstrated in the case of the Cargo Router application, discussed in Section 6.3.

The architectural type theory has enabled us to support component evolution in a systematic way. This approach results in added simplicity over traditional subtyping techniques, without sacrificing any of their power. The type theory allows us to identify the different directions in which a component evolves. By isolating and focusing on each direction individually, we divide the potentially complex subtyping relationships into a small set of simpler, sometimes trivial, relationships; by combining the different directions as needed, we achieve the power of standard, e.g., OOPL subtyping.

This approach to component evolution is also less error prone than manual component adaptation techniques. The relationship between a supertype component and its subtype is always explicitly specified; any expectations placed upon the subtype can be verified against the supertype and this relationship. By evolving components via subtyping, we can also define their substitutability criteria and avoid undesirable effects of ad-hoc evolution: component C_1 can *always* be used in the place of component C_2 if they export identical interfaces and behaviors or if C_1 is an *int* **and** *beh* subtype of C_2. At the same time, the flexibility of ad-hoc component adaptation is not sacrificed. The architect still has the discretion to allow two components to be interchanged in a given architecture even if they do not satisfy the substitutability criteria.

Furthermore, a component can always be evolved in *any* direction if the intent is to expand one's palette of design elements and use the component in a future architecture.

6.4.1.2 Type Checking an Architecture

Treating components as types also enables type checking (i.e., analysis) of architectures in the manner specified in Chapter 4. Our approach to architectural analysis introduces two novel concepts: architect's discretion (as with subtyping) and separation of provided from expected behavior. No other approach allows the architect to make the final decision regarding whether a potential error should be allowed to propagate into the implementation. We have demonstrated in the examples of partial and no utilization of a component's services in the KLAX architecture (Section 6.1) that, depending on the implementation infrastructure, certain architecture-level errors will have negligible effects on the resulting system. Forcing the architect to correct those errors may be more expensive than allowing him to acknowledge the errors, but proceed with system generation.

Several researchers have argued for the need to express a component's expectations of its environment; ADLs such as Wright and Rapide allow the specification of such information. However, in order to meaningfully express the expected external behavior, support for making reasonable guesses as to what that behavior may be is needed. This must be done in a manner that does not result in unrealistic assumptions or violated abstractions. The only other approach that explicitly models the behavior expected by a component *and* provides the means to describe that behavior in a generic way (à la our STATE_VARIABLE types) is CHAM [33]. Unlike CHAM, however, we use the type theory as a vehicle for evolution, as well as analysis.

Finally, by employing first-order logic, C2SADEL currently focuses on expressing static aspects of a component's behavior. This may be viewed as a disadvantage in comparison to CHAM, Rapide's posets, and Wright's CSP. However, our specific choice of formalism was guided by practicality and usability of the approach as our primary goals, and simplicity and understandability of architectural descriptions as a way of achieving these goals. The relevant aspects of the type theory, i.e., behavioral conformance, are entirely independent of the actual choice of formalism. Furthermore, our specification of component invariants presents the potential for modeling dynamic component behavior, should we choose to focus on it in the future.

6.4.2 Connector Evolution

Architecture-based software development approaches that lack explicit connectors require components to possess knowledge about their context and thus decrease the malleability of a system. At the same time, explicit connectors alone are not a guarantee of evolvability. If a connector is rigid in its expectations of attached components, its support for the evolution of the architecture is diminished. For example, a Wright connector explicitly specifies the number of components it can serve (via its roles) and the nature of the interaction with each component (via role protocols); adding or removing a component, or replacing an existing component with one that adheres to a different communication protocol requires replacing the connector itself as well. Connectors in this dissertation are inherently evolvable. Their evolution is enabled by context-reflective interfaces and different information filtering protocols.

6.4.2.1 Evolvable Interfaces

A connector's context-reflective interface directly enables addition, removal, replacement, and reconnection of components and other connectors. The implementation framework, OTS middleware-based connectors, and DRADEL all provide support for development with such connectors. Different variations of the KLAX and Cargo Router architectures demonstrate the usage of flexible connectors: all changes to an architecture are localized to the relevant connectors. Employing connectors that exhibit this degree of flexibility does not necessarily hamper the analyzability of an architecture. For example, an architecture changing at run-time can be analyzed for type correctness simply by taking a static "snapshot" at any point during its evolution.

6.4.2.2 Information Filtering

A connector also evolves by employing different information filtering mechanisms. The C2 implementation framework, and middleware-based connectors as its extensions, implement support for information (i.e., message) filtering. C2SADEL allows specification of different filtering mechanisms in a connector, while DRADEL's *CodeGenerator* ensures that the connector is instantiated properly in the implementation of the architecture. The Cargo Router architecture demonstrates usage of connectors with multiple filtering mechanisms, while KLAX and DRADEL employ broadcasting connectors only.

One of the benefits of message filtering is that it can improve the performance of an application. Message traffic can be computationally very expensive, especially in distributed applications; eliminating unnecessary traffic helps contain this cost. One of the reasons for commonly employing broadcast connectors in C2-style applications is to support run-time architecture evolution: unlike a broadcast connector, a connector that filters messages based on the current configuration may in fact filter out the messages needed by a component added in the future. The solution to this problem is to simply evolve the existing connector's filtering mechanism at run-time, rather than always allowing a potential flood of message traffic. However, the support for run-time connector evolution currently does not exist in the environment for dynamic manipulation of C2-style architectures, ArchStudio [61].

6.4.3 Configuration Evolution

Our support for evolving architectural configurations is based on minimizing component interdependencies, providing heterogeneous connectors, and application families that result from evolving configurations. While some of the resulting techniques have been explored by other researchers (e.g., implicit invocation, asynchronous communication, concurrency, message filtering), others are unique to our approach (substrate independence, connectors with context-reflective interfaces, transparency of connector implementation mechanisms). As a whole, these techniques provide a novel approach for evolving architectures both at specification- and run-time.

6.4.3.1 Minimal Component Interdependencies

Implicit invocation has been employed by other systems quite extensively for its benefits in separating modules. However, the C2 style extends this by providing a discipline for ordering components which use implicit invocation, yielding substrate independence. The two example applications we discussed, KLAX and Cargo Router, demonstrate the benefits of asynchronous, implicit invocation and substrate independence: KLAX was used as a basis of a large application family, while Cargo Router was easily extended to support additional views of application state and automatic planning; these modifications had minimal effect on other components in the respective architectures. In both applications, the evolution can be performed at specification-time, or while the application is executing.[11]

Without asynchronous, implicit invocation, the evolution of a configuration would require modification of surrounding components. Component addition would require changes to existing components in order to enable communication with the new component, since that communication is explicit. Component removal, on the other hand, would result in compile-time or run-time errors, e.g., when a method whose definition has been removed is invoked. Even if implicit invocation were employed, such a call could block indefinitely, unless the interaction was asynchronous. The effects of component addition and removal are further lessened in our methodology by explicitly restricting dependencies among those components via the substrate independence principle.

The C2 implementation framework and the middleware-integrated C2 connectors provide support for achieving asynchronous, message-based communication. The DRADEL environment supports the construction of applications characterized by minimal component interdependencies, via its *CodeGenerator* component. DRADEL also demonstrates this property, in that it is constructed according to C2 style rules.

6.4.3.2 Heterogeneous Connectors

Component interdependencies are further minimized by employing explicit connectors as interaction intermediaries. In addition to connector evolution properties discussed in Section 6.4.2, connectors aid configuration evolution by supporting different types of interaction and degrees of concurrency. The C2 implementation framework and DRADEL provide support for heterogeneous connectors, enabling architects and developers to adequately address the specific application requirements (e.g., performance, distribution, interoperability with legacy components, and so forth). The different middleware-based connectors, available as part of the framework, are a demonstration of heterogeneous connector implementations. They support the message passing and RPC types of interaction. DRADEL currently enables development with single- or inter-thread message-passing connectors that employ different filtering policies. DRADEL can be easily extended to support any type of interaction or degree of concurrency provided by the connectors available in the framework. Different variations of the KLAX, Cargo Router, and DRADEL architectures demonstrate the use of heterogeneous connectors.

6.4.3.3 Application Families

Finally, a direct by-product of evolution is creation of application families. Different variations of a given architecture often represent members of the same application family. Such a family can be formed in our approach by providing multiple variations of a set of components (KLAX and DRADEL), adding or removing functionality to an existing architecture (KLAX and

11. C2's *GraphicsBinding* components also represent examples of implicit, asynchronous invocation. Note that, although *GraphicsBinding* is typically at the bottom of a C2 architecture and, as such, should have knowledge of all components above it in the architecture, reusing it across applications mandates that it export a standardized interface. This issue was discussed more extensively in Chapter 2.

Cargo Router), or interchanging connectors with different implementations (KLAX, Cargo Router, and DRADEL). Due to its reflexive nature, DRADEL is, again, both an enabler and an example of this property.

6.4.4 Implementation of Architectures

The different facets of this dissertation's support for transitioning architecture-level decisions and properties into implementations are

- an implementation infrastructure that provides basic component and connector interoperation services,
- reuse of existing components and connectors that provide the desired functionality, and
- automatic generation of an implementation from an architecture, aided by the infrastructure and OTS components and connectors.

6.4.4.1 Implementation Infrastructure

The C2 class framework and GUI toolkit bindings are the basic elements of our implementation infrastructure. The framework is extensible, e.g., by integrating middleware technologies. DRADEL supports mapping of architectures to the class framework and utilizes the toolkit bindings in implementing architectures; it is thus also a part of the implementation infrastructure. KLAX, Cargo Router, and DRADEL represent examples of architectures that were implemented using the class framework and *GraphicsBinding* components. A variation of the Cargo Router application was partially generated by DRADEL.

The infrastructure greatly simplifies the construction of C2-style applications, since it eliminates from developers the responsibility of implementing the C2 concepts for each application. Furthermore, the infrastructure forms a platform that bounds the implementation space and directly enables automated code generation.

6.4.4.2 Component Reuse

Support for OTS component reuse is provided in DRADEL. DRADEL enables reuse and evolution of existing functionality through subtyping. Furthermore, its *CodeGenerator* separates the dialog of a component from its (reusable) internal object. The most ubiquitous example of component reuse in C2 style architectures are the *GraphicsBinding* components: they are used in any C2 application that has a GUI front end, including KLAX, Cargo Router, and DRADEL. Additionally, KLAX contains examples of reusing OTS UI constraint solvers. Other examples of OTS component reuse are discussed in [48], [50], [61].

6.4.4.3 Connector Reuse

Reuse of OTS connectors is an alternative to expending resources to implement the support for concurrency, distribution, and different types of interaction, already in existence. The four middleware-based connectors represent examples of OTS connector reuse. Several variations of the KLAX and Cargo Router architectures demonstrate reuse of (middleware-based) OTS connectors. The implementation for a generic, single-thread, message broadcasting connector in the implementation framework has been used as a basis of integrating OTS middleware (see Figures 4-16 on page 64 and 5-3 on page 79).

6.4.4.4 Implementation Generation

DRADEL supports systematic evolution and implementation generation of architectures. We had demonstrated many of the facets and benefits of our approach to implementing an architecture *prior* to DRADEL's development. Our support consisted of the implementation infrastructure and reused OTS components and connectors; we (manually) implemented a number of applications and tools using the infrastructure. One advantage of using DRADEL is its ability to automatically generate application skeletons and/or provide wrappers for OTS components. This can be a sizable task that also removes any chance of inadvertent interface and behavior mismatches and focuses debugging and testing efforts on functionality internal to a component.

A more quantitative metric that can be used to assess the benefit of automatic application generation is the ratio of the amount of generated code to the size of the completed implementation. For example, the combined sizes of component dialogs in an application are indicative of the overall amount of component interactions: construction of outgoing messages, accessing information from incoming messages, and invocation of internal object methods in response to incoming messages. The dialogs can represent a sizable fraction of a C2-style application; being able to generate this code alone adds substantial value to DRADEL. In the implementation of single-process KLAX, 47% (1,700 source lines of code, or SLOC) of the application handles component interactions; in the Cargo Router it is 53% (1,500 SLOC). In DRADEL itself, the percentage is lower, 14% (1,850 SLOC). Each component in DRADEL is a tool that contains more complex internal functionality, while the flow of messages among the tools is comparatively lighter.

This metric does not account for OTS component reuse. For example, only 9% (170 SLOC) of KLAX's *LayoutManager* implemented with SkyBlue (*LayoutManager*-2 in Table 6-1 on page 94) handles message traffic. However, in this case the component's internal object (SkyBlue) already exists; automatic generation of the dialog only completes the component.

6.4.4.5 Discussion

Most of the facets of our support for implementing architectures have been explored quite extensively by other researchers and their benefits are well understood. Component interoperability models, e.g., JavaBeans, provide underlying support that is similar to our implementation infrastructure. Similarly, DSSA approaches focus on specific implementation platforms. Software engineering researchers and practitioners are continuously investigating techniques for automatically generating implementations from specification and design artifacts. Also, the arguments for the potential advantages of reusing existing functionality over reimplementing it have been embraced by the community [8], [10], [24].

Our approach represents an improvement over existing work in that it addresses the issues of providing an implementation infrastructure, OTS reuse (including the reuse of connectors), and code generation in tandem and does so at the architectural level. With the partial exception of DSSA, existing architecture work has largely neglected to provide techniques for transferring architecture-level decisions to the implementation (see Chapter 3). As a result, the support for OTS component reuse is sparse. Additionally, no existing research has addressed the reuse of OTS connectors as a means of aiding interoperability and evolution. Solely focusing on reusing components limits the effectiveness of a software development approach.

CHAPTER 7: Future Work

This dissertation has made a significant contribution to the body of work in software architectures and, in particular, to architecture-based evolution of software systems. Our focus spans individual software components, connectors, and entire architectures. We have provided a collection of techniques that, individually and in concert, enable and support architecture-based evolution at specification-time. Several issues still remain unresolved, however. This chapter considers the open research questions and discusses our plans for future work.

This research will evolve in several different directions. We intend to expand the existing methodology to provide additional architecture-based evolution support. We will also attempt to apply the methodology to other application domains, architectural styles, ADLs, and implementation platforms. The scope of certain aspects of our work to date has been deliberately limited. For example, we have focused on two application domains: GUI-intensive systems, as the primary domain, and software development tool suites. Such decisions enabled us to identify and investigate the important properties of our methodology. Focusing on a well-defined (subset of the) problem also allowed us to establish the utility of the methodology.

This strategy has resulted in several natural outgrowths of this work: integration of our support for specification-time evolution with on-going work in supporting execution-time evolution of C2-style architectures [61], extensions to the type theory, incorporation of additional kinds of connectors, expansion of our treatment of and support for architectural refinement, and investigation of new techniques for supporting OTS reuse. Each of them may, in turn, require modifications to C2SADEL, DRADEL, or our implementation infrastructure.

7.1 Integrated Evolution Support

A task we will address in the most immediate future is integrating this dissertation's support for evolution of architectures at specification-time with the existing support for their evolution at run-time. Architecture-based run-time software evolution has been an important aspect of the work conducted in the C2 research group [61]. Current run-time support includes the ability to add, remove, replace, and reconnect components, and to ensure topological constraints while the application is executing. An environment, called ArchStudio, provides a graphical editor for manipulating architectures, a WWW browser for locating OTS components, and an agent for downloading remote components, determining their intended location in an architecture, and inserting them into the architecture.

ArchStudio's support for run-time evolution is based on manipulating C2-style architectures implemented using the infrastructure described in Section 5.1. Therefore, there is a natural connection between the research of this dissertation and ArchStudio: the output of the specification-time architecture-based development "phase" becomes the input to the run-time "phase." We believe that the specification-time support can be employed at run-time to evolve an existing component, model and generate a new component, or analyze the application's architecture for type conformance at any point during execution. ArchStudio can also employ any of the connectors discussed in this dissertation to support addition of multi-lingual components or distribution of the application. Our forthcoming work will investigate these issues and extend ArchStudio and DRADEL to enable their interactions as discussed above.

7.2 Type Theory

The intent behind the type theory is to identify and model important aspects of architectural types. The larger and potentially more complex problem of evolving an entire type can thereby be simplified by addressing only the type's properties of interest. Thus far, we have successfully applied this method to evolving components. In the future, we intend to apply this area of our work to other architectural constructs—connectors and configurations—as well as to other ADLs. We also intend to investigate possible extensions to the type theory to support modeling and evolution of additional aspects of architectural types. These issues are discussed below. Finally, we will work on applying the type theory to support the automatic generation of component adaptors, i.e., domain translators, discussed in Chapter 2.

7.2.1 Applying the Type Theory to Connectors

Though their roles are quite different, software connectors are in certain regards similar to components. Some aspects of the type theory may thus be used to complement our current support for connector evolution. Like components, connectors are *name*d and have an *implementation*. Furthermore, a connector's *behavior* is reflected in its information filtering and transaction mechanism. These three aspects of a connector can be modeled in the type theory and used to support connector evolution. This would also allow us to specify more precisely the conditions under which connectors using different filtering mechanisms may be interchanged in an architecture.[1]

The remaining feature modeled in the type theory, *interface*, is fundamentally different between components and connectors. Unlike components, connectors do not export a particular interface; instead, their interfaces are context-reflective. Therefore, interface subtyping as defined in Section 4.1 cannot be applied to connectors. This difference is not crucial, however, since connector interfaces in this dissertation have been specifically designed to be evolvable.

7.2.2 Applying the Type Theory to Architectural Configurations

The method for evolving architectural configurations discussed in Section 4.3 exploited the properties of individual components (e.g., substrate independence) and connectors (e.g., degree of concurrency). Although a configuration is a top-level architectural constituent, no specific techniques were provided by the type theory to support its evolution in a systematic manner; no specific constructs currently exist in C2SADEL to model that evolution. Applying the type theory to configurations and extending C2SADEL to include configuration evolution information at architecture specification-time would complement our current approach and remedy some of its shortcomings. For example, it would enable us to explicitly represent and evolve application families.

The concept that directly facilitates the application of the type theory to architectural configurations is hierarchical composition. As discussed in Section 4.4.4, we allow an entire architecture to be used as a single component in another architecture. This is also indicated in Figure 5-1 on page 72: the *Architecture* class is a subclass of the *Component* class in the implementation framework. As such, the architecture will have a *name* and export an *interface* and *behavior*.[2] The architecture will also (possibly) have an *implementation*.

1. Clearly, we expect the conformance rules for connector subtyping to be different from those specified for components in Section 4.1.

The configuration thus becomes a *composite* architectural type. The internal architecture of a composite type (see Figure 4-17 on page 69) is different from that of a *simple* architectural type (see Figure 2-2 on page 6). Therefore, evolving an aspect of the architecture will differ from evolving a simple component and may be accomplished by evolving one or more of the architecture's constituent components. For example, evolving the interface exported by the configuration may require (partially) changing the interfaces of its externally accessible components. Alternatively, an existing component could be retrieved from a repository and substituted for a component in the configuration to satisfy the needed interface. This is an unexplored area and we intend to investigate it further.

7.2.3 Applying the Type Theory to Additional ADLs

The structural features of architectures we model in the type theory are common across ADLs [51], giving us confidence that the type theory can be applied to other ADLs. Moreover, the constructs introduced specifically to support this dissertation—the internal component architecture, topological rules imposed by the C2 style, explicit modeling of connectors, separation of interface from behavior, and modeling semantics in first-order logic—are entirely independent of type theory's formal underpinnings. We employ the internal component architecture only when mapping an architectural description to the implementation in DRADEL. The rules of topological composition do differ across ADLs; however, as discussed in Section 4.1, the type theory does not impose particular rules, only a requirement that they be explicitly specified. Finally, certain ADLs, e.g., Rapide and Darwin, do not model connectors as top-level constructs. The connectors in our type system were used only to determine communication pathways; those pathways are already modeled explicitly in such ADLs.

Certain details of the type theory will have to be modified for use with a given ADL. For example, we separate a component's interface from its behavior; no other ADL does so. Furthermore, our definition of type conformance is based on modeling component semantics via invariants and operation pre- and postconditions expressed as first-order logic expressions. Other ADLs use different formalisms (e.g., Rapide uses posets, Darwin uses π calculus, and Wright uses CSP). In order to apply the type theory to those ADLs, conformance rules specific to their underlying formalisms must be specified. This is not an unreasonable requirement: notions of type conformance have already been established in the case of Rapide and Wright, for example.

7.2.4 Evolving the Type Theory

The aspects of architectural types (components) represented thus far have been name, interface, behavior, and implementation. As discussed in Chapter 4, this particular taxonomy was motivated by OO type theories. It has proven useful and elegant in modeling, analyzing, and evolving architectures. However, this set of concerns on which the type theory currently focuses is very limited. The issues of importance in a given architecture may also be throughput, reliability, security, schedulability, performance, and even non-technical, project-related concerns, such as budgets and deadlines. It is our hypothesis that an approach similar to our current method can be employed to model and evolve such aspects of architectures. As part of our future work, we will investigate which additional properties of architectures can be used to expand the type theory, develop new or adapt existing techniques for their modeling, establish conformance criteria for

2. Recall from Section 4.4.4 that the architecture's interface and behavior are functions of its component and connector interfaces and behaviors, respectively.

their evolution and analysis, analogous to those defined in Section 4.1, and provide appropriate tool support.

7.3 Connectors

The connectors introduced in this dissertation are uniquely suited to support architecture-based evolution, via their context-reflective interfaces. We believe the connectors to be applicable beyond C2. At the same time, our current support is essentially restricted to asynchronous message passing connectors, limiting their applicability. As part of our future work, we intend to expand the range of supported connectors. To this end, we will attempt to leverage connectors provided by other architectural approaches. We also plan to further examine the utility of our unique brand of connectors by applying it to other architectural approaches.

7.3.1 Expanding Our Support for Connectors

Section 4.3 discussed the variability of connectors based on types of interaction and degrees of concurrency and information exchange. Connectors modeled and implemented as part of this dissertation have provided extensive support for different degrees of concurrency and information exchange. However, for the most part we have only supported asynchronous, message passing connectors. The exceptions are the OTS middleware-integrated connectors that support RPC. Even in those cases, the communication is asynchronous; our approach was essentially to simulate message passing with RPC.

Asynchronous, message-based communication has proven very beneficial in the context of C2 work, enabling distribution, multi-lingual application development, specification- and execution-time evolution, and so forth. On the other hand, in certain situations, message-based communication, particularly intra-process, may be too inefficient. Also, some components may not be able to communicate via messages and may assume that they will only engage in synchronous communication. Thus far, our solution has been to build message wrappers for such components (see Section 5.2). An alternative is to employ connectors that more naturally support a component's assumed type of interaction. Other ADLs, e.g., UniCon [82], model and implement additional connector types. We will attempt to expand our support to include external connectors. An issue we will need to address in the process of doing so is how to reconcile the different connector modeling formalisms, and specifically how to modify third-party connectors to implement context-reflective interfaces.

7.3.2 Applying Our Connectors to Other ADLs

We also believe that other architectural approaches can benefit from the connectors employed in this dissertation. The utility of our connectors in facilitating execution-time evolution of architectures has been recognized by other researchers [36]. The connectors may be used as a complement to the evolution support already existing in ADLs that do not explicitly model connectors, e.g., Darwin and Rapide. Other ADLs, such as Wright and UniCon, use connectors with a more rigid structure. These ADLs introduce some flexibility, similar to our context-reflective interfaces, via connector interface parameterization. However, they deliberately limit the flexibility of connectors to maximize the analyzability of architectures. This dissertation has shown that flexibility can be maximized without sacrificing analyzability of C2SADEL architectures. We intend to further test this hypothesis by applying our connectors to other ADLs.

7.4 Refinement

This dissertation’s support for implementing an architecture rests on the assumption that the mapping between an architecture-level component or connector and an implemented module will be 1-to-1. Any deviation from this bijective relationship is masked by the internal component architecture, where a component’s internal object may contain multiple modules, but the component still provides a single dialog. Although this approach has proven successful, the relationship between architectural and implementation modules is likely to be more complex in general, so that, for example, connectors are not explicit in the implementation, but are fragmented and distributed across components.

7.4.1 Refining Architectures across Levels of Abstraction

Moriconi and colleagues [57] provide an approach for refining an architecture across several levels of abstraction, where a component at a given level may be represented by multiple components at a subsequent lower level. One of the shortcoming of their technique is that it does not eventually refine a low-level “architecture” into an implementation. However, their basic idea is cogent and we intend to investigate refinement techniques that may be used in our research.

Our adoption of such a technique will introduce some unique problems. Refinement will result in an abstraction hierarchy of architectural elements (components, connectors, and configurations) that is largely orthogonal to their type hierarchy. A related issue is the existence and maintenance of multiple repositories of architectural elements corresponding to the different levels of abstraction. We will also have to solve the problem of correct maintenance of type information and subtyping relationships across levels of refinement. Finally, we intend to investigate how the evolution of a component at a given level of refinement (i.e., abstraction) affects components in its preceding and subsequent refinements (i.e., above and below it in the abstraction hierarchy).

7.4.2 Ensuring Architectural Properties in the Implementation

Another facet of our approach to implementing an architecture is ensuring architectural properties in the implementation. Currently, this is accomplished by providing *guidance* to the developers in the form of comments that contain operation pre- and postconditions, as discussed in Section 5.4. However, there is no *guarantee* that a developer will properly implement an operation. This can be remedied by promoting pre- and postcondition comments to assertions and employing an assertion checking technique to ensure that they are satisfied during execution [75]. We will investigate the appropriate techniques for doing so.

A related issue is the modeling of basic types. Since architectures are intended to deal with high-level interconnections and protocols, rather than low-level data structures [83], we have decided not to model the properties of basic types. Section 5.4 discussed our *TypeChecker* component’s resulting pessimistic inaccuracy. Another potential problem is the lack of assurance that developers will correctly implement the basic types. Since an architectural refinement will eventually need to represent such low-level constructs, we intend to expand our support for basic types. We will investigate existing theorem provers and model checkers that support basic types as possible complements to the *TypeChecker*: NORA/HAMMR [79], Larch proof assistant (LP) [30], VCR [18], and PVS [64].

All three issues discussed in this section—architectural refinement, assertion checking, and modeling of basic types—will also result in modifications to our support for implementation generation, specifically to DRADEL's *CodeGenerator* component.

7.5 Reuse

While this dissertation provides a method for incorporating OTS components into an architecture in a systematic manner, the approach to *locating* those components is largely ad-hoc. Our component repository (see Section 5.4) is currently very simple. This simplicity came at the expense of the repository's support for reuse-driven, architecture-based development. The repository only accepts components modeled in C2SADEL and represented as ASCII files. Additional functionality is needed to link the architectural description of a component with the component's implementation(s). Moreover, the repository must be evolved to include support for automated browsing and retrieval. We intend to use the type theory to develop this support in the manner demonstrated by similar approaches [18], [99].

Our ultimate goal is to expand the kinds of components accessible through the repository. One aspect of this task is being addressed by our work on integration with middleware platforms, which will enable us to use middleware-compliant components. To enable such heterogeneous components to interoperate in a single application, we must bridge across implementation platforms, e.g., our class framework, CORBA [62], and JavaBeans [31]. Several of the concepts embodied in the DRADEL environment make it a promising candidate for accomplishing this task. We intend to extract a more general, "reference" architecture for environments for architecture-based development and evolution and, in particular, for transferring architectural decisions to the implementation. We plan to exploit DRADEL's component-based nature and evolvability to incrementally extend our support to multiple implementation platforms.

Another direction in which our support for reuse will evolve is the planned development of a "virtual" repository that would include facilities for locating and retrieving components remotely, e.g., over the Internet. Such components will likely be represented in standard (e.g., UML) or proprietary modeling languages. In order to expand our support to multiple classes of OTS components, we will need to address heterogeneous formalisms and relate them to C2SADEL's formalism. We intend to study the issues in applying the type theory to evolve components represented in heterogeneous formalisms, automating the evolution of existing components to populate partial architectures, and analyzing architectures composed of heterogeneous components. This is a long-term task that will be aided by our work on applying the type theory to additional ADLs, discussed above in Section 7.2.3.

CHAPTER 8: Conclusions

Software architectures show great promise for reducing development costs while improving the quality of the resulting software. Architecture addresses an essential difficulty in software engineering—complexity—via abstraction and its promise of supporting reuse. In a few short years, software architecture research has produced credible, if not impressive, results. At the same time, our in-depth survey of architecture research [46], [51], summarized in Section 3.9, indicates a number of areas in which the current support is insufficient or lacking altogether. One such area is evolution.

Architectures provide a fertile basis for supporting software evolution. However, architecture is not a “silver bullet” [12], and simply introducing it into an existing development lifecycle will not fulfill its potential. Improved evolvability cannot be achieved solely by explicitly focusing on architectures, just like a new programming language cannot by itself solve the problems of software engineering. A programming language is only a tool that allows (but does not force) developers to put sound software engineering techniques into practice. Similarly, one can think of software architectures as tools which also must be accompanied with specific techniques to achieve desired properties.

This dissertation has discussed a set of techniques and tools for supporting specification-time evolution of software architectures in a manner that preserves the desired architectural relationships and properties. The combination of the tools and techniques comprises a comprehensive methodology: it supports the evolution of individual architectural building blocks—components and connectors—as well as their configurations in an architecture; it also supports the systematic mapping of architectures to implementations.

One overarching characteristic of our methodology is *flexibility*. It has resulted from the recognition that evolvability both implies and requires flexibility. Architectures are at a high level of abstraction, where numerous decisions are deliberately delayed or only partially made. An architecture should reflect the interests and requirements of numerous stakeholders, including users, customers, managers, and developers. For these reasons, unlike, e.g., programming languages, software architectures need not always be rigid in establishing properties such as correctness, consistency, and completeness. Our objective has been to introduce flexibility in all the facets of architecture-centered development addressed by this dissertation: components, connectors, and configurations, as well as tool support for their modeling, analysis, implementation, and evolution.

Flexibility in Components. The components in this dissertation separate their interface (dialog) from behavior (internal object). This separation allows every component’s operation to export multiple interfaces if needed, the interface and behavior to be evolved (or even entirely replaced) independently of each other, and OTS components to be easily integrated. The architectural type theory allows systematic evolution of components in a number of directions, by preserving a component’s name, interface, behavior, implementation, or one of their combinations; this is not the case with other existing type theories. The rules for substituting components in an architecture are also flexible, as reflected in the notions of partial communication and partial component service utilization.

Flexibility in Connectors. The flexibility of connectors introduced in this dissertation is manifested in their context-reflective interfaces. Another facet of flexibility is a connector's ability to support different information filtering mechanisms, allowing it to adapt and thus be usable in numerous situations. The connectors of this dissertation give architects and developers an added degree of flexibility in composing a system by encapsulating their support for different degrees of concurrency and different implementation/middleware platforms. The appropriate connector can be chosen, and later replaced, without having to make any modifications to other parts of an application.

Flexibility in Configurations. This dissertation's components and connectors employ asynchronous, implicit invocation via messages, greatly simplifying their composition into an architecture. Message-based communication is coupled with topological constraints introduced by the C2 style, the notion of substrate independence, and the flexibility of connectors, to minimize the effects of modifying a configuration: architectural elements can be easily added, replaced, removed, or reconnected. Components that share a communication link in a configuration may not always be able to communicate, e.g., because of an interface mismatch (partial communication). This can be detected via type checking and prevented. However, even if such a configuration is allowed to propagate into the implemented system, it will result in the loss of communication messages, but still allow the rest of the system's architecture to perform at least in a degraded mode. Thus, informing the architect of the potential problem and leaving the decision up to the architect (architect's discretion) is often preferable to automatically rejecting the configuration. Finally, configuration flexibility is also enhanced by hierarchical composition: an entire configuration can become a single component in a larger configuration if needed.

Flexibility in Tool Support. This dissertation's implementation infrastructure and DRADEL environment have been constructed to be flexible. The implementation framework is simple, consisting of relatively few classes, and has been designed to be adopted by developers with minimal installation, understanding, or usage costs. The framework can be extended as desired to include support for additional kinds of components (e.g., artists) and connectors (e.g., new middleware). This is accomplished simply by using the subclassing mechanisms provided by the underlying programming languages. At any point, any subset of the framework (e.g., only single-thread components and connectors) can be used. The *GraphicsBinding* components exploit the internal architecture of C2 components to incrementally provide message-based access to GUI toolkits. A simple binding to a new toolkit can be generated quickly and inexpensively; the binding can then be extended as needed. Finally, the flexibility of the DRADEL environment is embodied in its reflexive, component-based nature. DRADEL can evolve itself using the methodology of this dissertation to support additional ADLs, repositories, analyses, evolution techniques, and implementation platforms, as well as their different configurations. DRADEL also adds flexibility to the "architecting" process by implementing the concept of architect's discretion.

The architecture-based evolution methodology of this dissertation is unquestionably its biggest contribution. However, we feel that our quest for flexibility is, in many regards, just as important, and certainly as unique when compared to existing architecture research. The early notions of architecture, as articulated by, e.g., Perry and Wolf [70], treated architecture as a bridge between software requirements and designs, i.e., customers and developers. As such, an architecture needs to provide a balance between formality and accessibility. It also needs to be flexible enough to easily incorporate the changes imposed by the stakeholders as their collective understanding of a system evolves.

Much of Perry and Wolf's vision has been lost over time and the existing architecture research efforts have been characterized by an increasing focus on formality. While well suited for architecture-based analysis, formality can render an architecture too rigid to be evolvable. This dissertation has demonstrated that a balance is possible, where formality and analyzability are coupled with specific techniques to increase flexibility and enable the evolution of architectures. The coupling of formality and evolvability has a great potential to fulfill the promise of software architectures stated in the first sentence of this chapter. Our future work will further explore and exploit this potential.

REFERENCES

1. M. Abadi and K. R. M. Leino. A Logic of Object-Oriented Programs. Digital Equipment Corporation, Systems Research Center Technical Report 161, September 1998.
2. R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pp. 71-80, Sorrento, Italy, May 1994.
3. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213-249, July 1997.
4. P. America. Designing an Object-Oriented Programming Language with Behavioral Subtyping. *Lecture Notes in Computer Science*, vol. 489, Springer-Verlag, 1991.
5. D. Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE Avionics Reference Architecture. In *Proceedings of AIAA Computing in Aerospace 10*, San Antonio, 1995.
6. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, pp. 355–398, October 1992.
7. T. J. Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. *IEEE International Conference on Software Reuse*, November 1994.
8. T. J. Biggerstaff and A. J. Perlis. *Software Reusability*, vol. I and II. ACM Press/Addison Wesley, 1989.
9. P. Binns, M. Engelhart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation, and Control. *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 2, 1996.
10. B. W. Boehm and W. L. Scherlis. Megaprogramming. In *Proceedings of the Software Technology Conference 1992, pp. 63-82*, Los Angeles, April 1992.
11. K. Brockschmidt. *Inside OLE 2*. Microsoft Press, 1994.
12. F. P. Brooks, Jr. Essence and Accidents of Software Engineering. *IEEE Computer*, vol. 20, no. 7, pp. 10-19, April 1987.
13. M. R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, vol. 1, no. 3, pp. 36–47, June 1990.
14. P. Chan and R. Lee. *The Java Class Libraries: An Annotated Reference.* Addison-Wesley, 1996.
15. D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, Redmond, WA, 1996.
16. K. K. Dhara and G. T. Leavens. Forcing Behavioral Subtyping through Specification Inheritance. Technical Report, TR# 95-20c, Department of Computer Science, Iowa State University, August 1995, revised March 1997.
17. E. Di Nitto and D. S. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. To appear in *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999.

18. B. Fischer, M. Kievernagel, and W. Struckmann. VCR: A VDM-Based Software Component Retrieval Tool. Technical Report 94-08, Technical University of Braunschweig, Germany, November 1994.

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

20. D. Garlan, editor. *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, April 1995.

21. D. Garlan. What is Style? In *Proceedings of the First International Workshop on Architectures for Software Systems,* pp. 96-100, April 1995.

22. D. Garlan. An Introduction to the Aesop System. July 1995. http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps

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

24. D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, vol. 12, no. 6, pp. 17-26, November 1995.

25. D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.

26. D. Garlan and D. Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. In *Proceedings of VDM'91: Formal Software Development Methods*, pp. 31-44, Noordwijkerhout, The Netherlands, October 1991.

27. D. Garlan, F. N. Paulisch, and W. F. Tichy, editors. *Summary of the Dagstuhl Workshop on Software Architecture*, February 1995. Reprinted in *ACM Software Engineering Notes,* vol. 20, no. 3, pp. 63-83, July 1995.

28. C. Ghezzi, M. Jazayeri, D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

29. M. M. Gorlick and R. R. Razouk. Using Weaves for Software Construction and Analysis. In *Proceedings of the 13th International Conference on Software Engineering (ICSE13)*, pp. 23-34, Austin, TX, May 1991.

30. J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

31. G. Hamilton, editor. JavaBeans API Specification, version 1.01. Sun Microsystems, July 1997.

32. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

33. P. Inverardi, A. L. Wolf, and D. Yankelevich. Checking Assumptions in Component Dynamics at the Architectural Level. In *Proceedings of the Second International Conference on Coordination Models and Languages (COORD '97)*, Berlin, Germany, September 1997.

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

35. R. Kadia (pen name for the authors involved). Issues Encountered in Building a Flexible Software Development Environment. In *Proceedings of the Fifth Symposium on Software Development Environments (SIGSOFT'92)*, pp. 169-180, Reston, VA, December 1992.

36. J. Kramer and J. Magee. Analysing Dynamic Change in Software Architectures: A Case Study. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pp. 91-100, Annapolis, MD, May 1998.

37. G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, August/September 1988.

38. C. W. Krueger. Software Reuse. *Computing Surveys*, vol. 24, no. 2, pp. 131-184, June 1992.

39. G. T. Leavens. Verifying Object-Oriented Programs that Use Subtypes. PhD thesis, MIT Laboratory for Computer Science, February 1989. Available as Technical Report MIT/LCS/TR-439.

40. B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811-1841, November 1994.

41. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336-355, April 1995.

42. D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717-734, September 1995.

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

44. M. J. Maybee, D. H. Heimbigner, and L. J. Osterweil. Multilanguage Interoperability in Distributed Systems: Experience Report. In *Proceedings of the Eighteenth International Conference on Software Engineering*, Berlin, Germany, March 1996.

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

46. N. Medvidovic. A Classification and Comparison Framework for Software Architecture Description Languages. Technical Report, UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, February 1997.

47. N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pp. 24-32, San Francisco, CA, October 1996.

48. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, pp.

190-198, Boston, MA, May 1997. Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, pp. 692-700, Boston, MA, May 1997.

49. N. Medvidovic and D. S. Rosenblum. Domains of Concern in Software Architectures and Architecture Description Languages. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pp. 199-212, Santa Barbara, CA, October 1997.

50. N. Medvidovic and R. N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, vol. 144, no. 5-6, pp. 237-248, October-December 1997.

51. N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 60-76, Zurich, Switzerland, September 1997.

52. N. Medvidovic, R. N. Taylor, and D. S. Rosenblum. An Architecture-Based Approach to Software Evolution. In *Proceedings of the International Workshop on the Principles of Software Evolution*, Kyoto, Japan, April 20-21, 1998.

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

54. R. Milner, J. Parrow, and D. Walker. *A Calculus of Mobile Processes, Parts I and II*. vol. 100 of *Journal of Information and Computation*, pp. 1-40 and 41-77, 1992.

55. R. Monroe. Armani Language Reference Manual, version 0.1. Private communication, March 1998.

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

57. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 356-372, April 1995.

58. R. Natarajan and D. S. Rosenblum. Extending Component Interoperability Standards to Support Architecture-Based Development. Technical Report, UCI-ICS-98-43, Department of Information and Computer Science, University of California, Irvine, December 1998.

59. K. Ng, J. Kramer, and J. Magee. Automated Support for the Design of Distributed Software Architectures. *Journal of Automated Software Engineering (JASE), Special Issue on CASE-95*, vol. 3, no. 3-4, pp. 261-284, 1996.

60. O. Nierstrasz. Regular Types for Active Objects. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'93)*, pp. 1-15, Washington, D.C., USA, October 1993.

61. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pp. 177-186, Kyoto, Japan, April 1998.

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

63. OVUM. OVUM Evaluates Middleware. Technical Report, OVUM Ltd., 1996.

64. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. *PVS:* Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, eds., *Computer-Aided Verification (CAV '96)*, vol. 1102 of Lecture Notes in Computer Science, July/August 1996, Springer-Verlag.

65. J. Palsberg and M. I. Schwartzbach. Three Discussions on Object-Oriented Typing. *ACM SIGPLAN OOPS Messenger*, vol. 3, num. 2, pp. 31-38, 1992.

66. ParcPlace Systems Inc. *VisualWorks 2.0 User's Guide*. Sunnyvale, California, 1994.

67. H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Computing Surveys*, vol. 15, no. 3, pp. 199-236, September 1983.

68. D. E. Perry. The Inscape Environment. In *Proceedings of the 11th International Conference on Software Engineering*, pp. 2-11, Pittsburgh, PA, May 1989.

69. D.E. Perry. Software Architecture and its Relevance to Software Engineering, Invited Talk. *Second International Conference on Coordination Models and Languages (COORD '97)*, Berlin, Germany, September 1997.

70. D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40-52, October 1992.

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

72. J. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 1, pp. 151-174, January 1994.

73. S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, vol. 7, no. 4, pp. 57–66, July 1990.

74. J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Extending Design Environments to Software Architecture Design. In *Proceedings of the 1996 Knowledge-Based Software Engineering Conference (KBSE)*, pp. 63-72, Syracuse, NY, September 1996.

75. D. S. Rosenblum. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering*, vol. 21, no. 1, pp. 19-31, January 1995.

76. 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*, pp. 137-146, Marina del Ray, CA, November 1994.

77. R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April 1986. Actually appeared June 1987.

78. A. Schill, editor. *DCE — The OSF Distributed Computing Environment*. Proceedings of the *International DCE Workshop*, Karlsruhe, Germany, Springer Verlag, October 1993.

79. J. Schumann and B. Fischer. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. In *Proceedings of Automated Software Engineering (ASE-97)*, Lake Tahoe, November 1997.

80. R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, New York, NY, 1997.

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

82. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering,* vol. 21, no. 4, pp. 314-335, April 1995.

83. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

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

85. K. J. Sullivan and D. Notkin. Reconciling Environment Integration and Software Evolution. *ACM Transactions on Software Engineering and Methodology,* vol. 1, no. 3, pp. 229–268, July 1992.

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

87. Sun Microsystems, Inc. Remote Method Invocation. http://java.sun.com:80/products/jdk/rmi/index.html

88. R. N. Taylor. Generalization from domain experience: The superior paradigm for software architecture research? In *Proceedings of the Second International Software Architecture Workshop (ISAW-2),* San Francisco, CA, October 1996.

89. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering,* vol. 22, no. 6, pp. 390-406, June 1996.

90. R. N. Taylor, K. A. Nies, G. A. Bolcer, C. A. MacFarlane, K. M. Anderson, and G. F. Johnson. Chiron-1: A Software Architecture for User Interface Development, Maintenance, and Run-Time Support. *ACM Transactions on Computer-Human Interaction*, vol. 2, no. 2, pp. 105–144, June 1995.

91. W. Tracz. DSSA (Domain-Specific Software Architecture) Pedagogical Example. *ACM SIGSOFT Software Engineering Notes*, vol. 2, no. 4, pp. 49-62, July 1995.

92. The UIMS Tool Developers Workshop. A Metamodel for the Runtime Architecture of an Interactive System. *SIGCHI Bulletin,* vol. 24, no. 1, pp. 32–37, January 1992.

93. S. Vestal. A Cursory Overview and Comparison of Four Architecture Description Languages. Technical Report, Honeywell Technology Center, February 1993.

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

95. E. J. Whitehead, Jr., J. E. Robbins, N. Medvidovic, and R. N. Taylor. Software Architecture: Foundation of a Software Component Marketplace. In *Proceedings of the First*

International Workshop on Architectures for Software Systems, pp. 276-282, Seattle, WA, April 1995.

96. A. L. Wolf, editor. *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, San Francisco, CA, October 1996.

97. Xerox Palo Alto Research Center. ILU — Inter-Language Unification. ftp://ftp.parc.xerox.com/pub/ilu/ilu.html

98. D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proceedings of OOPSLA'94*, Portland, OR, USA, October 1994.

99. A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 4, pp. 333-369, October 1997.

APPENDIX A: Formal Definition of the C2 Style

A.1 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 dissertation. For a complete Z reference, see [84].

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 $y \in Y$,
- $X \nrightarrow Y$, 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:

$INHABITANTS == ADDRESS \nrightarrow \mathbb{P}NAME$

Other Z operations and notational conventions used in the dissertation 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.

$$\begin{array}{|l}
\hline \mathit{SingleTown} \\
\mathit{residences} : \mathbb{P}\ \mathit{ADDRESS} \\
\mathit{residents} : \mathit{INHABITANTS} \\
\hline
\mathit{residences} = \mathrm{dom}\ \mathit{residents} \\
\hline
\end{array}$$

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

$$\begin{array}{|l}
\hline \mathit{SingleTown} \\
\mathit{Town} \\
\hline
\mathit{residences} = \mathrm{dom}\ \mathit{residents} \\
\hline
\end{array}$$

Finally, if *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,

$$\begin{array}{|l}
\hline \mathit{TownGrowth} \\
\Delta \mathit{Town} \\
\hline
\end{array}$$

is equivalent to

$$\begin{array}{|l}
\hline \mathit{TownGrowth} \\
\mathit{Town} \\
\mathit{Town}' \\
\hline
\end{array}$$

A.2 Z Types in the C2 Specification

We define four unelaborated Z types to describe C2 concepts: a component name type, a (component and connector) communication port type, a message type, and a type that corresponds to the state of a component, connector, or architecture.

$$[\mathit{COMP\_NAME}, \mathit{COMM\_PORT}, \mathit{COMM\_MSG}, \mathit{OBJ\_STATE}]$$

A.3 C2 Components

The canonical C2 component type is formally defined below. 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, 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.

$$\mathit{NEXT\_MSG} == \mathbb{P}\ \mathit{COMM\_MSG} \rightarrow \mathit{COMM\_MSG}$$

$C2Component$

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

In order to be able to properly specify component instances, two component types in a C2-style architecture cannot share names.

$ComponentNameUniqueness$

$$\begin{array}{l}
components : \mathbb{P}\ C2Component \\
\hline
\forall\, comp1,\ comp2 : components \bullet \\
\qquad comp1.name = comp2.name \Leftrightarrow comp1 = comp2
\end{array}$$

A component's state is defined by its current state and the incoming and outgoing data currently at its top and bottom ports.

$$
\begin{array}{|l}
\textit{C2ComponentState} \\
\hline
comp : C2Component \\
current\_state : OBJ\_STATE \\
top\_in\_data,\ top\_out\_data,\ bot\_in\_data,\ bot\_out\_data : \\
\qquad COMM\_PORT \pfun \mathbb{P}\ COMM\_MSG \\
\hline
current\_state \in comp.internal\_states \\
\\
\operatorname{dom} top\_in\_data = \{\ comp.top\_port\ \} \\
\operatorname{dom} top\_out\_data = \{\ comp.top\_port\ \} \\
\operatorname{dom} bot\_in\_data = \{\ comp.bot\_port\ \} \\
\operatorname{dom} bot\_out\_data = \{\ comp.bot\_port\ \} \\
\\
top\_in\_data(comp.top\_port) \subseteq comp.top\_in(comp.top\_port) \\
top\_out\_data(comp.top\_port) \subseteq comp.top\_out(comp.top\_port) \\
bot\_in\_data(comp.bot\_port) \subseteq comp.bot\_in(comp.bot\_port) \\
bot\_out\_data(comp.bot\_port) \subseteq comp.bot\_out(comp.bot\_port) \\
\hline
\end{array}
$$

A component handles messages by removing them from either its top or bottom port and processing them, as shown below. The substrate independence principle is reflected in the below schemas. 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 internal object.

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).*

$HandleMessageFromAbove$

$\Delta C2ComponentState$

$$
\begin{array}{l}
comp' = comp \\[4pt]
((current\_state,\ top\_in\_data), \\
\ (current\_state',\ \{\ top\_out\_data',\ bot\_out\_data'\ \})) \in \\
\quad comp.state\_transitions \\[4pt]
top\_out\_data'(comp.top\_port) = \\
\quad top\_out\_data(comp.top\_port) \cup \\
\quad \{\ comp.domain\_trans(\\
\qquad comp.dialog\_top\_out(\\
\qquad\ \ comp.dialog\_in(\\
\qquad\quad comp.domain\_trans(\\
\qquad\qquad comp.msg\_to\_handle(\\
\qquad\qquad\ \ top\_in\_data(comp.top\_port))))))\ \} \\[4pt]
bot\_out\_data'(comp.bot\_port) = \\
\quad bot\_out\_data(comp.bot\_port) \cup \\
\quad \{\ comp.wrapper(\\
\qquad comp.dialog\_in(\\
\qquad\ \ comp.domain\_trans(\\
\qquad\quad comp.msg\_to\_handle(\\
\qquad\qquad top\_in\_data(comp.top\_port)))))\ \} \\[4pt]
top\_in\_data(comp.top\_port) = \\
\quad top\_in\_data'(comp.top\_port) \cup \\
\quad \{\ comp.msg\_to\_handle(top\_in\_data(comp.top\_port))\ \} \\[4pt]
bot\_in\_data'(comp.bot\_port) = bot\_in\_data(comp.bot\_port)
\end{array}
$$

$HandleMessageFromBelow$

$\Delta C2ComponentState$

$$
\begin{array}{l}
comp' = comp \\[4pt]
((current\_state,\ bot\_in\_data), \\
\ (current\_state',\ \{\ top\_out\_data',\ bot\_out\_data'\ \})) \in \\
\quad comp.state\_transitions \\[4pt]
top\_out\_data'(comp.top\_port) = \\
\quad top\_out\_data(comp.top\_port) \cup \\
\quad \{\ comp.domain\_trans(\\
\qquad comp.dialog\_top\_out(\\
\qquad\quad comp.dialog\_in(\\
\qquad\qquad comp.msg\_to\_handle(\\
\qquad\qquad\ \ bot\_in\_data(comp.bot\_port)))))\ \} \\[4pt]
bot\_out\_data'(comp.bot\_port) = \\
\quad bot\_out\_data(comp.bot\_port) \cup \\
\quad \{\ comp.wrapper(\\
\qquad comp.dialog\_in(\\
\qquad\quad comp.msg\_to\_handle(\\
\qquad\qquad bot\_in\_data(comp.bot\_port))))\ \} \\[4pt]
top\_in\_data'(comp.top\_port) = top\_in\_data(comp.top\_port) \\[4pt]
bot\_in\_data(comp.bot\_port) = \\
\quad bot\_in\_data'(comp.bot\_port) \cup \\
\quad \{\ comp.msg\_to\_handle(bot\_in\_data(comp.bot\_port))\ \}
\end{array}
$$

$$
\begin{array}{l}
ComponentMessageHandling \mathrel{\widehat=} \\
\qquad HandleMessageFromAbove \wedge HandleMessageFromBelow
\end{array}
$$

A.4 C2 Connectors

A C2 connector can have multiple components and connectors on its top and bottom sides. The messages emitted on the bottom side of a connector are a subset of those that come in from above and the messages emitted on its top side are a subset of those that come in from below. 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.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{C2Connector}} \\
\hline
top\_ports,\ bot\_ports : \mathbb{P}\ COMM\_PORT \\
top\_in,\ top\_out,\ bot\_in,\ bot\_out : \\
\qquad COMM\_PORT \pfun \mathbb{P}\ COMM\_MSG \\
Filter\_TB : CONN\_FILTER \\
Filter\_BT : CONN\_FILTER \\
\hline
top\_ports \cap bot\_ports = \emptyset \\
\\
\operatorname{dom} top\_in = top\_ports \\
\operatorname{dom} top\_out = top\_ports \\
\operatorname{dom} bot\_in = bot\_ports \\
\operatorname{dom} bot\_out = bot\_ports \\
\\
\bigcup(\operatorname{ran} bot\_out) \subseteq \bigcup(\operatorname{ran} top\_in) \\
\bigcup(\operatorname{ran} top\_out) \subseteq \bigcup(\operatorname{ran} bot\_in) \\
\hline
\end{array}
$$

Unlike a component, a C2 connector does not perform any system functionality and is thus modeled as not having any “internal” state. Instead, its state is entirely determined by the incoming and outgoing data at its top and bottom ports.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{C2ConnectorState}} \\
\hline
conn : C2Connector \\
top\_in\_flow,\ top\_out\_flow,\ bot\_in\_flow,\ bot\_out\_flow : \\
\qquad COMM\_PORT \pfun \mathbb{P}\ COMM\_MSG \\
\hline
\operatorname{dom} top\_in\_flow \cup \operatorname{dom} top\_out\_flow \subseteq conn.top\_ports \\
\operatorname{dom} bot\_in\_flow \cup \operatorname{dom} bot\_out\_flow \subseteq conn.bot\_ports \\
\\
\forall\, port : conn.top\_ports \bullet \\
\qquad top\_in\_flow(port) \subseteq conn.top\_in(port) \land \\
\qquad top\_out\_flow(port) \subseteq conn.top\_out(port) \\
\forall\, port : conn.bot\_ports \bullet \\
\qquad bot\_in\_flow(port) \subseteq conn.bot\_in(port) \land \\
\qquad bot\_out\_flow(port) \subseteq conn.bot\_out(port) \\
\hline
\end{array}
$$

A connector routes a message by removing it from one of its ports’ incoming queues, filtering it as appropriate, and placing it on its opposite-side ports’ outgoing queues. For simplicity, the *FilterTB* and *FilterBT* functions are assumed to filter out a message by propagating a null message.

$$
\begin{array}{|l}
\textit{RoutMessageFromAbove} \\
\hline
\Delta C2ConnectorState \\
\hline
conn' = conn \\[1ex]
\forall\, msg : COMM\_MSG;\ port1 : conn.top\_ports \mid msg \in top\_in\_flow(port1) \bullet \\
\quad \forall\, port2 : conn.bot\_ports \bullet \\
\qquad\qquad top\_in\_flow(port1) = top\_in\_flow'(port1) \cup \{\ msg\ \} \\
\qquad \wedge \quad top\_out\_flow'(port1) = top\_out\_flow(port1) \\
\qquad \wedge \quad bot\_in\_flow'(port2) = bot\_in\_flow(port2) \\
\qquad \wedge \quad bot\_out\_flow'(port2) = \\
\qquad\qquad\qquad bot\_out\_flow(port2) \cup \{\ conn.Filter\_TB(port2,\ msg)\ \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{RoutMessageFromBelow} \\
\hline
\Delta C2ConnectorState \\
\hline
conn' = conn \\[1ex]
\forall\, msg : COMM\_MSG;\ port1 : conn.bot\_ports \mid msg \in bot\_in\_flow(port1) \bullet \\
\quad \forall\, port2 : conn.top\_ports \bullet \\
\qquad\qquad top\_in\_flow'(port2) = top\_in\_flow(port2) \\
\qquad \wedge \quad top\_out\_flow'(port2) = \\
\qquad\qquad\qquad top\_out\_flow(port2) \cup \{\ conn.Filter\_BT(port2,\ msg)\ \} \\
\qquad \wedge \quad bot\_in\_flow(port1) = bot\_in\_flow'(port1) \cup \{\ msg\ \} \\
\qquad \wedge \quad bot\_out\_flow'(port1) = bot\_out\_flow(port1) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
ConnectorMessageRouting \mathrel{\widehat=} \\
\qquad RoutMessageFromAbove \wedge RoutMessageFromBelow
\end{array}
$$

A.5 Rules of Architectural Composition

We define a communication link as a relation between two ports. Communication links are bidirectional.

$$LINK == COMM\_PORT \leftrightarrow COMM\_PORT$$

$$
\begin{array}{|l}
\textit{C2Link} \\
\hline
Link : LINK \\
\hline
\forall\, port1,\ port2 : COMM\_PORT \bullet \\
\quad (port1,\ port2) \in Link \Leftrightarrow (port2,\ port1) \in Link \\
\hline
\end{array}
$$

The properties that a component may only be attached to single connectors on its top and bottom sides, while a connector may be attached to multiple components and other connectors are expressed below. These, and all subsequent 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.

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

$$
\begin{array}{l}
\textit{ConnectorToComponentLinks} \\
\hline
C2Link \\
components : \mathbb{P}\ C2Component \\
connectors : \mathbb{P}\ C2Connector \\
\hline
\forall\ conn : connectors;\ tport,\ bport : COMM\_PORT \mid \\
\quad tport \in conn.top\_ports \land bport \in conn.bot\_ports \bullet \\
\qquad \exists_1\ comp1,\ comp2 : components \mid comp1 \neq comp2 \bullet \\
\qquad\quad (tport,\ comp1.bot\_port) \in Link \land (bport,\ comp2.top\_port) \in Link \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{ConnectorToConnectorLinks} \\
\hline
C2Link \\
components : \mathbb{P}\ C2Component \\
connectors : \mathbb{P}\ C2Connector \\
\hline
\forall\ conn : connectors;\ tport,\ bport : COMM\_PORT \mid \\
\quad tport \in conn.top\_ports \land bport \in conn.bot\_ports \bullet \\
\qquad \exists_1\ conn1,\ conn2 : connectors;\ c2tport, c1bport : COMM\_PORT \mid \\
\qquad\quad c2tport \in conn2.top\_ports \land c1bport \in conn1.bot\_ports \land \\
\qquad\quad conn \neq conn1 \land conn \neq conn2 \land conn1 \neq conn2 \bullet \\
\qquad\qquad (tport,\ c1bport) \in Link \land (bport,\ c2tport) \in Link \\
\hline
\end{array}
$$

$$
\begin{array}{l}
ValidC2Connections \mathrel{\widehat{=}} \\
\qquad ComponentToConnectorLinks \land \\
\qquad ConnectorToComponentLinks \land \\
\qquad ConnectorToConnectorLinks
\end{array}
$$

A.6 Communication among Components and Connectors

C2 connectors' interfaces are *context reflective*: a connector's domain of discourse is determined (dynamically) by the domains of the components attached to it at a given time. Hence a connector will accept and process every message sent by the components attached to it.

$$
\begin{array}{l}
\textit{ConnectorDomains} \\
\hline
vc : ValidC2Connections \\
components : \mathbb{P}\ C2Component \\
connectors : \mathbb{P}\ C2Connector \\
\hline
\forall\ conn : connectors;\ comp : components;\ conn\_port : COMM\_PORT \bullet \\
\quad conn\_port \in conn.bot\_ports \land (comp.top\_port, conn\_port) \in vc.Link \Rightarrow \\
\qquad comp.top\_out(comp.top\_port) \subseteq conn.bot\_in(conn\_port) \\
\land \quad conn\_port \in conn.top\_ports \land (comp.bot\_port, conn\_port) \in vc.Link \Rightarrow \\
\qquad comp.bot\_out(comp.bot\_port) \subseteq conn.top\_in(conn\_port) \\
\hline
\end{array}
$$

Message passing is modeled as a simple exchange between connector and/or component ports.

$TransmitMessageUpFromComponentToConnector$

$$
\begin{array}{l}
\Xi\, ValidC2Connections \\
\Delta\, C2ComponentState \\
\Delta\, C2ConnectorState \\
\hline
\forall\, msg : COMM\_MSG;\ bot\_conn\_port : conn.bot\_ports \bullet \\
\quad (comp.top\_port,\ bot\_conn\_port) \in Link \land \\
\quad msg \in top\_out\_data(comp.top\_port) \Rightarrow \\
\qquad (\ top\_out\_data(comp.top\_port) = \\
\qquad\qquad top\_out\_data'(comp.top\_port) \cup \{\ msg\ \} \\
\qquad \land\ bot\_in\_flow'(bot\_conn\_port) = \\
\qquad\qquad\quad bot\_in\_flow(bot\_conn\_port) \cup \{\ msg\ \} \\
\qquad \land\ conn' = conn \\
\qquad \land\ top\_in\_flow' = top\_in\_flow \\
\qquad \land\ top\_out\_flow' = top\_out\_flow \\
\qquad \land\ bot\_out\_flow' = bot\_out\_flow \\
\qquad \land\ comp' = comp \\
\qquad \land\ current\_state' = current\_state \\
\qquad \land\ top\_in\_data' = top\_in\_data \\
\qquad \land\ bot\_in\_data' = bot\_in\_data \\
\qquad \land\ bot\_out\_data' = bot\_out\_data\ )
\end{array}
$$

$TransmitMessageDownFromComponentToConnector$

$$
\begin{array}{l}
\Xi\, ValidC2Connections \\
\Delta\, C2ComponentState \\
\Delta\, C2ConnectorState \\
\hline
\forall\, msg : COMM\_MSG;\ top\_conn\_port : conn.top\_ports \bullet \\
\quad (comp.bot\_port,\ top\_conn\_port) \in Link \land \\
\quad msg \in bot\_out\_data(comp.bot\_port) \Rightarrow \\
\qquad (\ bot\_out\_data(comp.bot\_port) = \\
\qquad\qquad bot\_out\_data'(comp.bot\_port) \cup \{\ msg\ \} \\
\qquad \land\ top\_in\_flow'(top\_conn\_port) = \\
\qquad\qquad\quad top\_in\_flow(top\_conn\_port) \cup \{\ msg\ \} \\
\qquad \land\ conn' = conn \\
\qquad \land\ top\_out\_flow' = top\_out\_flow \\
\qquad \land\ bot\_in\_flow' = bot\_in\_flow \\
\qquad \land\ bot\_out\_flow' = bot\_out\_flow \\
\qquad \land\ comp' = comp \\
\qquad \land\ current\_state' = current\_state \\
\qquad \land\ top\_in\_data' = top\_in\_data \\
\qquad \land\ top\_out\_data' = top\_out\_data \\
\qquad \land\ bot\_in\_data' = bot\_in\_data\ )
\end{array}
$$

$$
\begin{array}{l}
\textit{TransmitMessageUpFromConnectorToConnector} \\
\hline
\Xi \textit{ValidC2Connections} \\
\textit{from\_conn},\ \textit{to\_conn} : \Delta \textit{C2ConnectorState} \\
\hline
\forall\ \textit{msg} : \textit{COMM\_MSG}; \\
\quad \textit{from\_port} : \textit{from\_conn.conn.top\_ports};\ \textit{to\_port} : \textit{to\_conn.conn.bot\_ports} \bullet \\
\qquad (\textit{from\_port},\ \textit{to\_port}) \in \textit{Link} \land \\
\qquad \textit{msg} \in \textit{from\_conn.top\_out\_flow}(\textit{from\_port}) \Rightarrow \\
\qquad\quad (\ \textit{from\_conn.top\_out\_flow}(\textit{from\_port}) = \\
\qquad\qquad\quad \textit{from\_conn.top\_out\_flow}'(\textit{from\_port}) \cup \{\ \textit{msg}\ \} \\
\qquad\quad \land\ \ \textit{to\_conn.bot\_in\_flow}'(\textit{to\_port}) = \\
\qquad\qquad\qquad \textit{to\_conn.bot\_in\_flow}(\textit{to\_port}) \cup \{\ \textit{msg}\ \} \\
\qquad\quad \land\ \ \textit{from\_conn.conn}' = \textit{from\_conn.conn} \\
\qquad\quad \land\ \ \textit{from\_conn.top\_in\_flow}' = \textit{from\_conn.top\_in\_flow} \\
\qquad\quad \land\ \ \textit{from\_conn.bot\_in\_flow}' = \textit{from\_conn.bot\_in\_flow} \\
\qquad\quad \land\ \ \textit{from\_conn.bot\_out\_flow}' = \textit{from\_conn.bot\_out\_flow} \\
\qquad\quad \land\ \ \textit{to\_conn.conn}' = \textit{to\_conn.conn} \\
\qquad\quad \land\ \ \textit{to\_conn.top\_in\_flow}' = \textit{to\_conn.top\_in\_flow} \\
\qquad\quad \land\ \ \textit{to\_conn.top\_out\_flow}' = \textit{to\_conn.top\_out\_flow} \\
\qquad\quad \land\ \ \textit{to\_conn.bot\_out\_flow}' = \textit{from\_conn.bot\_out\_flow}\ ) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{TransmitMessageDownFromConnectorToConnector} \\
\hline
\Xi \textit{ValidC2Connections} \\
\textit{from\_conn},\ \textit{to\_conn} : \Delta \textit{C2ConnectorState} \\
\hline
\forall\ \textit{msg} : \textit{COMM\_MSG}; \\
\quad \textit{from\_port} : \textit{from\_conn.conn.bot\_ports};\ \textit{to\_port} : \textit{to\_conn.conn.top\_ports} \bullet \\
\qquad (\textit{from\_port},\ \textit{to\_port}) \in \textit{Link} \land \\
\qquad \textit{msg} \in \textit{from\_conn.bot\_out\_flow}(\textit{from\_port}) \Rightarrow \\
\qquad\quad (\ \textit{from\_conn.bot\_out\_flow}(\textit{from\_port}) = \\
\qquad\qquad\quad \textit{from\_conn.bot\_out\_flow}'(\textit{from\_port}) \cup \{\ \textit{msg}\ \} \\
\qquad\quad \land\ \ \textit{to\_conn.top\_in\_flow}'(\textit{to\_port}) = \\
\qquad\qquad\qquad \textit{to\_conn.top\_in\_flow}(\textit{to\_port}) \cup \{\ \textit{msg}\ \} \\
\qquad\quad \land\ \ \textit{from\_conn.conn}' = \textit{from\_conn.conn} \\
\qquad\quad \land\ \ \textit{from\_conn.top\_in\_flow}' = \textit{from\_conn.top\_in\_flow} \\
\qquad\quad \land\ \ \textit{from\_conn.top\_out\_flow}' = \textit{from\_conn.top\_out\_flow} \\
\qquad\quad \land\ \ \textit{from\_conn.bot\_in\_flow}' = \textit{from\_conn.bot\_in\_flow} \\
\qquad\quad \land\ \ \textit{to\_conn.conn}' = \textit{to\_conn.conn} \\
\qquad\quad \land\ \ \textit{to\_conn.top\_out\_flow}' = \textit{to\_conn.top\_out\_flow} \\
\qquad\quad \land\ \ \textit{to\_conn.bot\_in\_flow}' = \textit{to\_conn.bot\_in\_flow} \\
\qquad\quad \land\ \ \textit{to\_conn.bot\_out\_flow}' = \textit{from\_conn.bot\_out\_flow}\ ) \\
\hline
\end{array}
$$

Unlike a connector, a component has explicitly defined top and bottom interfaces, and will accept only those messages it understands.

$$
\begin{array}{l}
\textit{TransmitMessageUpFromConnectorToComponent} \\
\hline
\Xi\, ValidC2Connections \\
\Delta\, C2ComponentState \\
\Delta\, C2ConnectorState \\
\hline
\forall\, msg : COMM\_MSG;\ top\_conn\_port : conn.top\_ports \bullet \\
\quad (comp.bot\_port,\ top\_conn\_port) \in Link\ \wedge \\
\quad msg \in top\_out\_flow(top\_conn\_port) \Rightarrow \\
\qquad (\ top\_out\_flow(top\_conn\_port) = \\
\qquad\qquad top\_out\_flow'(top\_conn\_port) \cup \{\ msg\ \} \\
\qquad \wedge\ \ msg \in comp.bot\_in(comp.bot\_port) \Rightarrow \\
\qquad\qquad bot\_in\_data'(comp.bot\_port) = \\
\qquad\qquad\quad bot\_in\_data(comp.bot\_port) \cup \{\ msg\ \} \\
\qquad \wedge\ \ msg \notin comp.bot\_in(comp.bot\_port) \Rightarrow \\
\qquad\qquad bot\_in\_data'(comp.bot\_port) = bot\_in\_data(comp.bot\_port) \\
\qquad \wedge\ \ conn' = conn \\
\qquad \wedge\ \ top\_in\_flow' = top\_in\_flow \\
\qquad \wedge\ \ bot\_in\_flow' = bot\_in\_flow \\
\qquad \wedge\ \ bot\_out\_flow' = bot\_out\_flow \\
\qquad \wedge\ \ comp' = comp \\
\qquad \wedge\ \ current\_state' = current\_state \\
\qquad \wedge\ \ top\_in\_data' = top\_in\_data \\
\qquad \wedge\ \ top\_out\_data' = top\_out\_data \\
\qquad \wedge\ \ bot\_out\_data' = bot\_out\_data\ ) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{TransmitMessageDownFromConnectorToComponent} \\
\hline
\Xi\, ValidC2Connections \\
\Delta\, C2ComponentState \\
\Delta\, C2ConnectorState \\
\hline
\forall\, msg : COMM\_MSG;\ bot\_conn\_port : conn.bot\_ports \bullet \\
\quad (comp.top\_port,\ bot\_conn\_port) \in Link\ \wedge \\
\quad msg \in bot\_out\_flow(bot\_conn\_port) \Rightarrow \\
\qquad (\ bot\_out\_flow(bot\_conn\_port) = \\
\qquad\qquad bot\_out\_flow'(bot\_conn\_port) \cup \{\ msg\ \} \\
\qquad \wedge\ \ msg \in comp.top\_in(comp.top\_port) \Rightarrow \\
\qquad\qquad top\_in\_data'(comp.top\_port) = \\
\qquad\qquad\quad top\_in\_data(comp.top\_port) \cup \{\ msg\ \} \\
\qquad \wedge\ \ msg \notin comp.top\_in(comp.top\_port) \Rightarrow \\
\qquad\qquad top\_in\_data'(comp.top\_port) = top\_in\_data(comp.top\_port) \\
\qquad \wedge\ \ conn' = conn \\
\qquad \wedge\ \ bot\_in\_flow' = bot\_in\_flow \\
\qquad \wedge\ \ top\_in\_flow' = top\_in\_flow \\
\qquad \wedge\ \ top\_out\_flow' = top\_out\_flow \\
\qquad \wedge\ \ comp' = comp \\
\qquad \wedge\ \ current\_state' = current\_state \\
\qquad \wedge\ \ bot\_in\_data' = bot\_in\_data \\
\qquad \wedge\ \ bot\_out\_data' = bot\_out\_data \\
\qquad \wedge\ \ top\_out\_data' = top\_out\_data\ ) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
C2MessageTransmition \mathrel{\widehat=} \\
\quad TransmitMessageUpFromComponentToConnector \land \\
\quad TransmitMessageDownFromComponentToConnector \land \\
\quad TransmitMessageUpFromConnectorToConnector \land \\
\quad TransmitMessageDownFromConnectorToConnector \land \\
\quad TransmitMessageUpFromConnectorToComponent \land \\
\quad TransmitMessageDownFromConnectorToComponent
\end{array}
$$

The *ServiceUtilization* schemas represent the communication between a component and a connector from the component's perspective. In other words, they represent the usage of the component's provided services. An informal discussion of the below schemas is given in Chapter 2.

$$
\begin{array}{|l}
\hline
FullServiceUtilization \\
vc : ValidC2Connections \\
components : \mathbb{P}\ C2Component \\
connectors : \mathbb{P}\ C2Connector \\
\hline
\forall comp : components;\ conn : connectors;\ conn\_port : COMM\_PORT \bullet \\
\quad (\ conn\_port \in conn.top\_ports \land (comp.bot\_port,\ conn\_port) \in vc.Link \Rightarrow \\
\qquad comp.bot\_in(comp.bot\_port) \subseteq conn.top\_out(conn\_port)\ ) \\
\ \land\ (\ conn\_port \in conn.bot\_ports \land (comp.top\_port,\ conn\_port) \in vc.Link \Rightarrow \\
\qquad comp.top\_in(comp.top\_port) \subseteq conn.bot\_out(conn\_port)\ ) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
PartialServiceUtilization \\
vc : ValidC2Connections \\
components : \mathbb{P}\ C2Component \\
connectors : \mathbb{P}\ C2Connector \\
\hline
\forall comp : components;\ conn : connectors;\ conn\_port : COMM\_PORT \bullet \\
\quad (\ conn\_port \in conn.top\_ports \land (comp.bot\_port,\ conn\_port) \in vc.Link \Rightarrow \\
\qquad conn.top\_out(conn\_port) \cap comp.bot\_in(comp.bot\_port) \neq \emptyset \\
\quad\ \land\ comp.bot\_in(comp.bot\_port) \cap conn.top\_out(conn\_port) \subset \\
\qquad\quad comp.bot\_in(comp.bot\_port)\ ) \\
\ \land\ (\ conn\_port \in conn.bot\_ports \land (comp.top\_port,\ conn\_port) \in vc.Link \Rightarrow \\
\qquad conn.bot\_out(conn\_port) \cap comp.top\_in(comp.top\_port) \neq \emptyset \\
\quad\ \land\ comp.top\_in(comp.top\_port) \cap conn.bot\_out(conn\_port) \subset \\
\qquad\quad comp.top\_in(comp.top\_port)\ ) \\
\hline
\end{array}
$$

Finally, the *Communication* schemas represent the communication between a component and a connector from the connector's perspective. In other words, these schemas represent the fulfillment of the requests submitted by a connector. Their informal discussion is given in Chapter 2.

$$
\begin{array}{|l}
\hline
\mathit{FullCommunication} \\
vc : \mathit{ValidC2Connections} \\
\mathit{components} : \mathbb{P}\ \mathit{C2Component} \\
\mathit{connectors} : \mathbb{P}\ \mathit{C2Connector} \\
\hline
\forall\, comp : components;\ conn : connectors;\ conn\_port : COMM\_PORT \bullet \\
\quad (\ conn\_port \in conn.top\_ports \land (comp.bot\_port,\ conn\_port) \in vc.Link \Rightarrow \\
\qquad conn.top\_out(conn\_port) \subseteq comp.bot\_in(comp.bot\_port)\ ) \\
\quad \land\ (\ conn\_port \in conn.bot\_ports \land (comp.top\_port,\ conn\_port) \in vc.Link \Rightarrow \\
\qquad conn.bot\_out(conn\_port) \subseteq comp.top\_in(comp.top\_port)\ ) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{PartialCommunication} \\
vc : \mathit{ValidC2Connections} \\
\mathit{components} : \mathbb{P}\ \mathit{C2Component} \\
\mathit{connectors} : \mathbb{P}\ \mathit{C2Connector} \\
\hline
\forall\, comp : components;\ conn : connectors;\ conn\_port : COMM\_PORT \bullet \\
\quad (\ conn\_port \in conn.top\_ports \land (comp.bot\_port,\ conn\_port) \in vc.Link \Rightarrow \\
\qquad conn.top\_out(conn\_port) \cap comp.bot\_in(comp.bot\_port) \neq \emptyset \\
\qquad \land\ comp.bot\_in(comp.bot\_port) \cap conn.top\_out(conn\_port) \subset \\
\qquad\qquad conn.top\_out(conn\_port)\ ) \\
\quad \land\ (\ conn\_port \in conn.bot\_ports \land (comp.top\_port,\ conn\_port) \in vc.Link \Rightarrow \\
\qquad conn.bot\_out(conn\_port) \cap comp.top\_in(comp.top\_port) \neq \emptyset \\
\qquad \land\ comp.top\_in(comp.top\_port) \cap conn.bot\_out(conn\_port) \subset \\
\qquad\qquad conn.bot\_out(conn\_port)\ ) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{NoInteraction} \\
vc : \mathit{ValidC2Connections} \\
\mathit{components} : \mathbb{P}\ \mathit{C2Component} \\
\mathit{connectors} : \mathbb{P}\ \mathit{C2Connector} \\
\hline
\forall\, comp : components;\ conn : connectors;\ conn\_port : COMM\_PORT \bullet \\
\quad (\ conn\_port \in conn.top\_ports \land (comp.bot\_port,\ conn\_port) \in vc.Link \Rightarrow \\
\qquad conn.top\_out(conn\_port) \cap comp.bot\_in(comp.bot\_port) = \emptyset\ ) \\
\quad \land\ (\ conn\_port \in conn.bot\_ports \land (comp.top\_port,\ conn\_port) \in vc.Link \Rightarrow \\
\qquad conn.bot\_out(conn\_port) \cap comp.top\_in(comp.top\_port) = \emptyset\ ) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\mathit{ComponentServiceUtilization} \mathrel{\widehat{=}} \\
\qquad \mathit{FullServiceUtilization} \lor \mathit{PartialServiceUtilization} \lor \mathit{NoInteraction}
\end{array}
$$

$$
\begin{array}{l}
\mathit{ConnectorToComponentCommunication} \mathrel{\widehat{=}} \\
\qquad \mathit{FullCommunication} \lor \mathit{PartialCommunication} \lor \mathit{NoInteraction}
\end{array}
$$

APPENDIX B: C2SADEL Syntax Summary

This Appendix contains the complete BNF specification of C2SADEL. For simplicity, all literals, including single-character literals (e.g., ‘}’ or ‘;’) are displayed in bold type. Single-character literals are displayed without quotation marks.Unless bolded, curly braces (‘{’ and ‘}’) represent repetition of the enclosed expression. “{...}*” represents zero or more occurrences, while “{...}+” denotes one or more occurrences.

arch_component_set ::=
   (arch_component_type)*

arch_component_type ::=
   component identifier **is** arch_component_type_decl

arch_component_type_decl ::=
   component_type_decl | virtual_comp_type

arch_component_types ::=
   component_types { arch_component_set **}**

arch_connector_type ::=
   connector identifier **is**
   {
      message_filter filtering_policy **;**
   }

arch_connector_types ::=
   connector_types { (arch_connector_type)* **}**

arch_topology ::=
   architectural_topology
   {
      component_inst
      connector_inst
      attachments
   }

attachments ::=
   connections { (connection_decl)* **}**

basic_subtype ::=
   identifier **is basic_subtype** identifier **;**

basic_subtype_decl ::=
   basic_types { (basic_subtype)* **}**

behavior_decl ::=
   operations { (operation_decl)* **}**

binary_operator ::=
   = | **<>** | **+** | **-** | ***** | **/** | **^** |
   \implies | **\equivalent** |
   \and | **\or** |
   \union | **\intersection** | **\in** | **\not_in**
   \greater | **\less** | **\eqgreater** | **\eqless**

```
C2_architecture ::=
   architecture identifier is
   {
      [basic_subtype_decl]
      arch_component_types
      arch_connector_types
      arch_topology
   }

C2_component_set ::=
   [basic_subtype_decl]
   (component_type)+

C2_SADEL_spec ::=
   C2_architecture | C2_component_set

component_inst ::=
   component_instances { (instance_decl)* }

component_type ::=
   component identifier is component_type_decl

component_type_decl ::=
   extern_comp_type | local_comp_type

connection_decl ::=
   [ component | connector ] identifier
   {
      top [ connection_list ] ;
      bottom [ connection_list ] ;
   }

connection_list ::=
   identifier , connection_list | identifier

connector_inst ::=
   connector_instances { (instance_decl)* }

digit ::=
   0 | 1 | ... | 9

dir_indicator ::=
   prov | req

extern_comp_type ::=
   extern { filename ; }

filtering_policy ::=
   no_filtering              |
   notification_filtering    |
   message_filtering         |
   prioritized               |
   message_sink

function_decl ::=
   identifier : identifier -> identifier ;

identifier ::=
   letter {_ | letter | digit}*

instance_decl ::=
   identifier : identifier ;
```

```
integer ::=
   (digit)+

interface_decl ::=
   interface { (interface_element_decl)* }

interface_element_decl ::=
   dir_indicator identifier :
      identifier ( param_decl ) [: [\set] identifier] ;

invariant_decl ::=
   invariant { [logic_expr ;] }

let_decl ::=
   let {var_decl ;}* [pre_decl | post_decl]

letter ::=
   A | B | ... | Z | a | b | ... | z

local_comp_type ::=
   [subtype_decl]
   {
      state_decl
      invariant_decl
      interface_decl
      behavior_decl
      map_decl
   }

logic_expr ::=
   subexpr [\and subexpr]

map_decl ::=
   map { (single_map)* }

numeric_literal ::=
   [-] integer [. integer] [^ integer]

operand ::=
   [\not | #] identifier     |
   numeric_literal           |
   subexpr                   |
   ( subexpr )

operation_decl ::=
   dir_indicator identifier :
   { let_decl | pre_decl | post_decl }

param_decl ::=
   var_decl ; param_decl | var_decl

param_to_var ::=
   identifier -> identifier , param_to_var |
   identifier -> identifier

post_decl ::=
   post [post_logic_expr] ;

post_logic_expr ::=
   post_subexpr [\and post_subexpr]
```

```
post_operand ::=
   [\not | #] [~] identifier      |
   numeric_literal                |
   post_subexpr                   |
   ( post_subexpr )

post_subexpr ::=
   post_operand binary_operator post_operand |
   \result = post_operand

pre_decl ::=
   pre [logic_expr] ; [post_decl]

single_map ::=
   identifier -> identifier ( param_to_var ) ;

state_decl ::=
   state { (var_decl ; | function_decl ;)* }

subexpr ::=
   operand binary_operator operand

subtype_decl ::=
   subtype identifier ( subtype_rel_expr )

subtype_rel ::=
   nam | int | beh | imp

subtype_rel_expr ::=
   [\not] subtype_rel {\and [\not] subtype_rel}*

var_decl ::=
   identifier : [\set] identifier

virtual_comp_type ::=
   virtual { }
```