A Family of Software Architecture Implementation Frameworks

Nenad Medvidovic Nikunj Mehta Marija Mikic-Rakic
Computer Science Department
Henry Salvatori Computer Science Center 300
University of Southern California
Los Angeles, CA 90089-0781, USA
Phone: +1-213-740-5579

{neno,mehta,marija}@usc.edu

ABSTRACT

Software architectures provide high-level abstractions for representing the structure, behavior, and key properties of software systems. Various architecture description languages, styles, tools, and technologies have emerged over the past decade. At the same time, there has been comparatively little focus on techniques and technologies for transforming architectural models into running systems. This often results in significant differences between conceptual and concrete architectures, rendering system evolution and maintenance difficult. Furthermore, it calls into question the ability of developers to consistently transfer the key architectural properties into system implementations. One solution to this problem is to employ architectural frameworks. Architectural frameworks provide support for implementing, deploying, executing, and evolving software architectures. This paper describes the design of and our experience with a family of architectural frameworks that support implementation of systems in a specific architectural style–C2. These frameworks have been implemented in different programming languages and used on different hardware platforms, including resource constrained (e.g., hand held) devices. The frameworks are lightweight and extensible, and allow application monitoring and analysis at run time. To date, the C2 frameworks have been used in the development of over 100 applications by several academic and industrial organizations. The paper discusses the issues we have encountered in implementing and using the frameworks, as well as the approaches adopted to resolve these issues.

Keywords

architectural framework, architectural style, object-orientation, architectural implementation

1. INTRODUCTION

Software architectures provide high-level abstractions for representing structure, behavior, and key properties of a software system [20,29]. These abstractions are useful in describing to various stakeholders complex, real-world problems in an understandable manner. Software architectures are described in terms of *components*, *connectors*, and *configurations*. Architectural components describe the computations and state of a system; connectors describe the rules of interaction among the components; finally, configurations define topologies of components and connectors.

There have been two largely disjoint paths of research in the field of software architectures: one path has focused on the *design* issues, formal foundations, and analysis of architectures, while the other has resulted in technologies for *implementing* software architectures. The first approach has focused on architectural design abstractions called *styles* and the semantics underpinning such styles [29]. An *architectural style* defines a *vocabulary* of component and connector types and a set of *constraints* on how instances of these types may be combined in a system or family of systems. Various formal architecture description languages (ADLs) and their supporting tools have emerged from this body of research over the last decade [16]. To date, most architectural tools have focused on the simulation and analysis of architectural models to exploit the semantic power of ADLs. At the same time, insufficient progress has been made on supporting implementation of applications based on styles and ADL models [16].

The second approach to software architectures has focused on providing software frameworks, often through object-oriented (OO) reuse techniques such as design patterns and object hierarchies. A *framework* is a skeletal group of software modules that may be tailored for building domain-specific applications, typically resulting in increased productivity and faster time-to-market [5,34]. This approach has lead to creation of a variety of “middleware” techniques and associated commercial technologies for component-based development [27,32,35]. However, software implementations resulting from such use of frameworks often differ widely from conceptual models and lack adequate semantic underpinnings for analytic purposes.

A major focus of our work has been precisely on alleviating this problem, and bridging the two approaches described above. While supporting powerful analysis of architectural models, we have also provided implementation support for conceptual architectures based on a specific architectural style, C2 [31]. This paper describes our experience over the past seven years in developing a family of *architectural frameworks* for supporting the implementation, deployment, execution, monitoring, and evolution of applications built according to C2. In this paper we demonstrate how applications designed in the C2 style are implemented using an architectural framework. We also describe how this framework has evolved over time to support various extra-functional properties required in applications, ranging from distributed enterprise applications to desktop systems and hand-held devices. This evolution has resulted in a family of architectural frameworks that has been used in more than 100 applications developed by various academic and industrial organizations. We have also provided special-purpose utilities (software connectors [17]) that allow seamless construction of C2-style applications from components implemented in different programming languages (PLs), on top of different frameworks.

The remainder of this paper is organized as follows. Section 2 summarizes the key characteristics of the C2 style. Section 3 presents our objectives for framework development, while Section 4

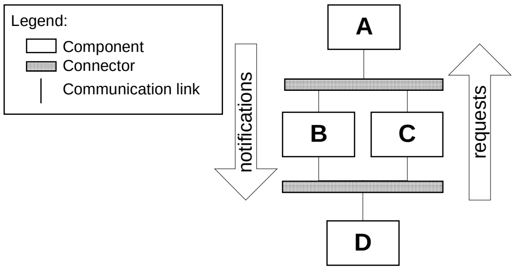

Figure 1. A simple architecture in the C2 style

describes our approach to realizing these objectives and a brief example illustrating the approach. Section 5 discusses the evolution of our framework design, resulting in a family of architectural frameworks. Section 6 highlights the lessons learned in the process. Section 7 discusses related approaches. The paper concludes with a brief overview of future work.

2. DESCRIPTION OF THE C2 STYLE

C2 [31] is an architectural style for highly distributed, dynamic, and evolvable systems. An ADL, C2SADEL [15] has been developed to formally describe C2-style architectures. C2-style architectures are described through a set of components and connectors, and the topology into which they are composed. C2-style components maintain state information and perform application-specific computation. They interact with other components by exchanging messages via their two interfaces (named “top” and “bottom”). Connectors mediate the interaction among components by controlling the transmission and distribution of messages. Messages consist of a name and set of typed parameters. A message in the C2 style is either a *request* for a component to perform an operation, or a *notification* that a given component has performed an operation or changed its state. Each component interface consists of a set of messages that may be received and a set that may be sent. Request messages may only be sent through the top interfaces, while notifications may only be sent through the bottom interfaces of components and connectors.

The C2 style mandates that components must always interact via connectors. The top (bottom) of a component can be attached to the bottom (top) of a single connector; there is no bound on the number of components or connectors that may be attached to a single connector. This decoupling of components greatly aids system flexibility and gradual evolution, where components can be added to or removed from an architecture with minimal impact on the rest of the system [19]. Components and connectors can have an internal architecture, such that they are composed of further components and connectors as long as they support the above C2-style rules. Figure 1 illustrates an architectural configuration in the C2 style.

The C2 style does not make any assumptions about the PLs in which the components or connectors are implemented, whether or not they execute in their own threads of control, their deployment to hosts, or the communication protocols used by them.

3. FRAMEWORK OBJECTIVES

Our primary objective was to create a runtime environment that would simplify the implementation, deployment, execution, monitoring, and evolution of C2-style architectures while preserving the properties implicit in the style. Based on the characteristics of the style and applications for which it is intended, we identified the following to be the key objectives to be met by an architectural framework for C2 applications:

- *Traceability* – The framework should support a simple, faithful mapping between the architectural elements and their implementation. This fidelity is necessary to ensure that the conceptual integrity and key properties of an application’s architecture are preserved by the architecture’s implementation.
- *Platform Independence* – The framework should support component development independent of any PL, operating system (OS), and communication protocol. However, the framework may make implementation choices that would allow for efficient execution of an application on a given platform.
- *Distribution* – The framework should not make any assumptions about the number or location of address spaces in which a system’s components will reside.
- *Dynamism* – The framework should enable modification of a system’s runtime architecture with minimal effect on the execution of existing components [19].
- *Efficiency* – The framework should impose minimal overhead on an application’s execution. The goal is to enable efficient execution of applications on platforms with varying characteristics (e.g., speed, capacity, network bandwidth). Ideally, the framework should permit full use of the power of the execution platform’s hardware architecture, OS facilities, and the PL’s runtime system.
- *Observability* – The framework should support runtime monitoring and analysis of an application. This is necessary for tracking the system performance and correctness, and for anticipating and detecting system faults.
- *Extensibility* – The framework should support design and implementation of new elements compatible with the C2 style. This allows one to tailor the framework, and create and compose components and connectors suitable for specific application needs.

4. OVERALL FRAMEWORK DESIGN

To develop the architectural framework for C2, we (1) identified the elements of the style and their characteristics, and (2) implemented a set of modules in a PL (e.g., classes in an OOPL) to represent the identified elements, their relationships, their base behaviors, and their interactions. An application’s concrete architecture is then created by extending and/or instantiating the appropriate modules from the framework. The framework thus supports direct transformation of application architectures from style elements to implementation constructs.

4.1 Overview

Figure 2 shows the external API of the C2 architectural framework, with classes that are used by application developers to realize a conceptual C2 architecture. *Architecture* records the configuration of its constituent components and connectors. The *Component* class provides primitives to *send* and *handle* (i.e., receive) messages. The *Connector* class keeps track of attached components and dispatches messages to the appropriate components. Components and connectors may run in a common thread of control (*Component* and *Connector* classes), or they may execute in their own threads (*ComponentThread* and *ConnectorThread*

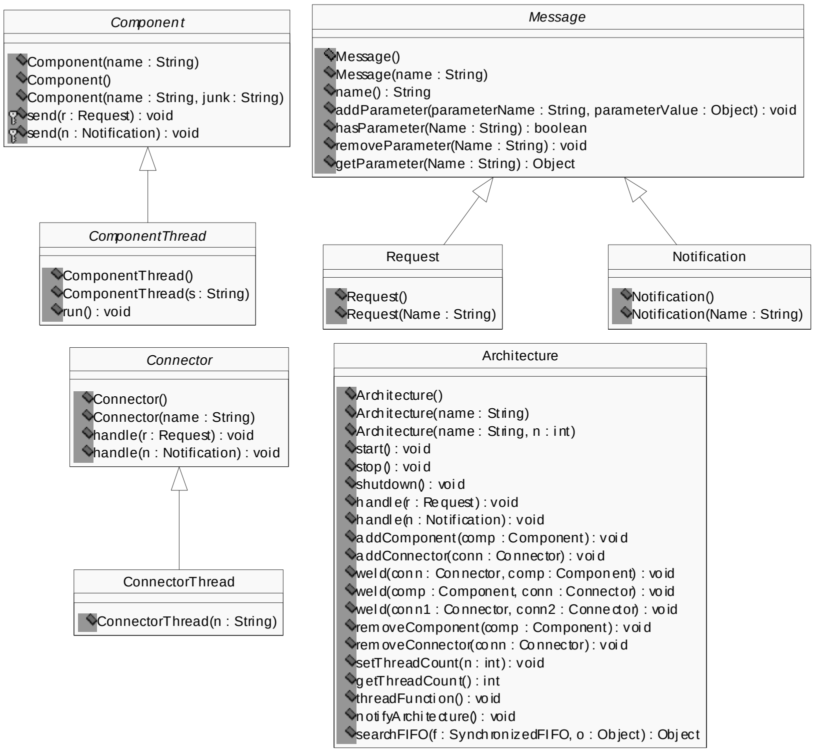

Figure 2. UML class diagram of the external framework API

classes). *Component* and *ComponentThread* classes are abstract. Application-level components are domain-dependent and provide application-specific functionality. Thus, it is necessary to subclass either the *Component* or *ComponentThread* class in order to create application-specific components. On the other hand, since connectors provide application-independent interaction services, the *Connector* and *ConnectorThread* classes are concrete, and may be directly instantiated in an implementation.

In addition to the generic *Connector*, we have developed a library of connectors, such as those supporting various message routing protocols (e.g., broadcast, multicast, unicast), connectors that encapsulate middleware to support distribution of architectural elements [2], and secure connectors. Developers can utilize connectors that provide the application interaction needs from this connector library, and instantiate them directly into the concrete architecture. At times, when additional features are required or certain interaction behaviors need to be amended, the application architect can subclass standard connectors from this library to create the desired connector behavior [17].

As discussed in Section 2, all communication in C2-style architectures is achieved by sending and receiving *Messages* (*Requests* and *Notifications*). The framework provides basic mechanisms for creating and exchanging messages. Components create messages and send them to connectors based on the rules of the C2 style, the components' internal processing algorithms, and communication needs. In response to a message, a receiving connector selects its attached components to which this message should be forwarded based on the connector's internal distribution policy (e.g. broadcast, unicast, etc). Each component implements application-specific processing of the incoming requests or notifications inside the corresponding *handle* method. In case of a connector-to-connector link, the messages are passed to another connector, which applies its own internal policy in further forwarding the messages on to its attached components.

Over the past seven years, we have developed several OO designs of the C2 architectural framework, and have implemented them in several languages (C++, Ada, Java, Python), resulting in a family of framework implementations. Various structural changes have been made to the framework's internal design over time to enhance its performance and improve its extensibility. In order to support portability of C2 applications, we have preserved the external API shown in Figure 2 across framework implementations. This means that an application written for any framework in this family undergoes no changes when run on top of any other C2 framework implemented in the same language. Such cross-compatibility between different versions of the framework allows us to choose the best framework available for an application based on its extra-functional characteristics such as speed, memory utilization, flexibility, and traceability of architectural decisions.

4.2 Tool Support

The process of translating the conceptual architecture of a system, described in the C2 style and the C2SADEL ADL [15], into an implementation is tool supported. The application architecture is designed using the DRADEL and ArchStudio environments [15,19].[1] The environments support graphical design of an architecture, with tools for analyzing and generating a skeleton implementation for that architecture. The generated code contains subclasses of the framework's *Component* and *ComponentThread* classes with application-specific method declarations. Each application component's message passing code is automatically generated from the architecture modeled in C2SADEL. Furthermore, the generated implementation also contains the subclassed *Architecture* class with code for instantiating the appropriate components and connectors and *weld*ing them into the configuration identified in the C2SADEL specification. A developer only needs to implement the method bodies of each component's message handlers.

4.3 Framework in Action

To illustrate the usage of the C2 framework, we introduce a simple banking application, shown in Figure 3. This application supports various types of accounts and customers, and provides account access via multiple tellers and ATMs. The application performs daily interest updates, and supports withdrawals, deposits, balance queries, and transfers.

Figure 4 shows a subset of the Java code for the banking application that uses our framework. The *BankArch* class's *main* method

---

[1] Both DRADEL and ArchStudio themselves are also developed using the C2 style and implementation framework.

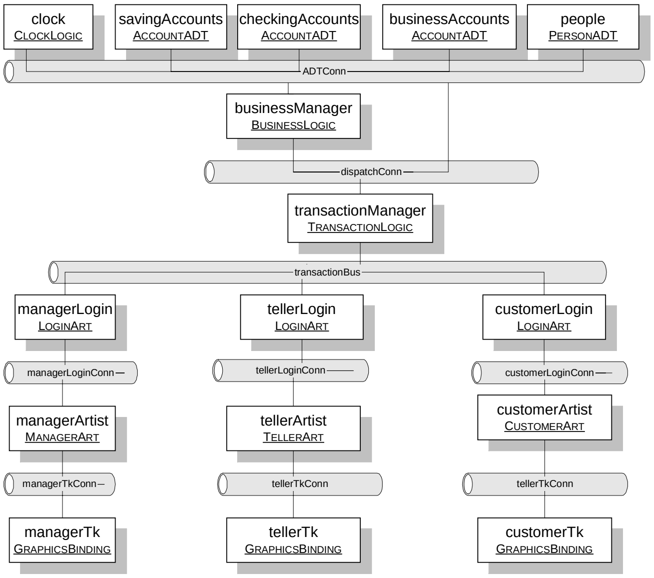

Figure 3. Example C2-style application architecture

calls its superclass’s methods for instantiating components and connectors and composing (*weld*ing) them into a configuration. Figure 4 also demonstrates message-based communication between two components. *AccountADT* creates and sends a notification when a new account has been added, in response to which the *BusinessManager* sends a request to associate the new account with an existing customer. Messages need not identify the recipient components. The recipient components of a message are uniquely defined by the topology of the architecture and filtering policies of the employed connectors. For example, if *ADTConnector* in Figure 3 has a broadcast policy, it forwards the messages to all its attached components, of which only the interested components will process the message and, possibly, take follow-on action.

Bank Architecture initialization

```
class BankArch extends Architecture {
   static public void main(String argv[]) {
       BankArch arch = new BankArch ("BankArch");

       // create components here
       ClockLogic clock = new ClockLogic ("clock", speed);
       ...
       // create connectors here
       Connector ADTConn = new Connector("ADTConn");
       ...
       // add components and connectors to the architecture
       arch.addComponent(clock);
       arch.addConnector(ADTConn);
       ...
       // establish the interconnections
       arch.weld(clock, ADTConn);
       ...
       arch.start();
   }
}
```

Account ADT sends a notification

```
n = new Notification (this.accountAddedStr);
       n.addParameter("number", newAccountNo);
       send (n);
```

Business Manager handles the notification and sends a request

```
public void handle(Notification n)
{
...
if (n.equals(this.accountAddedStr)) {
      Request r = new Request(this.linkAccountStr);
      r.addParameter("number", n.get(“number”));
      r.addParameter("owner", owner);
      send(r);
    }...
}
```

Figure 4. Bank application implementation

5. EVOLUTION OF THE FRAMEWORK

5.1 Early Framework Design

The initial C2 framework[2] was designed and implemented in C++. The class diagram of the C++ C2 framework, shown in Figure 5, contained a class for each C2-style concept. Each component maintained its own buffers of incoming and outgoing messages through its attached *Ports*, and connectors copied messages from the output ports of source components to the input ports of all appropriate destination components. It was the responsibility of components to retrieve incoming messages from their input ports and process them. The connectors were very simplistic and could only perform broadcast distribution of messages. Messages were recorded as comma-delimited strings. Although messages were simple, the framework’s approach to managing them resulted in high heap memory usage, primarily due to each message being copied in multiple ports during an application’s execution. Since *Architecture* was a subclass of *Component*, creation of composite connectors was not directly supported. In addition, support for dynamic changes and runtime monitoring of the architecture was not provided.

An Ada implementation of the framework followed, along with the use of off-the-shelf (OTS) software interoperability mechanisms inside connectors. These mechanisms were employed for supporting the interactions of components deployed in different OS processes, on different machines, and possibly implemented in different PLs (in this case, Ada and C++) [2,12].

A Java-based implementation of the same OO design was created to take advantage of Java’s support for dynamic class loading, and to enable *observability* of the runtime architecture. Dynamic loading was leveraged in enabling runtime addition, removal, replacement, and reconnection of components [19]. Monitoring

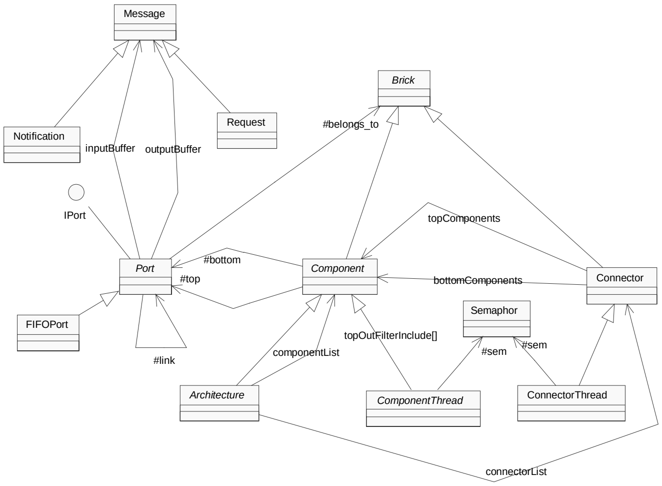

Figure 5. ***C2*** **framework design class model**

---

[2] This version of the framework will be referred to simply as “C2” in the remainder of the paper, except for those cases where the framework needs to be disambiguated from the C2 style.

support was added to the Java *C2* framework to track and possibly filter the messages flowing through component and connector ports. In order to enable more effective and efficient processing of messages, the messages in the Java framework were stored as hash tables. This allowed parameter retrieval by key rather than by position, as was necessary in the C++ and Ada frameworks' string-based messages.

This design of the framework provided a single, shared thread for processing messages in all components that did not run in their own thread of control (i.e., for all subclasses of the framework's *Component* class). The *Architecture* class managed scheduling of message processing inside such components in a round-robin fashion. This design resulted in a low degree of parallelism, as components had to wait for their time slice in order to process independent messages.

The initial implementation of the framework satisfied several of our objectives. It supported *platform independence* by adopting a layered approach, as shown in Figure 6. The framework, depicted in the bottom plane, hid all the platform specific details as well as the mechanism for achieving communication between components, rather than exposing them to the architectural elements and, thereby, to the developers. In this sense, the framework plays the role of a middleware. Additionally, the framework abstracted several platform-specific services into components. For example, in Figure 3, the *GraphicsBinding* component provides user interface services. *GraphicsBinding* issues requests each time a user interface event occurs and receives notifications to display information. In a number of C2-style applications built to date, cross-platform portability is aided by interchanging different *GraphicsBinding* components.

The framework achieves *distribution* by encapsulating aspects such as threads and inter-process communication (IPC). The framework provides a single "shepherd" thread for those components that do not have their own thread of control. When such components receive a message, the encapsulating architecture's thread is typically run through their *handle* method by the framework. All IPC is abstracted away inside special-purpose connectors, called *border connectors* [2] available from the connector library.

Dynamism in our framework is supported through operations for addition (*add*), removal (*remove*), connection (*weld*), and disconnection (*unweld*) of components and connectors in the *Architecture* class. Since the interaction among components is decoupled via flexible, first-class connectors (as mandated by the C2 style), a high degree of dynamism is provided with minimal disturbance to the rest of the running system [19].

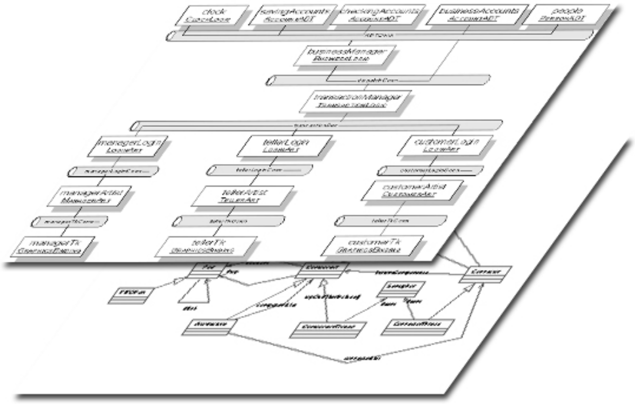

Figure 6. Layered construction of an application using the C2 implementation framework

Finally, due to the external API of the framework, which directly reflects the C2 style concepts (recall Figure 2), it is possible to *trace* the relationship between a conceptual architecture and its concrete implementation. In addition, tools such as DRADEL and ArchStudio further simplify an architect's job of keeping the architectural model synchronized with the implementation via automated code generation.

At the same time, the remaining objectives (*efficiency*, *observability*, and *extensibility*) were not met as successfully by this, initial implementation of the framework. The guiding principle behind this implementation was to ensure the maximum fidelity of an application to its architecture. While directly aiding traceability, the principle resulted in the increased framework weight (e.g., by duplicating the same message object in all recipient components' incoming queues) and slower speed (as evidenced by the performance numbers shown in Table 1). Similarly, there were no facilities in the framework to monitor an architecture at runtime, or to extend the framework itself without significant redesign. To achieve these goals, several modifications to the framework were undertaken, as discussed below.

5.2 Framework Optimizations

Recently, we have begun applying our architecture-based development support in the emerging area of hand-held, mobile, possibly embedded, and resource-constrained execution environments. To that end, we have had to carry out a number of enhancements to the *C2* framework, resulting in the *eC2* ("embedded" C2) framework. As stated above, we have identified that it is difficult to simultaneously maximize fidelity and efficiency in the framework. In the *eC2* framework, *efficiency* became our primary focus. We replaced component *Ports* that maintained private message queues, with a central FIFO message queue per each address space.[3] This change slightly decreased the fidelity of the framework to the style (since, internally, each component has a pointer to the central FIFO message queue), but it greatly increased the framework's performance. A pool of *shepherd* threads is kept ready to handle any messages sent by any component in a given address space. The size of the thread pool is parameterized and, hence, adjustable. For communication that spans address spaces or machine boundaries, the message is transported via a border connector to the recipient address space, and added to its message queue. A shepherd thread removes a message from the head of this queue as soon as it finishes processing the previous message. The shepherd thread is run through the connector attached to the sending component; the connector dispatches the message to relevant components using the same thread of execution for processing their *handle* methods (see Figure 7). If a recipient component generates further messages, they are added to the end of the message queue, and different threads are used for dispatching those

---

[3] Although the C2 style does not allow assumptions to be made regarding the address space of an element, it allows us to take advantage of the local implementation environment conditions to optimize performance [31].

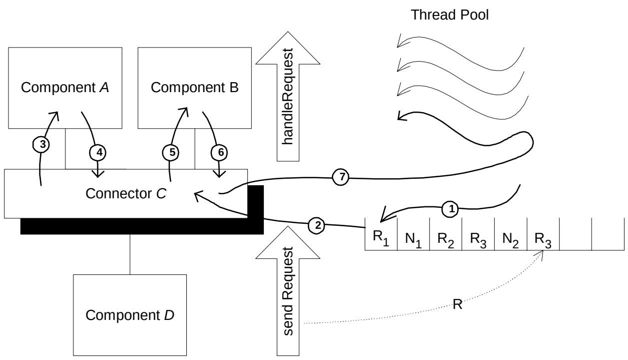

Figure 7. Message dispatching in *eC2*

messages to their intended recipients. An alternative design allows separate threads to be used for dispatching a message from the connector to each intended recipient component, thus increasing the parallelism in the architecture. The control over the thread pool and the message queue is exercised from the *Architecture* class in the eC2 framework. Unlike the original C2 framework, each message exchanged between components in the same address space in eC2 is accessed by reference, rather than by copy.

The requirement for supporting small devices (e.g., the Palm Pilot™ V_x, which limits the use of dynamic heap memory to 256 KB) led us to performing further optimizations on the framework. We observed that a large amount of dynamic system memory usage is a result of the exchange of messages among components and connectors. Since we had already eliminated message copies, we further optimized memory usage by adopting more memory-efficient data structures for storing messages. The eC2 framework initially replaced the original framework's synchronized hashtables with a dynamically linked list. This was later replaced by a fixed-sized, circular array of messages. This design reduced overall heap memory usage and placed an upper bound on the memory required. At the same time, the concurrency management of the circular array slightly reduced the speed of processing as compared to the linked-list solution: the producer-consumer algorithm was applied to keep the message production under control, and supply shepherd threads with a constant stream of messages to process.

Note that, as long as the rate of production of messages is maintained at or below the rate of processing them, a small finite message queue will suffice. However, this is not always possible: in some applications the rate at which messages are generated will exceed the rate at which they are processed. There are at least three possible solutions to this problem: (1) instantiation of more *shepherd* threads, (2) temporary assignment of higher priority to components that consume more messages than they generate, and (3) selective dropping of messages when the average queue size grows faster than the rate at which messages can be processed (assuming the application is able to adapt to dropped messages). We are currently looking into these alternatives as a way of minimizing the memory utilization in the applications running on top of the eC2 framework.

5.3 Framework Extensibility

The original *C2* framework did not provide adequate support for composing richer architectural elements using simple components and connectors. *Architecture* subclassed the *Component* class, which meant that a composite could be used in place of a simple component. However, one could not easily use a composite in place of a connector. Thus, composition of elements was asymmetric, although the C2 style itself is symmetric in this regard. Moreover, the optimizations performed in *eC2* had constrained the *flexibility* and *observability* of our framework by tightly coupling framework classes with the shared message queue and the shepherd thread pool. To overcome these difficulties, we decided to restructure the framework design and reduce the coupling between framework classes. While retaining most of the performance optimizations of the *eC2* framework, this resulted in a highly *extensible* framework, *xC2* ("extensible" C2).

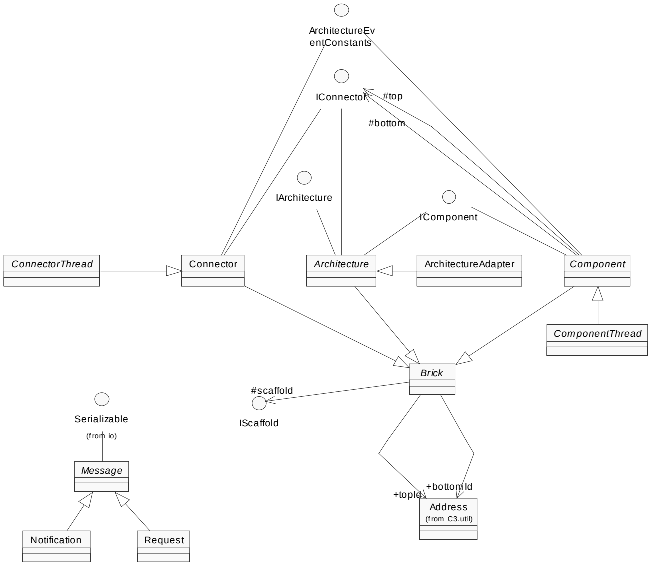

Figure 8. *xC2* framework design class model

Figure 8 shows a UML class model of the *xC2* framework. *Component*, *Connector*, and *Architecture* are all sub-classed from *Brick*. Each subclass of *Brick* implements a specific set of interfaces based on the type of architectural element to which it belongs. *IArchitecture* provides methods for managing the architectural configuration; *IConnector* provides methods for distributing messages; and *IComponent* provides the methods for processing messages. *Brick*s are attached to an *IScaffold* for providing execution and monitoring support. A class that implements *IScaffold* can selectively monitor messages flowing through the architecture based on their content, directly aiding architecture *observability*. Scaffolds are also used to store messages and pool threads so that message dispatching can be done in a way most suitable to the application. This also allows us to separate the management of threads and messages from the *Architecture*, allowing one to easily compose many sub-architectures in a single application.

The *xC2* framework is symmetric in its support for components and connectors. This has allowed us to create a set of reusable connectors, with complex C2-style internal architectures. For example, we have composed a security connector to authenticate communicating components. We have also created modular "border" connectors to allow components across virtual machine boundaries to communicate with each other, synchronous message connectors with procedure call-like semantics, and multiversioning connectors to support reliable runtime upgrades of components [23].

We have been able to support various distribution techniques such as broadcast, multicast, and unicast by assigning an *Address* to each *Brick*, and then using these addresses for more targeted distribution of messages. The address is either claimed by a component during architecture initialization, or the connectors attached to a component assign an address for each connected component. The addresses are used during communication for identifying component(s) that are intended to receive messages. It becomes the responsibility of sending components to specify message targets. This can be accomplished in a variety of ways: (1) hardwiring addresses into each component during development, (2) using a registry to record component types, and querying the registry to locate the target components; (3) broadcasting an initial message aimed at discovering the components with which communication is desired, and later using the discovered addresses for unicast distribution. We are currently investigating all three alternatives, in terms of the tradeoff between performance and degree of component coupling.

Support for dynamism, discussed in the context of the Java C2 framework in Section 5.1 and subsequently carried over into eC2, is taken a step further in xC2 by supporting component mobility. The framework provides connectors that migrate components across machine boundaries by leveraging the architecture's and components' *IScaffold* interfaces [13].

6. DISCUSSION

Together, the frameworks discussed above in Sections 4 and 5 address all of the objectives outlined in Section 3. However, individually, each framework covers only a subset of the stated objectives. One reason is that several of the objectives conflict. For example, observability can be achieved only at the expense of decreased framework efficiency. Similarly, platform independence may hamper framework extensibility (e.g., by disallowing platform-specific extensions) and dynamism (e.g., some implementation platforms, such as Ada, typically do not support dynamic class loading).

Table 1. Properties of Java C2 framework implementations[4]

Framework	*C2*	*eC2*	*xC2*
SLOC	2000	1800	1500
Time (sec), 1 thread	1025.3	2.2	3.8
Time (sec), 10 threads	87.1	2.7	4.2
Time (sec), 50 threads	N/A	3.0	4.4
Time (sec), 50 threads, 50 components	237.4	4.7	13.0
Memory usage (bytes)	5112	1400	2376
Other PL support	C++, Ada	Java KVM, EVC++	Java KVM, EVC++, Python
Flexibility	Low	Medium	High
Traceability	High	Low	Low

For illustration, Table 1 shows a summary of several properties of the different versions of C2 framework's Java implementations. Two simple applications were used for obtaining the benchmarks: one consisted of a single connector and two components (one above and one below the connector), while the other consisted of 50 identical components above the connector and one component below. The benchmarks were performed on an Intel Pentium III 500 MHz processor with 256 MB of RAM running JDK™ 1.4 beta 2 on Microsoft Windows 2000. In both cases 100,000 simple (parameter-less) messages are sent by the bottom component to the top component(s). *Time* reflects the amount of time required to complete the exchange of messages. *Memory usage* was recorded at the time of architecture initialization, and it indicates the amount of memory consumed by the framework and the first application's two components and one connector when used with one thread. Although all measurements are for the framework implementations in Java, the benchmarks are representative metrics for comparing the respective qualities of the different framework *designs*, which have been implemented in multiple PLs.

The original C2 frameworks (implemented in C++, Ada, and Java) is significantly outperformed by the more recent eC2 and xC2. The primary reason is that, with time, our understanding of the style itself and of its reification in a framework increased. For example, when the style was initially formulated, component communication ports were given an important role, despite the fact that each component always had a fixed number of ports (one on the top and one on the bottom) [31]. The original C2 frameworks tried to stay true to this vision, but eventually we realized that, unlike other styles such as Weaves [7], ports do not play an active role in a C2-style architecture. Therefore, explicit ports created unnecessary overhead in implementations and were removed in the subsequent frameworks.

Another major evolution point for the frameworks was their implementation of threading. The original C2 framework was again faithful to the conceptual model formulated in the style, where threads are associated only with the individual components and connectors. However, this implementation turned out to be inefficient for various reasons. One reason was that the frameworks did not make any provisions for the fact that some components in an architecture will generate many more messages than others. Another reason was that message dispatch by a connector was always performed sequentially to each recipient component because the connector had one thread of control. In the subsequent frameworks, the *Architecture* class and implementation of the *Iscaffold* interface, respectively, control the thread pool, allowing for "lending" of threads to a connector and simultaneous dispatching of a message to multiple components (recall discussion in Section 5.2).

The recent implementation of xC2 marks another shift in our understanding of the C2 architectural style and in the emphasis we place on the various architectural elements in general. Specifically, our experience since the development of the initial C++ C2 framework has indicated that in an architectural setting, and specifically in C2-style architectures, software connectors fundamentally influence the key properties of an architecture and of the resulting system [2,29]. xC2 recognizes this and provides the ability to incrementally construct connectors with arbitrarily com-

---

[4] The measurements for *C2* shown against 10 threads are performed with 3 threads, since C2 does not use a *shepherd* thread pool. EVC++ is Microsoft's Embedded Visual C++.

plex internal architectures. This view of complex, compositional software connectors has been gaining support in the software architecture community [17,30]. We are currently directly leveraging xC2 to further investigate this question.

7. RELATED RESEARCH

There exists a large body of research on OO frameworks and middleware. Due to space constraints, we provide only a brief classification in this paper. The research and use of frameworks can be classified into six distinct generations on the basis of the achieved level of component reuse: *(1)* Module interconnection languages [3] enabled the reuse of components implemented in a single programming language (PL). *(2)* Remote procedure calls and platform-neutral data representations (e.g., [1,24]) enabled distribution and reuse across PLs. *(3)* Platform-neutral runtime environments and dynamic component loading (e.g., [6,10]) enabled dynamism and reuse across computing platforms. *(4)* Domain-specific and GUI frameworks (e.g., [8,21]) enabled reuse across applications. *(5)* Provision of infrastructure services such as naming, threading, persistence, and transaction management (e.g., [8,27,35]) introduced the possibility of reuse of architecture-level abstractions. *(6)* Reuse of architecture-level abstractions became an explicit focus of architectural style-based frameworks (e.g., [28,31]). While it exhibits the properties of frameworks spanning several generations, the family of C2 frameworks is most closely related to the sixth generation.

8. CONCLUSIONS AND FUTURE WORK

Over the past decade, software architectures have been touted as a possible answer to many of the problems inherent in engineering large, complex, distributed, long-lived software systems. The many architectural styles, modeling notations, and analysis techniques that have emerged from the software architecture research community during this period have given developers *conceptual* tools with which to attack these problems. However, these architectural approaches have consistently failed to address the relationship between the abstract architectural models and concrete system implementations [16].

On the other hand, a number of software interoperability technologies have emerged primarily, though not exclusively (e.g., [11,22,25]), from industry [27,32,35]. These technologies provide solutions for composing *implementation-level*, coarse-grain software components, giving developers powerful system building tools. However, although it has been shown that they indeed influence the architectural characteristics of systems [4], these technologies rarely explicitly acknowledge the architectural models that typically precede the implementations, or architectural styles that influence the key properties of the implemented systems.

In this paper, we have discussed our attempt at bridging this gap between the model-centric and implementation-centric approaches. We have coupled an explicit architectural style (with its accompanying ADL and analysis tools) with an implementation infrastructure in the form of a collection of OO class frameworks, allowing for a straightforward transfer of architectural constructs and decisions into the running system. In doing so, we have defined a set of object models that underlie a component- and connector-based architectural style, further adding evidence to our previously stated argument that OO and architectural approaches, while not identical, are compatible [14,26]. We have also attempted to account for different situations that might arise during development:

- Different frameworks exhibit different extra-functional properties (e.g., performance, adaptability), allowing the selection of the framework most appropriate for the needs of the given application.
- The frameworks are implemented in different programming languages (multiple flavors of C++, Ada, Java, and Python), allowing developers to select a framework based on their preferred programming language, instead of the other way around.
- The frameworks employ different OTS component interaction mechanisms (e.g., Java RMI [33], CORBA [18], Q [11], Polylith [25]), giving developers the flexibility to select the interaction mechanism with which they are most familiar and/or comfortable, or the mechanism that is the best fit for their current application needs.
- The frameworks leverage their explicit connectors and the OTS interaction mechanisms to enable application development across language and platform boundaries.
- The frameworks also leverage their multi-lingual support and explicit connectors to enable the integration of third-party components that do not adhere to the assumptions of C2 (e.g., asynchronous message-based interaction) into a C2-style application [12].

Our frameworks have been used over the past seven years in the development of over 100 applications. The applications have ranged in size from very small "proof of concept" examples (on the order of 1,000 SLOC) to moderately-sized development environments (on the order of 50,000 SLOC, *not* counting large OTS components such as the Netscape Communicator or Rational Rose, which have been wrapped to be used as C2-style components in the environments). For example, as already discussed, the environments for modeling, analyzing, implementing, and evolving C2 applications (ArchStudio [19] and DRADEL [15]) are themselves implemented according to the rules of the C2 style and using the implementation frameworks presented in this paper. Finally, the relatively small size (about 1,750 undocumented SLOC on average) and light weight of the frameworks have made them well suited for use as a pedagogical tool for introducing and demonstrating the concepts of component-, connector-, and message-based application development. The original Java C2 framework (recall Table 1) has been used at several universities to teach the concepts of architecture-based development; more recently, we have also used the Java KVM, EVC++, and Python implementations of the xC2 framework in two graduate-level courses at USC.

While we have amassed extensive experience with the frameworks and have applied them across several application domains, a number of issues remain open. One key issue is the performance of implemented applications. Our objective of maintaining the traceability of architectural decisions in an implementation results in applications in which components *always* communicate via intermediaries (i.e., connectors). This is desirable in situations in which a given component needs to broadcast information to multiple recipient components, of if the application is expected to evolve at runtime. However, the communication indirection induced by explicit connectors may be overly costly in situations in which the connector mediates interaction between only two components, the components do not require asynchronous message-based interaction, or the application is unlikely to evolve at runtime. We have begun identifying situations such as these, in which, to a large extent, specific optimizations to an implementation may be applied while preserving traceability of architectural decisions. One example such optimization is implemented in our synchronous message passing connectors [23].

We have also begun using the frameworks as a platform for investigating the shortcomings of the C2 architectural style and extending C2's semantics as a result. In particular, we are looking at the semantics of message delivery. Because of its asynchronous nature, C2 has allowed us to adopt any arbitrarily simplistic message delivery scheme. Thus, while they all allow implementing arbitrary delivery schemes, our frameworks have relied on message arrival order to effect FIFO delivery of messages. In the C2 framework, the FIFO scheme is applied at the level of message queues contained in each component or connector port, while in eC2 and xC2 the scheme is applied at the level of messages that span multiple components and connectors in a given address space. This means that even the messages containing migrating components are treated in the same manner as regular messages (i.e., they must "wait for their turn"). We are currently investigating a suitable message prioritization scheme that would enrich the semantics of C2. Such a scheme would enable finer-grain control over code mobility and possibly allow one to leverage message priorities in a way that would ensure synchronous interaction without the need for the implementation of a separate connector and/or a separate message queue.

Our final, on-going research thrust is investigating the suitability of the frameworks as implementation substrates on networks of small, resource-constrained, mobile, possibly embedded devices. While eC2 was implemented specifically for this purpose and our initial results are very promising, this is still work in progress. We are currently investigating the possible role of XML as an enabler for communication across heterogeneous devices (e.g., Palm Pilot and Compaq Pocket PC), as well as the exact nature and role of border connectors in a wireless network. These issues will frame our work in the immediate future. We believe that our current framework implementations form a fertile ground for investigating these issues.

ACKNOWLEDGEMENTS

We wish to acknowledge the contributions of K. Anderson, E. Dashofy, P. Oreizy, S. Phadke, A. Rampurwala, and J. Robbins in the creation of various members of the family of C2 architecture frameworks.

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9985441. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement number F30602-00-2-0615. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

REFERENCES

1. A. Birrell, B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39-59, February 1984.
2. E.M. Dashofy, N. Medvidovic, R.N. Taylor, Using off-the-shelf middleware to implement connectors in distributed software architectures. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, CA, pp 3-12.
3. F. DeRemer and H. H. Kron, Programming-in-the-Large versus Programming-in-the-Small**.** *IEEE Transactions on Software Engineering*, June 1976.
4. E. Di Nitto, D. S. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. *21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
5. G. Fregonese, A. Zorer; G. Cortese. Architectural framework modeling in telecommunication domain. *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles CA 1999.
6. A. Goldberg. Smalltalk-80: The Language, *Addison-Wesley*, 1989.
7. M.M. Gorlick, R.R. Razouk, Using Weaves for Software Construction and Analysis. In *Proceedings of International Conference on Software Engineering (ICSE '91)*, IEEE CS Press, Los Alamitos, Calif.,1991, pp. 23–34.
8. I. F. Haddad. X/Motif Programming. *Linux Journal*. Issue 73 May 2000.
9. R.E. Johnson. Documenting Frameworks as Patterns. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92)*, Vancouver, BC, Canada, 1992.
10. T. Lindholm, F. Yellin. The Java Virtual Machine Specification. 2nd Edition Java Series. *Addison Wesley* 1999.
11. M. Maybee, D. Heimbigner, L.J. Osterweil. Multilanguage Interoperability in Distributed Systems: Experience Report. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany,* March 1996.
12. N. Medvidovic, P. Oreizy, R.N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 International Conference on Software Engineering*, 1997. pp 692 – 700.
13. N. Medvidovic, M. Rakic. Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility. In *Proceedings of the Software Engineering and Mobility Workshop*, Toronto, Canada, May 2001.
14. N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, J. E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*. To appear, December 2001.
15. N. Medvidovic, D. S. Rosenblum, R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, pp.44-53, 1999.
16. N. Medvidovic, R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, Jan. 2000, vol 26(1), p. 70-93.

17. N.R. Mehta, N. Medvidovic, S. Phadke. Towards a Taxonomy of software connectors, *2000 International Conference on Software Engineering*, 2000, Limerick, Ireland.
18. R. Orfali, D. Harkey, J. Edwards. The Essential Distributed Objects Survival Guide. *John Wiley & Sons*, Inc. 1996.
19. P. Oreizy, N. Medvidovic, R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering*, pp.177-186, Kyoto, Japan, April 1998.
20. D.E. Perry, A.L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Note*s, vol. 17, no. 4, pp. 40-52, October1992.
21. J. Prosise. Programming Windows with MFC. *Microsoft Press*, 2nd Edition. 1999.
22. J. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, January 1994.
23. M. Rakic, N. Medvidovic. Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. *Proceedings of the 2001 Symposium on Software Reusability*, Toronto, Canada, May 2001.
24. H.C. Rao. Distributed application framework for large-scale distributed systems, *Proceedings the 13th International Conference on Distributed Computing Systems*, 1993. pp 31 –38.
25. S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, July 1990.
26. J.E. Robbins, N. Medvidovic, D.F. Redmiles, D.S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *20th International Conference on Software Engineering*, April 1998, pp. 209-218.
27. B. Shannon, M. Hapner, V. Matena, et al. Java 2 Platform, Enterprise Edition: Platform and Component Specifications (The Java Series) by *Addison Wesley* 2000.
28. M. Shaw et al. Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering. Vol. 21, no. 4, pp 314-335, April 1995.
29. M. Shaw, D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice-Hall, 1996*.
30. B. Spitznagel, D. Garlan. A Compositional Approach for Constructing Connectors. Submitted to *The Working IEEE/IFIP Conference on Software Architecture*, The Netherlands, August 28-31, 2001.
31. R.N. Taylor, et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE-TSE*. 22(6), 1996.
32. S. Williams, C. Kindel. The Component Object Model: Technical Overview. *Dr. Dobbs Journal*, December 1994. http://msdn.microsoft.com/library/default.asp?URL=/library/techart/msdn_comppr.htm.
33. Sun Microsystems Inc. Remote Method Invocation. http://java.sun.com/docs/books/tutorial/rmi/index.html
34. Case studies of IBM San Francisco usage (http://www-4.ibm.com/software/ad/sanfrancisco/casestudies.html).
35. A Discussion of the Object Management Architecture (OMA) Guide, OMG, 1997.