*A Technique for Documenting the Framework of an Object-Oriented System**

Roy H. Campbell and Nayeem Islam
University of Illinois at Urbana-Champaign

ABSTRACT: This paper presents techniques for documenting the design of frameworks for object-oriented systems and applies the approach to the design of a configurable message passing system. The technique decomposes a framework into six concerns: the class hierarchy, protocols, control flow, synchronization, entity relationships and configurations of the system. An abstract description of each concern is specified using standard notations. Subtyping is used to ensure that the abstract specifications apply to the abstract classes, concrete classes, and instances of the system.

The message passing framework we document with these techniques is general, portable, and efficient. It supports parallel message based applications on both tightly coupled shared memory architectures and loosely coupled distributed memory architectures. The message passing system framework has been coded in C++, runs on the *Choices* operating system, and has been benchmarked on a system of Encore Multimax

*This work was supported in part by NSF grant CISE-1-5-30035 and by NASA grants NSG1471 and NAG 1-163. The second author was supported in part by an IBM Graduate Fellowship.

320 tightly-coupled multiprocessors. The system is being implemented on a network of SUN SPARCstation 2s.

1. Introduction

Frameworks [8] [7] characterize the design of an object-oriented system. A description of a framework serves to document the specification, implementation, use, and reuse of the design and the code. Framework descriptions are commonly found in descriptions of user interfaces. Examples include the Model/View/Controller [14] and Unidraw [19] frameworks. In the *Choices* object-oriented operating system, we have used frameworks extensively to describe the design and behavior of subsystems [4, 3]. We found it hard to specify and document the relationships between the various components of a *Choices* framework. In this paper, we present an approach to describing frameworks that captures many complex design issues. We apply the approach to a description of the framework of an efficient, portable message passing system for *Choices*.

In a recent paper [4, 3], we elaborated more precisely the way in which we used the notion of an object-oriented framework in our design of *Choices*. The paper identified several common concerns that could be addressed by examining an object-oriented system as a framework and we specified those concerns for some example subsystems of *Choices*. However, as we and other authors [13, 11] have remarked, it is difficult to produce a complete documentation of a framework. Descriptions of frameworks are informal. If object-oriented systems are to become easier to describe, design and implement, we must find a better approach to specify a framework.

In practice, there are few notations that support the description of the abstract properties of interacting components in a large object-oriented software system. A notable exception is the work on Contracts [10]. The flow of control diagrams used for describing the Model/View/Controller [14] and Unidraw [19] frameworks are abstract. The control flow of the system and the abstract class protocols are, however, the only aspects of the frameworks that are described.

In this paper, we extend and unify previous techniques for specifying frameworks. Our technique decomposes a framework into six concerns: the class hierarchy, protocols, control flow, synchronization, entity relationships and configurations of the system. An abstract description of each concern is specified using standard notations. Subtyping is used to ensure that the abstract specifications apply to the abstract classes, concrete classes, and instances of the system.

As an example, we describe the design of a message passing system developed for *Choices*. The message passing system is implemented and its behavior has been benchmarked and studied using example parallel applications [12]. The message passing system is efficient, compares favorably with other message passing systems, and is portable. It exhibits the expected object-oriented system characteristics of specialization, customization, and reuse. The performance measurements of the system include many different customizations, specializations and optimizations.

Section 2 discusses related work. We describe the properties of frameworks in Section 3. Section 4 describes how we use types to classify classes, synchronization, control flow and entity relationships. Section 5 introduces the message passing system example and provides sample documentation of each of the six concerns. Section 6 discusses our conclusions and future work.

2. *Related Work*

There has been very little work in understanding how to design, implement and document frameworks. The two most important frameworks designed and implemented have been for user interfaces. These two frameworks are the Model/View/Controller framework for building user interfaces for Smalltalk-80 applications and the Unidraw framework for building graphical editors.

Most descriptions of the Model/View/Controller paradigm are informal [14]. A method invocation diagram is used that shows the three objects in the system. Arrows indicate method invocations in the diagram. The sequence of method calls is informally specified. The Model/View/Controller paradigm defines how different Models of an application domain, different Views of data and different input Controllers can be composed if certain protocols are obeyed between the three components. They do not consider extending the method invocation diagram to more complex frameworks for user interfaces. For example, new input and output technologies may require a new framework that may be able to use parts of the Model/View/Controller in a disciplined manner.

Contracts [10], on the other hand, do provide a semi-formal method for specifying the obligation of objects to each other. Much like our work they extend the usual type signatures to include constraints on behaviors which capture the behavioral dependencies between objects. Contracts may be composed with other Contracts and they may also be refined. Contracts have been used to describe the interaction of the components in the Model/View/Controller framework. For example, Contracts can express the fact that a View must always reflect the current state of the Model. A Contract defines preconditions required to be satisfied by objects in order to participate in a framework and the invariants that these objects must satisfy. Contracts are limited in that they only look at one aspect of the relationship between objects. It does not, however, make control flow relationships as explicit as a control flow diagram. It does not explicitly state entity relationships between instances of classes and it does not specify the synchronization constraints in objects.

The description of frameworks in Unidraw [19] is also limited. The authors of that framework employ method (interface protocols) tables for each class, and flow of control diagrams to elucidate the behavior of the system. They do not consider how to create abstract control flow diagrams or how to reuse control flow. Their use of ordered control flow is an improvement over the previous uses of control flow [14].

Deutsch [8] defines some simple properties of a framework. He maintains that subclasses in a framework should extend and not invalidate the specification (interface protocols) of their superclass. This means a subclass may re-implement a method but it cannot change the interface of a superclass method or remove a superclass method. This type of programming produces more maintainable and reusable code. It also provides full substitutability: a subclass may be used when a superclass is expected.

3. *What is a Framework?*

A framework is an architectural design for object-oriented systems. It describes the components of the system and the way they interact. In frameworks, abstract classes define the components of the system. The interactions in the system are defined by constraints, inheritance, inclusion polymorphism, and informal rules of composition. *Choices* is defined as a framework that guides the design of subframeworks for subsystems.

The framework for the system provides generalized components and constraints to which the specialized subframeworks conform. The subframeworks introduce additional components and constraints and subclass some of the components of the framework. Recursively, these subframeworks may be refined to further frameworks. Frameworks simplify the construction of a family of related systems by providing an architectural design that has common components and interactions. An instance of a framework is a particular member of the family of systems. Frameworks may be refined and composed with other frameworks.

In order to present and document the message passing system framework of *Choices,* we will use existing notations to describe its constraints, inheritance, inclusion polymorphism, and informal rules of composition. We will present the design of a framework using:

1. Abstract classes and class hierarchies.
2. Abstract and concrete class protocols (interfaces).
3. Flow of control between abstract and concrete classes.
4. Synchronization constraints expressed through path expressions on method invocations on objects.
5. Entity relationship models between abstract classes and instances of concrete classes.
6. Constraints based on static configurations of the framework.

We use *extended subtyping* in discussing control flow, synchronization and entity relationships. It is possible to compose and refine control flows, entity relationships and synchronizations constraints. This leads to design *reuse*.

4. Typing and Documenting Behavior

Our documentation of an object-oriented framework records the implementation of the framework using only those definitions and objects introduced by abstract classes. We propose that we can type the classes in the framework with respect to control flow, synchronization and entity relations. We assert that each of these properties imposes a subtyping relationship between an abstract class, its subclasses, and its instances. Such a subtyping relationship simplifies documentation. In terms of software engineering design methodology such subtyping could be implemented as a set of guidelines that must be adhered to during system design and coding. However, an abstract class may be a generalization of several concrete classes rather than an abstract data type produced by stepwise refinement. Enforcing such a methodology is a difficult problem and is not addressed in this paper.

In the subsequent section, we use this approach in a slightly more formal description of the aspects of the framework we are intent on documenting.

SUBTYPING AND CLASSES The set of components in the framework F is defined as a collection $\{M_f | f \in F\}$. We assert that there is a function *Abs* that maps each component M_f into its abstract class $Abs(M_f) = M_c$. Each abstract class M_c is the root of a class hierarchy defining s concrete subclasses M_{c_s}. Each abstract class M_c exports a set of p methods $M^e_{c_s}$ that are inherited by each subclass M_{c_s} as methods $M^p_{c_s}$ and, in general, $\forall p$, $M^p_{c_s} \equiv M^p_c$. A concrete class can be instantiated to create multiple instances $M_{c_{s_i}}$. Here, the assumption is that if two methods have the same signature [20], they also share a similar implementation behavior. In C++ there may be additional methods or parameters defined for subclasses. These additional methods will be inherited by further subclassing and invoked by objects in the framework at run-time. We assume that for the purposes of describing the framework, they do not need to be represented in the specifications of more abstract classes. If they do then this class should be made an abstract class as it represents a new *abstraction* in the system.

We specify the type of a class M_c as $class_{type}$ (M_c) and for our C++ implementation assert this is equivalent to M^p_c. The types in the framework form a complete partial order [18]. The types of subclasses are similarly defined.

Each of the abstract classes is at the root ("bottom") of a class hierarchy. Each subclass is more specialized than its superclass, and contains more implementation information. Each superclass and subclass are ordered by a subtype which induces a partial order on the class hierarchy. The leaves of the hierarchy are concrete classes, which are also the least upper bounds of the complete partial orders.

Given a class hierarchy A, with classes a, a' where a is a superclass of a', we assert that by construction the following relationship holds: $class_{type}\ (a) \sqsubseteq_{classtype} class_{type}(a')$. More importantly, $\forall s$, $class_{type}\ (M_c) \sqsubseteq_{classtype} class_{type}(M_{c_s})$. We refer to this as signature based inheritance. In C++, the compiler offers some help to check this sub-typing.

4.1 Extending Subtyping to Aspects of Behavior

In this section, we apply the notion of subtyping to the control flow and synchronization between method invocations on objects and the entity relationships between objects.

SYNCHRONIZATION We specify the abstract sequences of method calls to instances using a variant of the path expression notation [2]. Each abstract class M_c has a path expression, $path(M_c)$ specifying the permitted sequences of method executions of its methods M_c^p. The path expressions for subclasses are similarly defined. We define the type of a path expression, $path_{type}\ (M_c)$, for a class M_c as the set of possible traces of method executions of M_c^p. An abstract class is a generalization of its concrete classes, and attempts to capture all the path expressions specifications of its concrete classes.

In the following, we are going to assume that the execution trace of the inherited methods $M_{c_{s_i}}^p$ of an object $M_{c_{s_i}}$ instantiated from a subclass M_{c_s} should be a member of $path_{type}(M_c)$. In order to structure the design of the framework and to simplify its description, by construction we restrict the C++ implementation of the framework to maintain the following type invariant between superclass and subclass: Given a class hierarchy A, with classes a, a', where a is a superclass of a', $path_{type}(a) \sqsubseteq_{pathtype} path_{type}(a')$. More importantly, $\forall s$, $path_{type}(M_c) \sqsubseteq_{pathtype} path_{type}(M_{c_s})$.

Since subclasses may add more methods, new method execution sequences may be specified for these *new* methods. As a result, the subclasses may have more possibilities for method execution sequences than their abstract class. However, our notion of subtyping requires the execution sequences of the methods inherited from the abstract class to include those specified by the abstract class. It specifies nothing about the executions of methods that were introduced by subclasses of the abstract class.

CONTROL FLOW We represent control flow as a connected graph in which a vertex is an object and an edge is a message that is sent from one object to another object. We will refer to a control flow graph as a control flow diagram. In C++, sending a message corresponds to the transfer of control from one method to another. Briefly and informally, we can define the type of a control flow diagram for a class M_c as the set of control flow edges $CF_{type}(M_c)$ flowing into and out of an instance of the class. Again, given a class hierarchy A, with classes a, a', where a is a superclass of a', $CF_{type}(a) \sqsubseteq_{CFtype} CF_{type}(a')$. Similarly, $\forall s$, $CF_{type}(M_c) \sqsubseteq_{CFtype} CF_{type}(M_{c_s})$. The set of edges associated with the inherited methods of an instance of a subclass must include the edges associated with the methods of an instance of its superclass. The subclass may introduce new flows of control associated with new methods, but these edges will not be documented by the control flow specified for the abstract class.

ENTITY RELATIONSHIPS The notion of types is similarly introduced for entity relationships. An abstract class is a generalization that attempts to capture all the possible entity relationships between one abstract class and another. A subclass may introduce further relationships. The type of an entity relation of an abstract class M_c, $ER_{type}(M_c)$, is an ordered tuple denoting the associations a class has with all other classes. Each entry in the tuple has values, 1 (one-to-one), 0 (no relations), and N (one-to-many). The following total order exists between the values $0 \sqsubset 1 \sqsubset N$. We assert that the type of an entity relation of a subclass is a sub-type of its superclass. Since a subclass may introduce other relations we assert that the following holds: Given a class hierarchy A, with classes a, a' where a is a superclass of a', $ER_{type}(a) \sqsubseteq_{ERtype} ER_{type}(a')$. $\forall s$, $ER_{type}(M_c) \sqsubseteq_{ERtype} ER_{type}(M_{c_s})$.

4.2 Typing and Frameworks

In this section, we define the class type, path expression type, control flow type and entity relation type of a framework F.

- The class type of a framework F is a tuple whose elements are of the form $class_{type}(Abs(M_f))$ for $\forall f \in F$. The set of all such tuples is the cross product $\times_f class_{type}(A)$ where $A \in Abs(M_f)$ *or* $subclassOf(Abs(M_f))$.
- Similarly, for path expressions, control flows, and entity relationships we define tuples whose elements are of the form $path_{type}(Abs)(M_f))$, $CF_{type}(Abs(M_f))$, and $ER_{type}(Abs(M_f))$ and cross products $\times_f path_{type}(A)$, $\times_f CF_{type}(A)$, and $\times_f ER_{type}(A)$, respectively.

We extend the definitions from the previous section to describe frameworks. However, for brevity, we state only a simple framework relationship for types. Given two class hierarchies A and B, the set of all possible types of instances of frameworks using A and B is: $\{< class_{type}(a), class_{type}(b) > \mid a \in A \text{ and } b \in B\}$ partially ordered by the

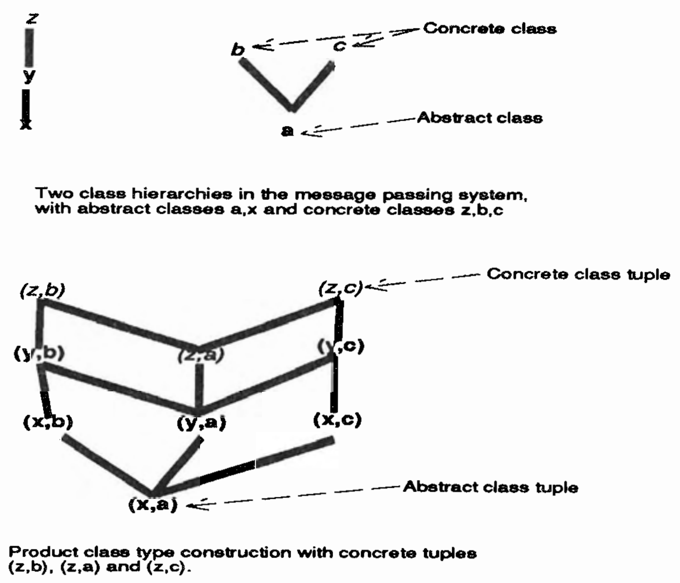
z
y
x
b
c
Concrete class
a
Abstract class
Two class hierarchies in the message passing system, with abstract classes a,x and concrete classes z,b,c
(z,b)
(z,c)
Concrete class tuple
(y,b)
(z,a)
(y,c)
(x,b)
(y,a)
(x,c)
(x,a)
Abstract class tuple
Product class type construction with concrete tuples (z,b), (z,a) and (z,c).

Figure 1: Product construction for class types

tuple relation $\sqsubseteq_{A\times B}$: $< class_{type}(a), class_{type}(b) > \sqsubseteq_{A\times B} < class_{type}(a'), class_{type}(b') >$ *iff* $class_{type}(a) \sqsubseteq class_{type}(a')$ *and* $class_{type}(b) \sqsubseteq class_{type}(b')$. Figure 1 shows the result of applying this construction to two simple class hierarchies.

A similar relationship is induced on the path expression, control flow, and entity relationship types. These relations can now be used to describe an instance of the framework in terms of both its abstract classes and concrete classes. In particular, we may now document each class in terms of its type, synchronization, method invocations, and entity relationships. Tuples of these types for all the classes define the type of a framework.

5. *Documenting the Message Passing System Framework*

We have described a scheme for documenting frameworks. In this section, we use the techniques to document the design of a message passing system that we have built in *Choices*.

5.1 Message Passing System Overview

Modern operating systems support distributed computing on local area networks of workstations using message passing systems [5, 17]. Some operating systems provide message passing on shared memory machines [1, 17] for parallel programming. This section describes a subframework for message passing designed to support parallel message-based applications. It describes facilities for creating structured messages and sending and receiving messages on a variety of architectures.

In *Choices* messages are sent to `MessageContainers` that are similar to Mach ports [17]. A message may be sent to a `MessageContainer` in the same address space, in a different address space (or a different protection domain) on the same machine or on a different machines. The message system provides several reliability models including unreliable and "exactly once" message transmission. Different implementations of these models are required depending on the reliability of the underlying network hardware. The message passing system supports applications on the Encore Multimax shared memory multiprocessor and a network of SPARCstations. The software architecture of the system has been geared towards high performance [12]. The components of the message passing system are given below:

1. The MessageContainer
   is a named communication entity for buffering messages. A `MessageContainer` can have multiple senders and multiple receivers. Once a `MessageContainer` has been created, it is registered with a `NameServer` using an appropriate name. A process intending to send a message to a `MessageContainer` must look the name up in the `NameServer`. On lookup, a sender is given a handle called a `ContainerRepresentative` that forwards messages to the `MessageContainer`.
2. The Message System Interface
   is an adaptation layer encapsulating features specific to a particular parallel or distributed programming paradigm. It uses the basic entities in the system to provide a particular style of parallel or distributed programming. For example, we have implemented the entire suite of Intel iPSC/2 message passing primitives [15, 6] by subclassing the framework described in this paper [12] using both kernel level and user level interfaces. These classes provide naming schemes, information about the last message received and broadcast resolution that completes the implementation of the iPSC/2 style message passing system. The Kernel Message System Interface and User Message System Interface support two alternate implementations of the message passing system, the former in the kernel and the latter in user space. In the `UserMessageSystemInterface`, a `send` or a `receive` passes message data through user shared memory, not through the kernel. In the `KernelMessageSystemInterface`, a `send` or a `receive` passes message data through the kernel and the kernel checks any message parameters.
3. The Transport
   class specifies the mechanism that is used to move a message from a sender to a receiver. A local message may be transported

by a separate process or copied by the sender and receiver processes. A remote message is transmitted across the network.

4. The Synchronization
   between processes may be through busy-waiting or blocking.
5. The Data Transfer
   class concerns the data movement strategies used in sending a message. On a shared memory multiprocessor, message sends may be double buffered, single buffered, or passed by reference.
6. The Reliability
   class allows messages to be sent unreliably, with at-least-once semantics, and exactly-once [9] semantics.
7. The Flow Control
   class uses rate based flow control to ensure that the sender and the receiver are not overrunning one another's data buffers.

5.2 Class Hierarchies

Each of the different components of the messaging system are defined by an abstract class and the abstract class is subclassed to provide the different implementations listed in Section 5. Figure 2 shows the class hierarchies for the message passing system. For simplicity, and because it adds little to this explanation, we omit the subclasses associ-

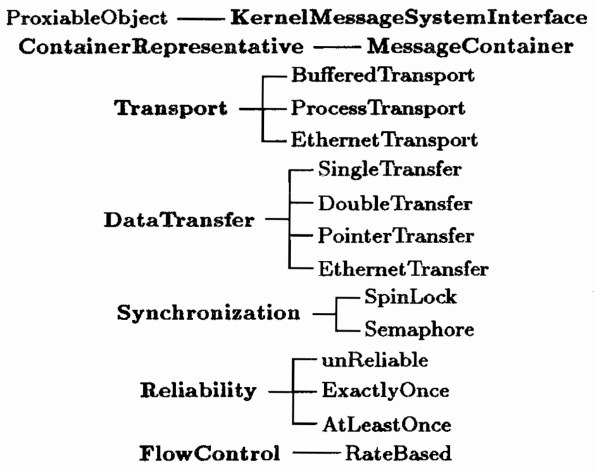

Figure 2: Message Passing System Class Hierarchies

ated with a user level implementation of the message passing system. For similar reasons, we also omit details about proxy objects and how they are used to access objects in the kernel from user space.

The abstract classes in the system corresponding to the components are `Transport`, `MessageContainer`, `ContainerRepresentative`, `KernelMessageSystemInterface`, `UserMessageSystemInterface`, `Synchronization`, `DataTransfer`, `Reliability`, and `FlowControl`.

`ContainerRepresentative` is an abstract and concrete class. It is used to send messages to `MessageContainers`. Similarly, `MessageContainer` is an abstract as well as a concrete class. It is a repository of messages. `Transport` has subclasses `ProcessTransport` that transports messages using a separate process, `BufferedTransport` that provides single or double buffering for a message, and `EthernetTransport` that delivers the message across an ethernet.

`Synchronization` has subclasses `Semaphore` that blocks a process and `Spinlock` that busy waits a process on a shared variable.

`DataTransfer` has subclasses `DoubleTransfer` in which a sender process copies the message into a temporary buffer and a receiver process copies it from that buffer into a receiver buffer, `SingleTransfer` in which the receiver process copies the message from the sender buffer to a receiver buffer in a shared memory region, `PointerTransfer` in which the sender and receiver exchange buffer pointers but message data is not physically copied, and `EthernetTransfer` in which the sender and receiver copy the message data to and from ethernet driver buffer regions.

`KernelMessageSystemInterface` and `UserMessageSystemInterface` support both asynchronous and synchronous communications. By subclassing `KernelMessageSystemInterface` from `ProxiableObject` this interface can cross protection boundaries.

5.3 Interface Protocols

Each abstract and concrete class has a set of methods that it exports as its interface protocol. An interface protocol table is a table that lists all the public methods of a class, the argument types of all these methods and the return types of all these methods. These methods also define the type signature of the class as defined in section 4 and in [20].

Interface Protocol of the Abstract Reliability class		
return value	method name	parameters
void	deliver	Packet *pkt
ErrorCode	deliverWithNotification	Packet*pkt,int len
void	pickUp	Packet*pkt,int len

Table 1: Interface Protocol of Reliability Abstract Class

Interface Protocol of Concrete class AtLeastOnce		
return value	method name	parameters
void	setTimeout	int timeout
void	setNumAttempts	int attempts

Table 2: Interface Protocol of AtLeastOnce Concrete Class

They determine the syntactic legal operations on an object, but say nothing about the semantics or implementation of the object.

Table 1 shows the protocol interface for the abstract class `Reliability`. The interface protocol represents the public methods that are available to instances of a concrete class that implements all the methods. A concrete class must have an implementation for all these methods either through inheritance or its own implementation. Table 2 shows the interface protocol for the concrete subclass `AtLeastOnce` which has two new methods that do not appear in its superclass. This concrete subclass it has not removed or changed the interface of the inherited methods. It has thus extended the interface of its abstract class. Instances of this class will respond to its superclass methods as well the two additional methods that have been defined. A subclass may re-implement a method defined in a superclass without changing the interface.

5.4 Synchronization

Path expressions specify the synchronization behavior of the system. For an object-oriented system, a path expression represents a list of possible *object–method* executions. The following path expressions specify the order in which the methods of the abstract `Reliability` class may be invoked.

path (deliver, deliverWithNotification)* **end**

This expression indicates that only one of the following methods may be invoked at a time: `deliver` or `deliverWithNotification` (notification of errors).

In the concrete subclass `AtLeastOnce`, before either of the deliver methods is called, the `setTimeout` and `setNumAttempts` methods have to be invoked. `setTimeout` and `setNumAttempts` may be called concurrently. The following path expressions define this constraint.

path (setTimeout ; (deliverWithNotification, deliver))* **end**
path (setNumAttempts ; (deliverWithNotification, deliver))* **end**

The superclass restrictions on method invocations still exists, but new restrictions have been defined on new declared methods, in the subclass. These restriction specify the order of method invocations between subclass and superclass but still do not change the constraints specified on the superclass methods. We have thus safely extended the synchronization constraints of a superclass in a subclass.

We have documented the $path_{type}$ of an example abstract class and its concrete subclass in the message passing system. Using our subtyping notation, each class has a $path_{type}$, each class hierarchy has a corresponding $path_{type}$ hierarchy, and the $path_{type}$ of the framework with its abstract classes is of the form $< path_{type}(M_{ContainerRepresentative}), path_{type}(M_{MessageContainer}), \ldots, path_{type}(M_{Transfer})>$. The tuple above has an entry for every component of the framework, and the components are named. In general, subtyping allows one to infer properties about the traces of the method executions that are permitted by the synchronization properties of the framework.

5.5 Control Flow through the Message Passing System

Control flow diagrams can depict the runtime behavior of the system. For an object-oriented system, such a diagram represents an ordering on *object–method* invocations. Again, by following object-oriented design techniques, concrete classes represent implementation specializations of abstract classes. This can be applied to the control flow descriptions. Consider a control flow diagram that specifies an ordering of *object–method* invocations on abstract classes of the message passing system. For it to provide useful and consistent documentation of the framework, the diagram must specify all the control flows that occur between the methods of the message passing system objects. Thus, the diagram must permit the possible control flows associated with the concrete subclasses of the abstract classes. In the *Choices* systems we have studied to date, this form of documentation is helpful and fairly easy to generate. However, it is simpler to present individual examples of the control flow for particular configurations of a system.

Figure 3 shows three abstract control flow diagrams for the message passing system. The top diagram contains the `DataTransfer` class, the second diagram contains the `Reliability` and `FlowControl` classes and the third diagram contains the remainder of the abstract classes of the message passing system. The three control flow diagrams are not connected. The control flow defined in Figure 3 is

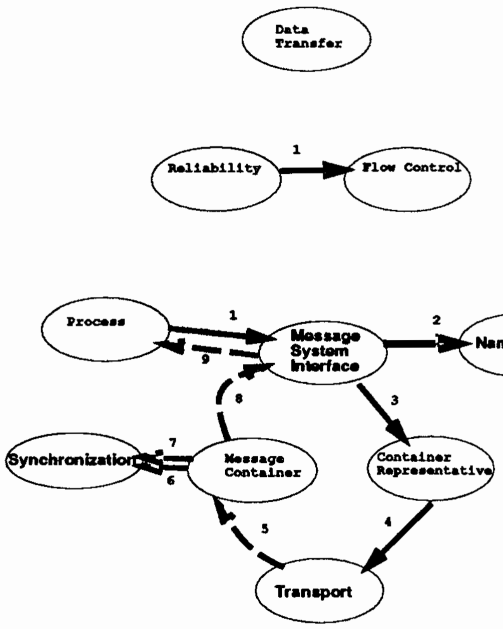

Figure 3: The Abstract Control Flow of a message send/receive pair

abstract, it does not appear in any implementation. Particular implementations refine and compose these three control flows. As a result of composing control flow diagrams the method invocation numbering changes.

For example, Figure 4 shows a concrete control flow diagram for the concrete classes of the message passing system framework for the distributed version of *Choices*. The three abstract control flow diagrams are combined and concrete subclasses replace the abstract classes. In addition, the flow control numbering changes to reflect the control flow in the composition. The flow control diagram for the shared memory version of the message passing system is composed of two of the abstract diagrams since we do not need the flow of control through the `Reliability` or `FlowControl` classes. Therefore, two of the control flow diagrams are *reused*.

Before we describe the control flow in the message passing system, we briefly describe the programming model of *Choices*. `MessageContainers` are created by `ApplicationProcesses`, are associated with a particular `Domain`, and can be registered with a *Choices* `Name-`

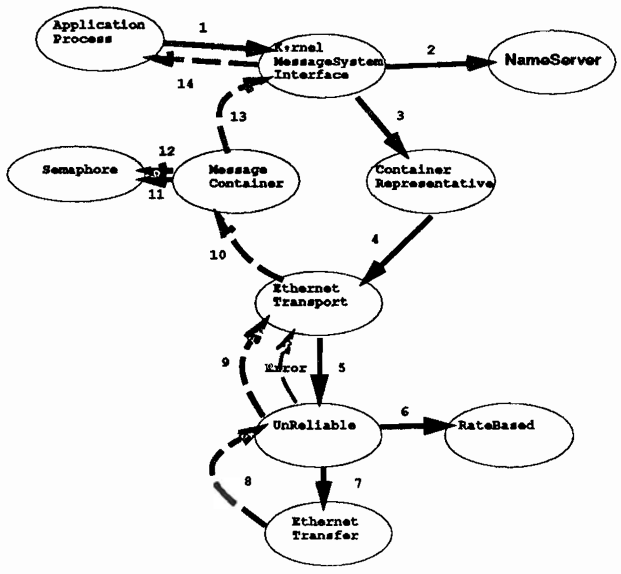

Figure 4: The Concrete Control Flow of a message send/receive pair

Server so that ApplicationProcesses in other Domains and on other machines can send messages to it.

All Processes in the same Domain may retrieve messages from any MessageContainer in its Domain. If a MessageContainer is associated with the kernel Domain, any Process in any Domain on the same machine may retrieve messages from it. However, a particular message system interface may impose further restrictions on which processes can access a MessageContainer. Similarly, a ContainerRepresentative associated with a Domain can be used by any Process in that Domain. Also, if the ContainerRepresentative is associated with the kernel Domain, then Processes of any Domain on the same machine may use the same ContainerRepresentative to send messages.

A process intending to send a message to a MessageContainer may look up its name in the NameServer to get a ContainerRepresentative or it may request that the message system interface does the lookup. The NameServer returns a handle called a ContainerRepresentative. A ContainerRepresentative delivers messages to the MessageContainer for which it is a representative. The ContainerRepresentative has a send method, that invokes the appropriate method on a Transport object.

In this example, a process sends a message through the KernelMessageSystemInterface. A second process then performs a receive on the KernelMessageSystemInterface. The solid arrows show the direction of the flow of control on the sender side and the dashed arrows show the flow of control on the receiver. The thin dashed line, labeled Error, shows the possible redirection of flow of control when a packet is not delivered and a notification is returned.

A detailed account of the concrete control flow trace for a send/receive interaction pair is given below for the distributed implementation of the message passing system. Such traces may also be obtained for each of the abstract control flow diagrams. In the following trace, the numbers correspond to the numbers on the arrows in Figure 4.

1. At the top level an application process makes a call into the KernelMessageSystemInterface to send a message.
2. The KernelMessageSystemInterface looks up the appropriate MessageContainer in the NameServer. The NameServer returns a handle called a

ContainerRepresentative. A ContainerRepresentative delivers messages to the MessageContainer which it represents.

3. The KernelMessageSystemInterface invokes the send method on the ContainerRepresentative.
4. The send method of the ContainerRepresentative invokes the deliver method on the EthernetTransport object.
5. The EthernetTransport generates transport headers and creates a network packet and invokes the UnReliable object to send unreliably the packet across the network. The UnReliable object may fragment the packet to send it across the ethernet.
6. The unReliable object calls the regulate method of the RatedBased flow control object when too many packets are lost.
7. The UnReliable object then invokes the networkSend method on the EthernetTransfer object. This method transfers the packet across the network.
8. The EthernetTransfer object at the receiving machine picks up the data. It passes the packet up to the Unreliable object by invoking the pickUp method on the Unreliable object.
9. The Unreliable object assembles the packet and sends it up to the EthernetTransport object. It generates no acknowledgements.
10. The EthernetTransport invokes the put method on the MessageContainer.
11. The MessageContainer will enter a critical section for queuing messages.
12. The MessageContainer will then exit the critical section for queuing messages.
13. A KernelMessageSystemInterface performs a get on the MessageContainer.
14. The message is then passed onto the application process.

In this section, we have documented an example control flow through the abstract and concrete classes of the message passing system framework. The diagrams are useful because subtyping allows one to infer general properties about the various message passing systems that can be built from the specializations of each class. Using our subtyping notation, each class of the message passing system has a control flow behavior described by its *CFtype*. Each class in the abstract control flow diagram has a CF_{type} and belongs to a CF_{type} hierarchy. Elements of these hierarchies form the tuples that describe the possible control flows of the framework. The CF_{type} of the framework shown is a tuple of the form $<CF_{type}(M_{ContainerRepresentative}), CF_{type}(M_{MessageContainer}), \ldots, CF_{type}(M_{DataTransfer})>$. This is a tuple of the control flow types of all the abstract classes. Comparing Figures 3 and 4, the concrete control flow diagram has only added new control flows and has not removed any from the abstract control flow diagram. This conforms to the subtyping relationship defined in section 4 for control flows.

5.6 Entity Relationship Model

Entity relationship diagrams [16] show quantitatively the relationships between various instances in a system. They may be used in a similar way within a framework. Figure 5 shows the notation we will use to represent one-to-one and one-to-many relationships. A single bar denotes a one-to-one relationship, and a three-pronged fork indicates a one-to-many relationship. These relationships may be mandatory (denoted by an additional line) or optional (denoted by a additional small circle). In Figure 5, **A** shows a one-to-one optional connection, **B** shows a one-to-one mandatory connection, **C** shows a one-to-many optional connection and **D** shows a one-to-many mandatory connection. Again, exploiting the design of *Choices* using object-oriented techniques we may make entity relationship diagrams between abstract classes that describe the entity relationships between concrete subclasses and instances.

Figure 6 shows the abstract entity relationship diagrams for the message passing system. It is important to note that the entity relationship diagram itself is abstract in the same sense that classes may be abstract. The figure contains three separate diagrams. Each connected graph forms a complete diagram. The top diagram contains the `DataTransfer` class, the second diagram contains the classes `Reliabil-`

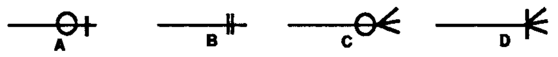

Figure 5: Description of Links between Classes

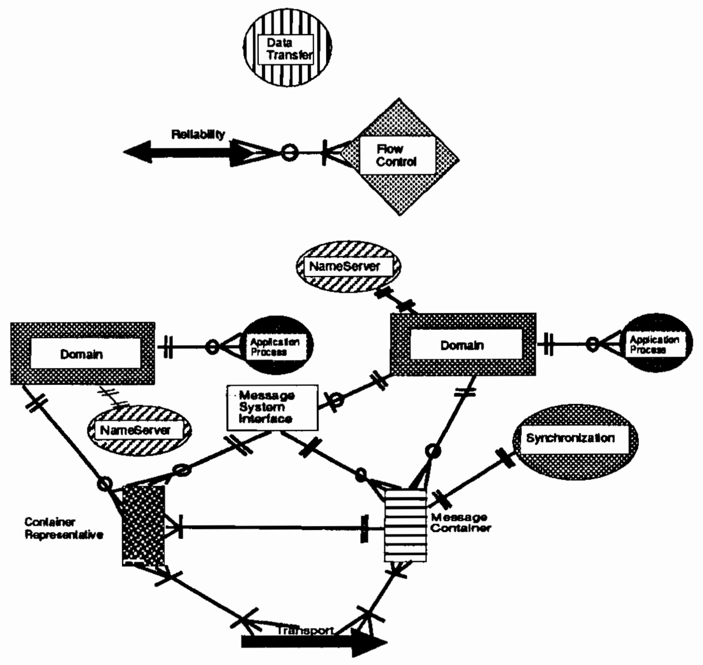

Figure 6: Abstract Entity Relationship Diagrams for the Message passing System

ity and FlowControl and the third diagram contains the rest of the classes. These entity relationships may be combined and refined for implementations of the message passing system framework.

Figure 7 is a concrete entity relationship diagram for the message passing system framework for a distributed implementation of *Choices*. The framework permits combinations of the subclasses of the abstract classes to be used to create a specific message passing system. All the subclasses are concrete. This entity relationship diagram is a refinement and a composition of the three abstract entity relationship diagrams. In particular, all the three diagrams have been joined and the classes have more relationships than their abstract superclasses. A concrete entity relationship diagram for the shared memory version of the message passing system would only use two of the three diagrams since there is no need for the Reliability and FlowControl classes. The other two diagrams are, however, *reused*.

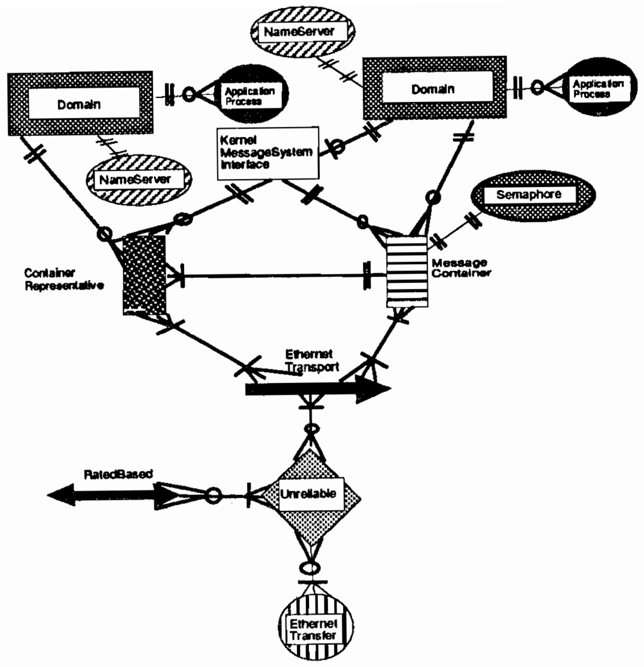

Figure 7: A Concrete Entity Relationship Diagram for the Message Passing System

For example, a collection of instances of the classes, `Ethernet-Transport`, `KernelMessageSystemInterface`, `MessageContainer`, `ContainerRepresentative`, `Semaphore`, `Ethernet-Transfer`, `UnReliable` and `RateBased` defines a message passing system that is kernel based, provides synchronization through semaphores, and sends messages unreliably through the ethernet with rate based flow control.

In the following discussion, we focus on the concrete entity relationships of Figure 7 but also show how the shared memory implementations may be described. Each `Domain` may have a `KernelMessageSystemInterface` or a `UserMessageSystemInterface`[1] and possibly several `MessageContainers`. `ApplicationProcesses` use

1. A `KernelMessageSystemInterface` is used for the kernel `Domain` and a `UserMessageSystemInterface` is used for a user `Domain`.

ContainerRepresentatives to send messages to remote Message-Containers.

Each ContainerRepresentative communicates with only one MessageContainer through the Transport scheme. However, each MessageContainer may have several ContainerRepresentatives that can send messages to it. The ContainerRepresentative provides asynchronous sends. Synchronous message sends and remote procedure call type interfaces may be built from asynchronous sends. Both synchronous and asynchronous gets are supported by the MessageContainer class. When a process attempts to receive an *asynchronous* message that has not yet arrived in the MessageContainer, the method get returns immediately with an *identifier* that can be used later to retrieve the message. If the message is in the MessageContainer, the get returns with the message immediately.

The ContainerRepresentative delivers messages using the EthernetTransport with which it is associated. The MessageContainer buffers incoming messages received from its associated EthernetTransport.

For shared memory, Transport transfers control between the sending process activating ContainerRepresentative and the receiving process activating MessageContainer. This incurs little overhead. Each MessageContainer is associated with a synchronization object, which in this case is a Semaphore. Each Semaphore is associated with exactly one MessageContainer.

For distributed systems, the ContainerRepresentative and MessageContainer and their two associated EthernetTransports are located on different machines. The EthernetTransports use the Reliability mechanism to send the messages through the DataTransfer scheme across the network using the appropriate reliability model. Reliability selects from an appropriate FlowControl policy. Currently, only rate based flow control is used. The example shows UnReliable, EthernetTransfer and RateBased concrete subclasses.

Each class in an abstract entity relationship diagram has an ER_{type} and belongs to an ER_{type} hierarchy. Figure 6 specifies the ER_{type} of the framework which is the tuple $< ER_{type}(M_{ContainerRepresentative})$, $ER_{type}(M_{MessageContainer})$, . . . , $ER_{type}(M_{DataTransfer})>$. This is a tuple of the entity relationship types of all the abstract classes. The diagram is useful because subtyping implies that a relation between any two abstract classes in the framework shown is also a relation between the subclasses of these abstract classes. By comparing Figure 6 with Figure 7 we see that entity relationships have only been added in the concrete entity relationship diagram, and no entity relationships that existed in the abstract diagram have been removed. This conforms to the typing we defined in section 4 for entity relationships.

5.7 Configuration Constraints

Instances of the concrete classes cannot be combined indiscriminately. For example, the abstract transport and transfer classes have concrete classes that correspond to the shared memory and distributed implementations. An `EthernetTransfer` cannot be used with a `BufferedTransport`. To build a specific implementation of a framework for a specific machine, the objects used in the implementation must be instantiated from the appropriate concrete subclass. It is convenient to document this property by associating the abstract classes that have such machine dependent concrete subclasses with a parametrized attribute (machine "type"). Such documentation is a static configuration of the framework. These *configuration* constraints cannot be expressed by the previous techniques. We use Venn diagrams to represent static configurations. Figure 8 shows message passing system abstract classes with particular "types". Those with the **Framework** type may be used without restrictions, those with the **Protection Dependent** type must be used only with implementations

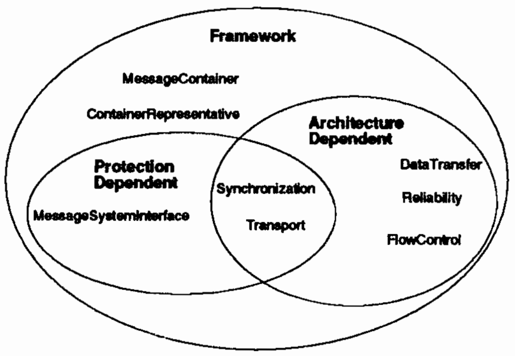

Figure 8: A Venn diagram of the various sets in the framework

for the same type of protection, and those with the **Architecture Dependent** type must all be implementations for the same hardware architecture.

6. Conclusions

A framework defines the architectural design of an object-oriented system. It describes the components of the system and the way they interact. In this paper, we propose a method for documenting a framework and use that method to describe a message passing system that has been built for an object-oriented operating system. Our work provides a starting point for understanding and documenting frameworks. We have found it useful for documenting the message passing system. We intend to use it to further document other parts of *Choices*.

We have found the extended typing of synchronization, control flow and entity relationship diagrams particularly useful. Typing has allowed us to *reuse* design in a consistent manner for different implementations of *Choices*. As we move *Choices* to platforms requiring different message passing implementations we hope that we will be able to reuse large parts of our original design. To complete our work we will also develop extended typing rules for concurrency and the states of the various objects.

There are, however, certain weaknesses in our model. In particular, the use of configuration constraints is not as formal as we would like. One way to remove the need for attributes on classes is to use multiple inheritance but this creates other problems. The use of multiple inheritance often produces extremely complicated class hierarchies. We have not found a good way to describe out-of-bounds flow of control. Irregular control flow tends to complicate the diagrams considerably.

We would also like to develop appropriate program annotations so that a parser can perform consistency checks for proper subtyping. Our current approach relies too heavily on programmer conventions. It would be useful to have a visual tool that shows the abstract entity relationships and control flows. Such a tool should also show the legal combinations and refinements.

References

Forest Baskett, J. H. Howard, and John T. Montague. Task Communication in DEMOS. *ACM Operating Systems Review,* pages 23–31, November 1977.

R. H. Campbell. The Specification of process synchronization by Path-Expressions. In *Lecture Notes in Computer Science,* pages 89–102, 1974.

Roy Campbell and Nayeem Islam. "A Parallel Object-Oriented Operating System (to appear)." In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

Roy H. Campbell, Nayeem Islam, and Peter Madany. *Choices,* Frameworks and Refinement. *Computing Systems,* 5(3), 1992.

David Cheriton. The V Distributed System. *Communications of the ACM*, pages 314–334, 1988.

Paul Close. The iPSC/2 Node Architecture. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications,* January 1986.

Peter Deutsch. *Levels of Reuse in the Smalltalk-80 Programming System.* IEEE Computer Society Press, Cambridge, Mass, 1987.

Peter Deutsch. *Design Reuse and Frameworks in the Smalltalk-80 Programming System.* ACM Press, Cambridge, Mass, 1989.

A. Goscinki. *Distributed Operating Systems: The Logical Design.* Addison-Wesley, Sydney, Australia, 1991.

Richard Helm, Ian Holland, and Dipayan Gangopadhay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *ECOOP/OOPSLA '90,* pages 169–180. ACM, 1990.

Nayeem Islam and Roy H. Campbell. "Reusable Data flow diagrams". Technical Report UIUCDCS-R-92-1770, University of Illinois Urbana-Champaign, Urbana, Illinois, November 1992.

Nayeem Islam and Roy H. Campbell. "Design Considerations for Shared Memory Multiprocessor Message Systems". In *IEEE Transactions on Parallel and Distributed Systems,* pages 702–711, November 1992.

Ralph E. Johnson and Vincent F. Russo. Reusing Object-Oriented Design. Technical Report UIUCDCS-R-91-1696, University of Illinois, May 1991.

Glen E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View-Controller Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming,* pages 26–49, 1988.

Paul Pierce. The NX/2 Operating System. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications,* January 1986.

Roger Pressman. *Software Engineering*. McGraw Hill, New York, New York, 1987.

Richard Rashid. Threads of a New System. *UNIX Review,* 1986.

David Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm C. Brown Publishers, Dubuque, Iowa, 1988.

John M. Vlissides and Mark Linton. Unidraw: A framework for building domain-specific graphical editors. In *User Interface Software Technologies,* pages 81–94. ACM SIGGRAPH/SIGCHI, 1989.

Peter Wegner. The Oject-Oriented Classification Paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[Received Mar. 5, 1993; Revised Apr. 14, 1993; Accepted Apr. 28, 1993]