Quaject Composition in the Synthesis Kernel

Calton Pu* and Henry Massalin
Department of Computer Science
Columbia University
New York, NY 10027
caltoncs.columbia.edu

Abstract

We describe the mechanisms in the Synthesis kernel to compose quajects, kernel objects that encapsulate state and operations. An example using ByteQueue quaject is used to illustrate the composition mechanism. This composition produces quajects with very low overhead, contributing significantly to high performance in the Synthesis kernel.

1 Introduction

Synthesis is an operating system kernel being developed for high performance distributed and parallel processing [3, 5]. Two unusual implementation techniques used systematically in the Synthesis kernel are run-time code generation of kernel calls [7] (as in the I/O system) and software feedback for resource management [4] (as in fine grain CPU scheduling). If naively implemented, these techniques might add considerable complexity to the Synthesis kernel. Part of the research in the implementation of Synthesis is in the integration of these techniques with well-known software engineering principles such as modularity.

All the existing versions of the Synthesis kernel have been implemented using some kind of Motorola 680X0 macro-assembler as a rapid prototyping programming language. The main reason is the unavailability of a high-level language compiler with a low-overhead code generator and supporting dynamic code generation. Despite this handicap, we still organize the Synthesis kernel using quajects, objects that give the Synthesis kernel a manageable structure. In this position paper, we outline how quajects are built and used in the current version of Synthesis kernel.

Quajects are similar to abstract data types and some object-oriented (O-O) systems such as Hydra [9] due to their rigid interface and consequent encapsulation. Quaject interfaces are defined in such a way as to support abstract and concrete types, as in Emerald [1]. However, quajects are different from O-O languages and some other O-O systems such as Choices [8] in several ways. Most salient of the differences is the lack of high-level language support and inheritance. Secondly, quaject interfaces may be defined procedurally. Although message-based and procedure-based systems are considered equivalent in some sense [2], traditional O-O systems have insisted on a message-based interface definition and implementation. Third, quajects are implemented in an unorthodox way, using run-time code generation and linking.

All in all, we think that the Synthesis kernel implementation differs from traditional O-O systems in form, but is similar in spirit. This position paper outlines the similarities through an example.
2 Synthesis Concepts

Some of the Synthesis kernel calls are synthesized at run-time for particular situations. This dynamic code generation takes advantage of the information available at run-time to reduce the interpretation overhead of “normal” kernel calls. For example, at open time the kernel can generate a particular code sequence for later read, given that these subsequent reads refer only to this file and this thread of execution.

There are three basic methods to synthesize code. The first is called factoring invariants, which bypasses redundant computations much like constant folding. The main source of overhead bypassed is the data structure traversals that are repeated at every kernel call to recover a specific kernel state. For example, a generic kernel read routine must traverse several (potentially large) kernel data structures to find out which device driver will be used for this file. This traversal may be avoided by a specialized read routine for the file that jumps directly into the driver code.

The second method is called collapsing layers, which eliminates unnecessary procedure calls and context switches. Several kinds of overhead are eliminated such as procedure parameter copying and protection checks across different contexts. For example, a naive network message driver may have a procedure to interpret the headers and trailers of each level in a multi-level protocol. The procedure calls may be avoided by integrating different layers into flat code, as apparently is done in the x-kernel [6]. The third method is called executable data structures. We omit it here because it has been described in some detail [3] and is not crucial to our example.

To illustrate the quaject creation, we use the example of a data channel. In Synthesis, all the data movement is represented as a data channel, which includes the I/O activities as a particular case. A data channel is usually a pipeline of cooperating threads that communicate with each other (pass data) using a pipe. More generally, a data channel can be an arbitrary graph, where the nodes are threads and the arcs the pipes (or other data communications or data sharing facility).

For concreteness, we show an example in Figures 1 and 2 from the TTY driver. The example is one of the stages in the data channel that writes to the terminal display. Although the example is very simple and thus does not describe the kernel organization, it does illustrate the actual composition of quajects. The interface and composition of larger quajects are similar.

3 Quaject Interface

Each quaject has a well-defined interface. Unlike most O-O systems, quajects do not support notions such as inheritance. Nevertheless, quajects are abstract data types in the sense that except for their interfaces no internal states of quajects are revealed to the outside world, thus quaject programmers are free to change quaject implementation as long as the interface is maintained.

Another traditional assumption of O-O systems is the definition of interfaces through messages. When a message is received, the receiving object interprets the content and invokes the appropriate method to handle the message. Some optimization techniques such as pointer swizzling have been proposed to reduce the amount of interpretation. In quajects, we define their interfaces in a way that minimizes interpretation cost. Our approach is based on the observation that messages are not the only way to define interfaces. In particular, an OS kernel is wholly contained in a single shared memory, therefore a procedural interface is a valid and desirable alternative. In addition, many message-based interfaces are defined as remote procedure calls, thus are procedural in form.

The quaject interface is defined in terms of callentry references and callback references. Each callentry reference is equivalent to a method in O-O systems. However, instead of a message-based interface description and implementation,
we adopt a procedural interface description. The implementation uses either direct jumps and branches for asynchronous communications or procedure calls for the synchronous case. Registers contain the input values and parameters, or pointers to large amounts of data.

A simple quaject example is the FIFO queue. As described in another paper [5], the Synthesis kernel supports four concrete types of queues, for the different combinations of single or multiple producers and consumers. The four concrete types all support the same abstract type [1], defined by two callentry references: Q-put and Q-get, which puts in and gets out an element of the queue. Both these callentry references return synchronously under the normal condition (successful insertion or deletion). Under other conditions, the queue invokes the callback references.

Quaject callback references are used for handling exceptions, status codes different from the normal return, and sometimes just connecting back to the caller. A queue, for example, has four callback references. Q-empty is invoked when a Q-get fails because the queue is empty. After Q-empty, the first element insertion calls Q-empty+1. Similarly, Q-full is invoked when a Q-put fails because the queue is full. And after Q-full, the first element removal calls Q-full-1. So instead of returning a condition code for interpretation by the invoker, the queue quaject calls directly the appropriate handling routines supplied by the invoker.

In addition to callentry and callback, we also consider as part of the quaject interface the external calls to other quajects. These external calls are named callout references. Functionally they are equivalent to procedure calls or method invocations in traditional O-O programs. The callout references are resolved at quaject composition time. Typically, the kernel links them to appropriate callentry references in the quajects being invoked.

### 4 Quaject Composition

At quaject instantiation time, the kernel resolves the quaject callentry and callback references. A kernel symbol table translates string names into the actual addresses and displacements. For example, Q-get is a callentry, represented in the symbol table as a displacement from the start of the queue quaject. The kernel links a queue to a data channel by inserting the queue quaject’s base address and the displacement for all the references. Some references may be filled by kernel default exception handling routines, so the programmer needs to specify only the references that have explicit connections.

We will use part of the TTY driver to illust-
trate the usage of queues and reference resolution. The quaject used in the driver is of the type ByteQueue. The driver supports two kinds of interfaces: blocking and non-blocking.

In Figure 1 we show a producer thread using the Q.put callentry reference to store bytes in the queue. (We omit the read operation since it is symmetric.) When the queue becomes full, the queue invokes the Q.full callback reference, which suspends the producer thread. When the ByteQueue’s reader removes a byte, the Q.full-1 callback reference is invoked, awakening the producer thread. This implements the familiar synchronous interface to a TTY driver.

Contrast this with Figure 2, which shows a non-blocking interface implemented using the same queue quaject. Only the connections between the ByteQueue and the thread change. The callentry reference is the same, but the callback references do not suspend or resume the producer thread. Instead, the Q.full condition, which previously blocked the thread waiting for output to drain, now returns the control to the producer thread without having written the bytes that did not fit. After output drains, Q.full-1 is called, invoking an exception handler in the producer thread which checks whether there are remaining bytes to write, and if so, it goes back to Q.put to finish the job.

This example illustrates the main points of a quaject interface. First, callentry references implement the equivalent of methods in O-O systems, bypassing interpretation in the server. Second, callback references implement the equivalent of return codes and exception handling, bypassing interpretation in the client. Third, kernel code synthesis produces tight straight line code that do not necessarily resemble the layered interface supported by the system. Finally, all the control flow between the client quaject and the server quaject is made explicit and documented.

The quaject composition need not be complete, i.e., not all references must be resolved at the quaject composition time. Quajects may provide default exception handlers for its callout and callback references, so an “unresolved” reference will produce a sensible result. These actions may be as simple as printing an error message and aborting the operation, or as complicated as dynamically creating the missing quaject, linking the reference, and continuing.

5 Protection

Up to now we have assumed that quajects being composed all reside in the same address space and execute at the same privilege level, so that direct procedure call is possible. Some quajects may run in either user mode or supervisor mode, so different instances may be composed with user programs or kernel routines. On the other hand, when user-level programs invoke kernel quajects, e.g., to read a file, the invocation crosses a protection boundary. One of the advantages of message-passing implementations is their ease of crossing protection boundaries. In our case, a direct procedure call from user program would not work because the kernel routine called needs to run in the supervisor mode.

User-level programs call kernel quajects by making system calls using the trap instruction, which jumps to a selection routine and switches from user mode to supervisor mode. Two of the parameters passed to trap tell it what quaject to operate on and which method to invoke. These parameters are small integers, which are indices used to find addresses in the current thread’s kernel quaject table (KQT). The KQT is conceptually a two-dimensional table of callout references, organized by quajects along the rows and each quaject’s methods along the columns. Because the user program can only specify indices into the KQT, the kernel quajects are guaranteed to execute only from legitimate entry points.

For user programs written in C, the Synthesis runtime library for the C language will translate the symbolic method names to the correct
method index number. For example, Synthesis reserves four trap numbers for general use: trap 1 for putchar, trap 2 for getchar, trap 3 for write, trap 4 for read. Other than these, each kernel quaject may assign the method index number independently of other quajects.

### 6 Conclusion

In the implementation of Synthesis kernel, quajects provide encapsulation and makes all inter-module dependencies explicit. This helps the kernel changes and debugging in an experimental system. It also increases parallelism in the kernel, while decreasing the need for synchronization, important properties in a multiprocessor kernel.

Using the data channel example, we have described the way quajects are composed together to provide important services in the Synthesis kernel. Although quajects differ from traditional O-O systems because of a procedural interface and run-time code generation implementation, the benefits of encapsulation and abstraction are preserved in a highly efficient implementation.

Our positive experience in using quajects shows that a highly efficient implementation of an object-based system can be achieved. The main ingredients of such an implementation are:

- a procedural interface using callentry references,
- explicit callback references for asynchronous return,
- run-time code generation and linking.

### References


