

Inheritance in Actor Based Concurrent Object-Oriented Languages

DENNIS G. KAFURA KEUNG HAE LEE

Department of Computer Science, Virginia Tech
Blacksburg, Virginia 24061, U.S.A.
kafura@vtopus.cs.vt.edu

Abstract

Inheritance is a valuable mechanism which enhances reusability and maintainability of software. A language design based on the actor model of concurrent computation faces a serious problem arising from the interference between concurrency and inheritance. A similar problem also occurs in other concurrent object-oriented languages. In this paper, we describe problems found in existing concurrent object-oriented languages. We present a solution which is based on a concept called behavior abstraction.

1. Introduction

Is inheritance inconsistent with concurrency? The interference between inheritance and object-based concurrency has been noted by others [Briot 87, Wegner 87]. These observations center on the difficulty of locating or copying methods at run-time in systems without shared memory. However, we do not consider this to be a fundamental difficulty because the performance penalty induced by inheritance may not always be a problem. At an extreme, the sharing problem may be avoided by allowing multiple copies of the same code and data on different nodes. [Briot 87] discusses a copy technique which is useful in this situation.

We believe that the fundamental interference between inheritance and concurrency is more deeply rooted. This difficulty can be observed in existing object-oriented languages, only a few of which support both concurrency and inheritance. The problem, as will be described later, is that inheritance and concurrency control tend to interfere with each other. This interference results in concurrent object-based languages which either do not support inheritance or which do so only by severely compromising some other property. For example, one language supporting both concurrency and inheritance compromised object encapsulation [Yokote 86]. A second language excludes the possibility of inheriting synchronization code [Caromel 88]. A result of this restriction is limited leverage in reusability. In yet another language, inheritance had been tried but was removed later because of limited reusability [America 87]. The same basic problem was found in the *initial approach of our own exploratory language design, called ACT++ [Kafura 88]. The language is a concurrent extension of C++ [Stroustrup 86] based on the actor computation model of Agha and Hewitt [Agha 87].*

This paper analyzes the approaches to inheritance and concurrency control in existing object-oriented languages and proposes a solution to the interference problem using what we call *behavior abstraction*. In the remainder of this paper, sections 2 and 3 provide background for the research while the major research contribution of this paper is found in sections 4 and 5. In section 2, we present our view of inheritance and delegation. Reuse and sharing are distinguished in order to motivate our use of inheritance rather than delegation. Section 3 classifies approaches to concurrency control in existing object-oriented languages. Based on this classification, an analysis of currently existing concurrent object-oriented languages is provided. The conflict between inheritance and concurrency found in an actor based language is described in section 4. Section 5 discusses our solution to this problem.

2. Inheritance and Delegation

Both inheritance and delegation are mechanisms for sharing knowledge in object-oriented programming. Recently, there has been a great debate on the power of these two mechanisms [Lieberman 86, 88, Stein 87]. In this section, we discuss a viewpoint on the difference between inheritance and delegation. This discussion motivates our attempt to combine the actor model with inheritance rather than delegation. In order to put our view on this issue in perspective, we distinguish two concepts, reuse and sharing. Our position is that inheritance provides more power in reusability while delegation provides more power and flexibility in sharing.

2.1 Reuse and Sharing

Reuse is the activity of using an existing component in defining a new component. Reuse has been recognized as an important activity in software engineering [Freeman 81, Biggerstaff 87]. It is possible to reuse a component without sharing it.

Sharing denotes that the same component is used by more than a single client at run time. For example, code sharing occurs if a code segment in memory is used by different processes. Sharing offers several advantages. One important advantage of sharing is the flexibility in modifying properties of objects at run time. For example, in Smalltalk-80, a class object is shared by all instances of the class at run time. A modification of a method in the class is automatically reflected in the behavior of every instance of the class. If each instance has its own copy of the method, this localized modification is not possible at run time. These advantages are the main motivations behind the development of prototype-based systems such as Act1 [Lieberman 87] and Self [Ungar 87], where sharing among objects occurs dynamically.

2.2 Inheritance vs Delegation

Inheritance is closely tied with the notion of a class. A class captures static properties of objects such as attributes and methods in an explicit form. A new class can be defined as an extension of existing classes with the support of inheritance. Class hierarchies provide a natural classification of components and enhances modularized modification. These expectations have been evidenced in [Goldberg 83, Campbell 87, Johnson 88].

While class-based languages allow limited sharing of values between objects, delegation-based languages provide strong support for dynamic and efficient sharing of object properties [Lieberman 86]. However, in a delegation-based system, sharing seems

to be limited to run-time entities such as values and objects. No explicit organized collection of reusable components exist.

Classes and inheritance are valuable where reusability and maintainability are emphasized more than flexibility. Prototype and delegation based languages are more suitable when the power of dynamic and flexible sharing can be maximally exploited. The support for classes and inheritance in languages seems more natural when strong type-checking and efficient code are favored over flexibility and dynamicity.

3. Concurrency Control in Object-Based Languages

This section discusses the relationship between concurrency and inheritance in existing languages. An observation is made about the interference between the two mechanisms.

An object in a concurrent object-oriented language may proceed in parallel with another object. Such an object has its own thread of control. We call an object with its own thread of control an *active* object. In contrast, an object in a sequential language does not possess its own thread of control. We will refer to an object without its own thread of control as a *passive* object.

Concurrency implies the need for synchronization, without which the state of an active object may become inconsistent. Since the internal state of an object can only be accessed via method invocation, previous object-based concurrency control techniques were implemented inside the object. There are two directions in providing concurrency control. One approach centralizes concurrency control in a single procedure. We call this approach *centralized control*. The other approach distributes concurrency control among methods without a centralized procedure. We call this *decentralized control*.

In centralized control, message reception is explicitly programmed using guarded commands or SELECT constructs. CSP [Hoare 78], ADA, ABCL/1 [Yonezawa 87], POOL-T [America 87], and Extended Eiffel¹ [Caromel 88] belong to this category. There is a common problem in attempting to incorporate inheritance into these languages: synchronization constraints specified in the centralized procedure cannot be inherited by a subclass. This point will become clear when we review these languages later in this section.

Two different approaches to concurrency control are found in languages with decentralized control. One approach uses *critical sections* and the other approach uses what we call *interface control*. A majority of languages adopt the critical section approach. In these languages, each method is responsible for ensuring a certain condition before entering a critical section. Several languages use a locking mechanism. Each method must explicitly lock a variable before entering a critical section and must unlock the variable when exiting the critical section. Other languages use a construct similar to a conditional critical region [Hansen 72]. For example, a newer version of Concurrent Smalltalk provides a construct called *relinquish* which allows a thread to wait on a condition [Yokote 87]. In Trellis/Owl [Moss 87], the *lock block* structure automatically performs an unlock for a lock variable when its scope is exited.

¹ The language is a concurrent extension of Eiffel [Meyer 87]. Since the language was not given a name in [Caromel 88], we will refer to the language as *Extended Eiffel*.

Two problems exist in the approach based on critical sections. First, object encapsulation is weakened. Relying on a lock variable requires the variable to be visible to any subclass in the class hierarchy, which is a violation of encapsulation. A similar observation was made by [Snyder 87]. Second, it is possible for a method to violate the critical section protocol. For example, if explicit locking is used, a method may enter the critical section without performing the locking. This problem is compounded in a language which supports inheritance. Because the subclass is separated from the superclass, there is a greater possibility that methods defined in a subclass may not observe the critical section protocol.

The other approach in decentralized control is based on direct control of the object interface. In this approach, called *interface control*, message reception is implicit. A method execution is initiated only when the method is allowed to access the internal state of the object. The underlying mail system delivers a message when the receiver is ready. Hybrid [Nierstrasz 87] and actor based languages such as Act2 [Theriault 83] and Act3 [Agha 86] are found in this category. For example, Hybrid provides constructs which control the availability of methods. A method may be closed temporarily so that messages for that method are not allowed to cross the object boundary. The blocked messages are processed later when the method is opened. In Act2 and Act3, synchronization of an active object is achieved with the operation called *become*. This operation allows an object to change to another object, which may have a different interface and even different data structures. Neither Act2 nor Act3 supports inheritance.

A serious problem occurs when adding inheritance to languages using interface control. Defining a new method in a subclass may invalidate many superclass methods.

The remainder of this section describes how the problem of combining inheritance and concurrency is manifested in the following concurrent object-based languages: POOL-T, Extended Eiffel, Concurrent Smalltalk, Hybrid, ACT3, and ACT++. Each of these languages uses a different approach to concurrency control. While some of these languages do not support inheritance, they are included here since a review of these languages provides insight into the conflict between inheritance and concurrency.

POOL-T: the concurrency control approach of POOL-T [America 87] is centralized. In POOL-T, the class definition of a concurrent object consists of a list of methods and a separate procedure called *body* which specifies concurrency constraints. An object explicitly states its willingness to accept messages in the *body* using a construct similar to guarded command. POOL-T does not support inheritance. In fact, inheritance was tried in the initial design, but was removed later [America 87]. The decision to remove inheritance from POOL-T illustrates the general interference between inheritance and centralized control. The problem is that inheritance in a language with centralized control does not allow synchronization code to be reused. In centralized control languages like POOL-T, each time a subclass with a new method is defined, the *body* must be revised since otherwise no new methods defined in the subclass can be executed. It is this consideration that led the designer of POOL-T to choose not to include inheritance in the language.

Extended Eiffel: a concurrent extension of Eiffel proposed in [Caromel 88] supports both concurrency and inheritance using centralized control. An active object is defined as an instance of a subclass of a class called "PROCESS-POWER". A method in a class is not

concerned with synchronization. Concurrency control is centralized into a single procedure called *Live* which is similar to *body* of POOL-T. Extended Eiffel suffers from the problem manifested in an earlier design of POOL-T. The method *Live* must be rewritten if a subclass adds a new method, regardless of the semantics of the method being added.

The approach of Extended Eiffel excludes the inheritance of synchronization code, and thereby severely restricts reusability. The synchronization code of *Live* may be a result of an extensive reasoning process. A subtle error may creep in during the process of copying and modifying the *Live* method. This is the very problem that inheritance intends to solve. While the separation of concurrency control from sequential action may allow a more readable definition of an object's behavior, we believe that readability can also be provided by a language which uses decentralized control. We discuss this in more detail when we present our solution in section 5.

Concurrent Smalltalk: Concurrent Smalltalk is a concurrent extension of Smalltalk-80 [Yokote 86] which supports both concurrency and inheritance. The language uses critical sections for concurrency control. An active object, called an *atomic object*, serializes messages to maintain consistency of its internal state. Locking is used for concurrency control. An active object allows a method to be executed even when the method execution is immediately blocked. In this case, the client object should block itself, terminating its current process. A provision is required in the code of the client which will send the same message again to the object when the client is restarted. Since the client is terminated and restarted, it must have a separate method which will do the retransmission.

This approach has several disadvantages. One is a weak object encapsulation. In the language, a sender must provide the method which will retransmit a message. This method obscures the readability of the program and imposes a burden on the sender. The sender is also required to understand the internals of the receiver object. This violates the encapsulation principle of object-oriented languages. Another disadvantage is the use of unstructured constructs. For example, the BoundedBuffer problem described in [Yokote 86] uses a wait-signal primitive. The drawback of such a low-level primitive has been well recognized in operating systems research. A later version of Concurrent Smalltalk [Yokote 87] improves this situation by using a *relinquish* operation and the concept of a *secretary*, which is similar to conditional critical regions [Hansen 72]. This approach still has the disadvantages intrinsic to an approach based on critical sections.

Hybrid: Hybrid [Nierstrasz 87] is a concurrent object-oriented language based on decentralized control. The language provides a message queue called *delay queue* for concurrency control within an active object. Each method of an active object is associated with a delay queue. Synchronization control for accessing an object is achieved by explicitly closing and opening delay queues. Each method contains explicit statements for controlling delay queues. A message which requests the execution of a method is blocked if the delay queue associated with the method is closed. The message is processed later when the delay queue is opened by some method.

Hybrid supports multiple inheritance. The concurrency control approach used in Hybrid presents a problem when we attempt to define a subclass of an active object class. To appreciate this problem, consider adding a new method in defining a subclass. The new method may need to have its own delay queue which was not present in its superclass. The

question then is how the methods of its superclass can control this delay queue. Unless the new delay queue is controlled solely by the new method itself, all superclass methods that need to open or close the delay queue must be revised so that the name of the delay queue may be referenced in their definitions.

Act3: Act3 is a concurrent object-oriented language based on the actor model as defined by Agha [Agha 86]. The language represents another approach in interface control. A main synchronization device of ACT3 is the *become* operation. It is also the only synchronization primitive other than message passing operations. A *become* operation in a method specifies a replacement behavior, which receives the next unprocessed message. Each method execution must use a *become* operation to name a replacement behavior. Specifying a replacement behavior is the way an actor changes its state. In the actor model, both state change and synchronization control is accomplished using a single *become* operation. ACT3 does not provide inheritance. A language which intends to support inheritance and the actor model of concurrency faces a fundamental problem, which is similar to that of Hybrid but more serious. The problem was noticed in the initial design of our own language ACT++.

4. The Actor-Inheritance Conflict

In this section, we use ACT++ to illustrate the conflict between inheritance and concurrency in a language based on the actor model. Although we are using ACT++, the interference problem is not specific to ACT++ and also occurs in other languages combining concurrency and inheritance. Before presenting the description of the problem, we provide a description of the relevant parts of ACT++. Other aspects of ACT++ are described in [Kafura 88].

ACT++ is a language design which supports both class inheritance and the actor model of concurrency. As an expedient implementation strategy, we used C++ as the base language, extending it with the concurrency abstraction of the actor model. In ACT++, actors represent active objects. All non-actor objects are passive objects. A passive object represents a C++ object, which is local to a single active object. A shared object must be an actor. An actor class, a class whose instances are actors, is defined as a direct or indirect subclass of the special class ACTOR. Like passive objects, an actor class can inherit properties from an existing actor class by defining itself to be a subclass of the existing actor class. ACT++ distributes concurrency control into each method.

We now describe by example the interference of inheritance and actor concurrency. Consider producers and consumers communicating through a bounded buffer. The bounded buffer is modeled as an active object which is shared by producers and consumers. The buffer provides *get()* and *put()* methods to clients. Producers are actors which send *put()* requests when they want to deliver data items to consumers. A consumer actor sends a *get()* message to the buffer when the consumer needs a data item. A bounded buffer actor is empty when it is initially created. An empty buffer accepts only a *put()* message. If the buffer is neither empty nor full, it acts as a partially filled buffer which honors both *get()* and *put()* requests. If the buffer is full then it must accept only a *get()* request from a consumer. We will call these three states *empty_buffer*, *partial_buffer*, and *full_buffer*, respectively. A possible transition sequence in the states of a bounded buffer is

empty_buffer -> partial_buffer -> full_buffer -> partial_buffer -> empty_buffer.

A subtle semantic question now arises. What will happen if the current state of an actor does not recognize the method name in a message? For example, what should be done if the next message to be processed contains a `get()` request while the buffer is empty? The answer to this question in the context of ACT++ is being investigated. For the purpose of this paper, we assume that a message will be put back at the end of the message queue. Figure 1 shows the definition of `bounded_buffer` in ACT++¹.

```

class bounded_buffer : ACTOR {
    int_array buf[MAX];
    int in,out;
public:
    bounded_buffer()
        { in=0; out=0}
    int get()
        {
            reply buf[out++];
            out %= MAX;
            if (in==out)
                become(empty_buffer(buf,in,out));
            else
                become(partial_buffer(buf,in,out));
        }
    void put(int item)
        {
            buf[in++] = item;
            in %= MAX;
            if (in==out%MAX)
                become(full_buffer(buf,in,out));
            else
                become(partial_buffer(buf,in,out));
        }
};

```

Figure 1 - Definition of `bounded_buffer`

The syntax of ACT++ is close to that of C++. A few new constructs were added to support the actor abstraction. In Figure 1, the operation `become` is used to specify a replacement behavior. A `become` operation takes an actor class as an argument. An actor class corresponds to a behavior script of the primitive actor model [Agha 86]. The operation `reply` is used to send a message to the sender of the message being processed. Since the class `bounded_buffer` is defined as a subclass of `ACTOR`, the `bounded_buffer` is an actor class whose instances are active objects, namely actors. An instance of `bounded_buffer` contains instance variables `in`, `out`, and the array `buf`. In C++, a method

¹While the primitive actor model assumes no structured types, such as array, ACT++ provides all data types of C++. For the purpose of this paper, we assume an array parameter is passed by value.

with the same name as the class name denotes a constructor. The procedure `bounded_buffer()` is a constructor.

To recognize the operations which are appropriate for different behaviors (eg. empty, full, partial) we introduce three classes of bounded buffer: namely, `empty_buffer`, `full_buffer`, and `partial_buffer`. These three classes are defined as subclasses of the class `bounded_buffer`. The subclass `empty_buffer` is the same as `bounded_buffer` except that it does not have the `get()` method. The subclass `full_buffer` is a `bounded_buffer` without `put()`. The subclass `partial_buffer` is exactly the same as the `bounded_buffer` class. These subclasses can be defined as restrictions of the class `bounded_buffer`. The definitions of the three subclasses follow.

```
class empty_buffer : bounded_buffer {
public:
    bounded_buffer::put;
};
class full_buffer : bounded_buffer {
public:
    bounded_buffer::get;
};
class partial_buffer : bounded_buffer {
public:
    bounded_buffer::get;
    bounded_buffer::put;
};
```

The first concern is that many similar classes must be defined to implement a `bounded_buffer`. This is a result of the natural mapping of the primitive actor model into a class-based object-oriented language. The use of the become operation implies a different class be defined for each different interface. This is unpleasant since all of the different behaviors have almost the same methods, yet they all must be defined as distinct classes. However, the real problem occurs when a subclass with its own method needs to be defined.

Suppose that we want to implement a bounded buffer with a new method `get_rear()`, which returns the most recently deposited item, rather than the oldest one. We call this an `extended_buffer`. A plausible solution is to define the `extended_buffer` class as a subclass of `bounded_buffer` with an addition of a new method `get_rear()`. The `extended_buffer` should be able to inherit all other methods from `bounded_buffer` without change. This is not an unusual expectation of a language with inheritance. Unfortunately, this solution does not work as described below.

The possible behaviors of an `extended_buffer` are:

<code>extended_empty_buffer</code>	<code>put()</code>
<code>extended_full_buffer</code>	<code>get(), get_rear()</code>
<code>extended_partial_buffer</code>	<code>get(), get_rear(), put()</code>

Comparing these behaviors with those of `bounded_buffer`, we find that `extended_empty_buffer` and `empty_buffer` have the same interface. Hence `empty_buffer` may be used in place of `extended_empty_buffer` in the new class definition. However, `extended_full_buffer` is different from `full_buffer` because of the new method `get_rear()` in the `extended_full_buffer`. Similarly, the behaviors `extended_partial_buffer` and `partial_buffer` are also different. Therefore, `extended_full_buffer` and `extended_partial_buffer` must be defined as new classes.

However, the problem does not end here. Notice that every method of `bounded_buffer` must be redefined in the definition of `extended_buffer` if the method refers to either of the two class names, `full_buffer` and `partial_buffer`. Since both `get()` and `put()` uses `partial_buffer`, none of these method can be inherited. Hence, `extended_buffer` inherits no methods from its superclass. All of its methods must be implemented within its own definition! This argument equally applies to an attempt to define `extended_full_buffer` as a subclass of `full_buffer` and `extended_partial_buffer` as a subclass of `partial_buffer`. The point of this example is that no methods of the superclass can be reused in the definition of a subclass.

We have already observed that a similar interference problem exists in Hybrid. In both ACT++ and Hybrid, superclass methods are not independent of new methods being defined in a subclass. The degree of dependency is, however, higher in a language based on the actor model of computation.

5. Inheritance in Actors

Having described the conflict of concurrency and inheritance, we now present our solution to this problem. Our solution is presented in the framework of an actor based language.

5.1 A Model of an Object Manager

Each active object (actor) is associated with an *object manager*. The object manager is responsible for protecting the object from unauthorized requests and for dispatching method invocations. An object manager is automatically created when an object is created. The object manager immediately starts and continues until the object is destroyed.

The object manager of an object protects the object by enforcing the interface of the object. Using the terminology of Hybrid, the *interface* of an object consists of all open methods. A method is *open* if the current interface of object can accept a message for the method. Otherwise, a method is *closed*. The interface of an object is dynamically changed since methods can be opened or closed during computation. Methods are closed by the object manager and opened by a method in execution, called a thread (see below). A message for a method invocation is *authorized* if the method is open. A message for a closed method is *unauthorized*.

The object manager waits for the arrival of an authorized message. On finding such a message, the object manager closes all methods and creates a thread which will perform the requested method. Unauthorized messages are buffered by the object manager until their corresponding methods are opened. Closed methods may be opened by a *become* operation executed by a thread. A *become* operation specifies a set of methods to be opened. A thread can perform the *become* operation only once in its life. Since a thread is created as the result of the previous *become* operation, no thread but the most recently

dispatched thread can execute the become operation. The become operation will open at least one method; otherwise, the actor is garbage collected.

There may exist multiple threads inside an object since the become operation may be executed prior to the termination of a thread. All the threads proceed in parallel. A thread dies when the execution of the method represented by the thread is completed. Among threads, no variables are shared.

5.2 Implementation of the object manager

The function of an object manager is well-defined and uniform for every object. Hence, a programmer does not need to write the object manager. The object manager can be provided through compiler and run-time support. This obviates the need for concurrency control mechanisms to be centralized in one method. The object manager can be implemented either as a function of the mail queue or as a special thread. The former will result in a sophisticated mail queue while the latter is similar to a process scheduler.

The object manager will need to keep track of the object's interface, which is changed by a become operation of that object's most recently dispatched thread. This problem along with the interference problem can be solved by redefining the way a replacement behavior is specified. For this purpose, we introduce the concept of *behavior abstraction*.

5.3 Behavior abstraction

A *behavior name* is a handle for a set of open method names. For example, consider the `bounded_buffer`. The buffer actor has one of the following behaviors:

```
empty_buffer = { put() }
full_buffer = { get() }
partial_buffer = { get(), put() }
```

With these, we have defined three behavior names; namely, `empty_buffer`, `full_buffer`, and `partial_buffer`.

A become operation specifies a replacement in terms of a behavior name. For example, "become `full_buffer`" is acceptable if `full_buffer` is a behavior name. The language should provide a convenient way for specifying behavior names. For example, defining a behavior name using a regular expression may be desirable. Note the difference in the usage of "behavior" in the primitive actor model and in our model. In our model, a behavior denotes a set of open methods while a behavior in the primitive actor model means a script of an actor.

We now present the solution to the problem of `extended_buffer` which inherits from the `bounded_buffer`. The `bounded_buffer` and the `extended_buffer` are defined using the behavior names defined earlier in this section. Figure 2 shows a new definition of the `bounded_buffer` using behavior abstraction. The definition of an `extended_buffer` which inherits from the `bounded_buffer` is shown in Figure 3. The `extended_buffer` has three distinct behaviors. Each of these constitutes a behavior name. We define the following names as the relevant behaviors for an `extended_buffer`:

```

extended_empty_buffer = {put()}
extended_full_buffer = {get(), get_rear()}
extended_partial_buffer = { get(), get_rear(), put()}

```

```

class bounded_buffer : Actor {
  int_array buf[MAX];
  int in,out;
  behavior:
  empty_buffer = {put()};
  full_buffer = {get()};
  partial_buffer = {get(),put()};
  public:
  buffer()
  {
    in=0;
    out=0;
    become empty_buffer;
  }
  void put(int item)
  {
    buf[in++]=item;
    in %= MAX;
    if (in==(out+1)%MAX)
      become full_buffer;
    else
      become partial_buffer;
  }
  int get()
  {
    reply buf[out++];
    out %= MAX;
    if (in==out)
      become empty_buffer;
    else
      become partial_buffer;
  }
};

```

Figure 2 - Bounded_buffer with behavior abstraction

We must now consider the relationship between the behavior names of the subclass and those of the superclass. Consider the `put()` method, which is inherited from the superclass `bounded_buffer`. The new operation `get_rear()` does not belong to any behavior names named by `put()`. It is necessary to let the method `put()` know that `get_rear()` is added in the definition of the `extended_buffer`. This is accomplished by redefining the behavior names used in superclass methods. The redefinition is expressed by the "redefines" construct. The new definition of a behavior name will be used by all superclass methods. In some cases, the redefinition of a behavior name does not change the set of methods. Such renaming may be desired to provide the object with a more appropriate name. For this reason, we

redefine `empty_buffer` as `extended_empty_buffer` without changing its meaning using the "renames" construct.

```

class extended_buffer : public bounded_buffer {
behavior:
    extended_empty_buffer                renames empty_buffer;
    extended_full_buffer    = {get(), get_rear()}    redefines full_buffer;
    extended_partial_buffer = {get(), get_rear(), put()} redefines partial_buffer;
public:
    extended_buffer()
    {
        in=out=0;
        become extended_empty_buffer;
    }
    int get_rear()
    {
        reply(buf[--in%max]);
        if (in==out)
            become extended_empty_buffer;
        else
            become extended_partial_buffer;
    }
};

```

Figure 3 - Definition of extended_buffer

Using behavior names also has several other advantages. First, behavior names improve program readability. With more expressive and meaningful names, a program is more readable because the next interface is denoted by the behavior name used in a become operation. Second, an active object requires that a programmer understand its dynamic run time behavior. While the centralized approach provides an effective way to tackle this issue by separating concurrency control from sequential actions, the approach has the drawback of excluding the inheritance of synchronization code. While our model allows inheritance, it also allows concurrency control to be separated. Third, the synchronization mechanism is structured because no matching primitive is needed for the become operation. This is an important requirement of synchronization primitives proposed for incremental programming. This avoids such problems as a new method defined in a subclass which forgets to signal superclass methods or fails to observe a critical section protocol. Fourth, it supports an object-oriented design methodology. The behavior names provide a level of abstraction whose granularity is smaller than data abstraction but larger than procedural abstraction. With the behavior abstraction, the behavior of an object can be modeled as state transitions among behavior names. Each of these names provides a higher level abstraction which is more relevant to a programmer's conceptual view of an object. The state-transition behavior of an object is naturally expressed with the behavior abstraction.

5.4 Implementation of behavior abstraction

The mechanism of redefining a behavior name in a subclass is analogous to that of virtual function in C++ [Stroustrup 86]. Every behavior name declared in a class may be regarded

as a "virtual behavior" whose meaning a subclass may override. As in a virtual function invocation, a behavior name used by an object in a become operation may denote different behaviors which are decided by the type of the object. One implementation scheme resembles the virtual function table of C++. The compiler creates a table of behavior names for each class. The become operation is translated into specifying a set of open methods which are found using a behavior name as an index into the table.

6. Limitations and Future Research

There are several limitations to our approach. The most fundamental limitation is the assumption of a closed system. One of the fundamental principles of the actor model is the *openness* of the model. Openness means that an actor can modify itself dynamically (i.e., at run-time) upon receipt of a message which requires a computation unanticipated by the original behavior. The reconfigurability of actor relationships is extended beyond that conceived of by the programmer. While openness provides a flexible computation model, it is a significant obstacle to be overcome in the design of a language like ACT++, which prefers safety to flexibility. In the presence of openness, type-checking of a message is impossible since an actor's behavior may mutate without restriction during execution. While our model assumes a closed system, we do not consider this a weakness since type safety is one of our design goals as other languages have chosen type safety over flexibility. A natural next step is to relate the concept of behavior abstraction to a type system. We are currently investigating a type system based on the behavior abstraction which will allow more flexible behavior replacements.

7. Conclusions

The interference between inheritance and concurrency has been identified by several researchers as difficulty in sharing methods in distributed environments. There is a more fundamental problem in combining the two mechanisms in a single language. In this paper, we described this problem, and presented an analysis of existing concurrent object-oriented languages from this perspective. Finally, we presented our solution in the framework of the actor model of concurrent computation. A solution to the problem of combining concurrency and inheritance, based on the concept of behavior abstraction, was discussed in detail using our exploratory language ACT++.

Acknowledgements

We are grateful to the anonymous referees and Oscar Nierstrasz for their comments. We also thank the members of Real-Time Systems Group at Virginia Tech. Discussions with Greg Lavender, Michael Leahy, Jeff Nelson, and Sanjay Kohli helped in clarifying the concepts of the actor model.

References

- [Agha 86] G. Agha, *A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.
- [Agha 87] G. Agha and C. Hewitt, *Concurrent Programming Using Actors*, In *Object-Oriented Concurrent Programming*, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 37-53.

- [America 87] P. America, POOL-T: A Parallel Object-Oriented Language, In *Object-Oriented Concurrent Programming*, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 199-220.
- [Biggerstaff 87] T. Biggerstaff and C. Richter, Reusability Framework, Assessment, and Directions, IEEE Software, March 1987, 41-49.
- [Hansen] P. B. Hansen, Structured Multiprogramming, CACM, Vol. 15, No. 7, July 1972.
- [Briot 87] J.-P. Briot and A. Yonezawa, Inheritance and Synchronization in Concurrent OOP, ECOOP '87 European Conference on Object-Oriented Programming, Springer-Verlag, June 1987, 33-40.
- [Campbell 87] R. H. Campbell, J. Johnston, V. F. Russo, Choices: Class Hierarchical Open Interface for Custom Embedded systems, Operating Systems Review 21, July 1987, 9-17.
- [Caromel 88] D. Caromel, A General Method for Concurrent and distributed Object-Oriented Programming, Extended Abstract, Workshop on Object-Oriented Concurrent Programming, OOPSLA '88, San Diego, California, September 1988.
- [Freeman 81] P. Freeman, Reusable Software Engineering: Concepts and Research Directions, Proceedings of Workshop on Reusability in Programming, ITT, Shelton, Conn., 1983.
- [Goldberg 83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
- [Hewitt 76] C. Hewitt, Viewing Control Structures as Patterns of Passing Messages, AI MEMO 410, MIT Artificial Intelligence Laboratory, 1976.
- [Hoare 78] C. A. R. Hoare, Communicating Sequential Processes, CACM, August, 1978.
- [Johnson 88] R. E. Johnson, J. O. Graver, and L. W. Zurawski, TS: An Optimizing Compiler for Smalltalk, OOPSLA '88 Conference Proceedings, 1988.
- [Kafura 88] D. G. Kafura, Concurrent Object-Oriented Real-Time Systems Research, Technical Report, TR 88-47, Dept. of Computer Science, Virginia Tech, 1988.
- [Lieberman 86] H. Lieberman, Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Languages, OOPSLA '86 Conference Proceedings, 1986.
- [Lieberman 87] H. Lieberman, Concurrent Object-Oriented Programming in Act 1, In *Object-Oriented Concurrent Programming*, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 9-36.
- [Lieberman 88] H. Lieberman, L. Stein, and D. Ungar, Treaty of Orlando, Addendum to the Proceedings of OOPSLA '87, Special Issue of SIGPLAN Notices 23, 5, May 1988.
- [Moss 87] J. B. Moss and W. H. Kohler, Concurrency Features for the Trellis/Owl Language, ECOOP '87 European Conference on Object-Oriented programming, Springer-Verlag, June 1987, 171-180.

- [Meyer 87] B. Meyer, Eiffel: Programming for Reusability and Extendibility, SIGPLAN Notices, Vol. 22, 2, January 1987.
- [Nierstrasz 87] O. M. Nierstrasz, Active Objects in Hybrid, OOPSLA '87 Conference Proceedings, 1987, 243-253.
- [Schaffert 86] C. Schaffert et al., An Introduction to Trellis/Owl, OOPSLA '86 Conference Proceedings, 1986.
- [Snyder 86] A. Snyder, Encapsulation and Inheritance in Object-Oriented Programming Languages, OOPSLA '86 Conference Proceedings, 1986.
- [Stein 87] L. Stein, Delegation is Inheritance, OOPSLA '87 conference Proceedings, 1987, 138-146.
- [Stroustrup 86] B. Stroustrup, The C++ Programming Language, Addison-Wesely, Menlo Park, Calif., 1986.
- [Theriault 83] D. G. Theriault, Issues in the Design and Implementation of Act2, Technical Report 728, MIT Artificial Intelligence Laboratory, 1983.
- [Ungar 87] D. Ungar and R.B. Smith, Self: The power of Simplicity, OOPSLA '87 Conference Proceedings, October 1987, 227-242.
- [Yokote 86] Y. Yokote and M. Tokoro, Concurrent Programming in Concurrent Smalltalk, In *Object-Oriented Concurrent Programming*, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987.
- [Yokote 87] Y. Yokote and M. Tokoro, Experience and Evolution of Concurrent Smalltalk, OOPSLA '87 Conference Proceedings, 1987, 406-415.
- [Yonezawa 87] A. Yonezawa, E. Shibayama, et al., Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, In *Object-Oriented Concurrent Programming*, (ed.) A. Yonezawa and M. Tokoro, MIT Press, 1987, 55-89.
- [Wegner 87] P. Wegner, Dimensions of Object-Based Language Design, OOPSLA '87 Conference Proceedings, 1987, 168-182.