Specifying and Inheriting Concurrent Behavior in an Actor-Based Object-Oriented Language

by Greg Lavender and Dennis Kafura

TR 90-56

Specifying and Inheriting Concurrent Behavior in an Actor-based Object-Oriented Language

Greg Lavender
lavender@vtopus.cs.vt.edu
(703) 953-3458

Dennis Kafura
kafura@vtopus.cs.vt.edu
(703) 231-5568

Department of Computer Science, Virginia Tech
Blacksburg, Virginia, U.S.A 24061

Fax: (703) 231-6075

Abstract

Using CCS *behavior equations* to specify and reason about the observable behavior of concurrent objects, we demonstrate that a language mechanism called a *behavior set* can be used to capture the behavior of actor-like objects. Using behavior equations as a formal representation of concurrent object behavior results in the explication of a mapping from the domain of object states to a domain of behavior sets. We call this mapping the *behavior function.* By expressing relevant object states, behavior sets, and the behavior function as first-class, inheritable, and mutable entities in a concurrent object-oriented language, we have defined the conditions which must be met in order to inherit concurrent behavior free of known anomalies.

Keywords: Actors, ACT++, Behavior Sets, CCS, Concurrency, Inheritance, Object Behavior

Specifying and Inheriting Concurrent Behavior in an Actor-based Object-Oriented Language

Greg Lavender
lavender@vtopus.cs.vt.edu

Dennis Kafura
kafura@vtopus.cs.vt.edu

Department of Computer Science, Virginia Tech
Blacksburg, Virginia, U.S.A 24061

Abstract

Using CCS *behavior equations* to specify and reason about the observable behavior of concurrent objects, we demonstrate that a language mechanism called a *behavior set* can be used to capture the behavior of actor-like objects. Using behavior equations as a formal representation of concurrent object behavior results in the explication of a mapping from the domain of object states to a domain of behavior sets. We call this mapping the *behavior function.* By expressing relevant object states, behavior sets, and the behavior function as first-class, inheritable, and mutable entities in a concurrent object-oriented language, we have defined the conditions which must be met in order to inherit concurrent behavior free of known anomalies.

1 Introduction

In a concurrent object-oriented language, we would like to be able to inherit behavior and realize synchronization control without compromising the flexibility of either the inheritance mechanism or the synchronization mechanism. A problem called the *inheritance anomaly* [Matsuoka 1990] arises when we implement synchronization control within a class and then attempt to specialize behavior through inheritance.

In this paper, we formalize the notion of concurrent object behavior using Milner's Calculus of Communicating Systems (CCS) [Milner 1989]. The formalization facilitates the identification of the fundamental cause of the inheritance anomaly and leads to the definition of a set of conditions which are necessary for inheritance and synchronization control to coexist in concurrent object-oriented languages.

For the those not familiar with the inheritance anomaly, we briefly review the problem in section 2. Those familiar with the problem may wish to continue with section 3.

In section 3, we use the notation of CCS to specify and reason about the behavior of a concurrent object. We also define the elements of a model which are necessary for identifying the fundamental cause of the inheritance anomaly

Having developed a means for reasoning about object behavior and having identified the cause of the inheritance anomaly, we discuss in section 4 what it means to inherit concurrent behavior. We use our prototype object-oriented actor language ACT++ as an example.

We conclude with a summary of the ideas presented in the paper and we discuss the current status of our work and the ACT++ prototype.

2 The Inheritance Anomaly

In the object-oriented paradigm an object is viewed as an encapsulation of state and code in the form of instance variables and methods, respectively. In object-oriented languages, the declaration of the types and names of instance variables and the signatures of the methods are most often declared in a class definition. The class definition serves as a contract between objects of that class and clients or subclasses that intend to use instances of the class. A subclass is a specialization of a particular class. The specialization is achieved through an inheritance mechanism. These are the features of the object-oriented paradigm that we are interested in here. A more thorough discussion can be found in [Wegner 1990a].

In standard object-oriented models, all the methods declared in a class definition are always available for execution by a client regardless of the internal state of an object of that class. The class implementor typically provides, within the implementation of each method, the code necessary to determine whether or not the object is in a state in which execution of the requested method is appropriate.

For example, a stack object with a pop method must first verify that the stack is non-empty before proceeding. Typically, the pop method will return an error value indicating that an underflow condition has occured. The usual mechanism for communicating the underflow condition to the client requires an overloading of the return type of the method.

That is, the return value is outside the domain of values returned by a legitimate pop operation. The client of the stack object must be aware of this value and always verify that either the stack is non-empty before requesting the pop operation or check the return value of each pop operation.

A non-standard object-oriented model can be defined in which the collection of methods in a class definition are partitioned into subsets with respect to the state of an object. Depending on the object state, only a subset of the methods declared in the class definition are available for execution. In a non-concurrent object-oriented language, objects with such semantics may or may not be useful. In a concurrent object-oriented language, mechanisms based on such semantics provide a natural and elegant means for expressing synchronization control.

In this paper we are interested in concurrent objects which by necessity employ some form of synchronization control. In particular, we are interested in specializing the concurrent behavior of such objects. The concurrent behavior of an object is captured in part by the class definition of the object and in part by the mechanism employed by the class to guarantee synchronization. The inheritance anomaly occurs when we attempt to specialize concurrent behavior using an inheritance mechanism. The anomaly occurs because the inheritance mechanism and the synchronization mechanism seem to interfere with one another, limiting the ability of the subclass to reuse the method implementations of the superclass. Furthermore, the anomaly has been observed across a spectrum of concurrent object-oriented languages regardless of the kind of synchronization mechanism employed [America 1987], [Briot 1990], [Kafura 1989], [Nierstrasz 1987].

There is a general consensus that we do not yet fully understand what it means to inherit concurrent behavior [Wegner 1990b]. In the remainder of this paper, we address this issue in a formal way. A formalism is presented which exposes the essential elements of concurrent object behavior and leads to conditions which must exist if the inheritance anomaly is to be avoided.

3 Defining Concurrent Object Behavior

When we speak of the behavior of an object, we are concerned with the set of messages that an object will accept at a given point in time, or alternatively, the set of methods that are visible in the interface of the object upon receipt of a message. From this perspective, the behavior of an object is its *observable behavior* since we are concerned with how the object appears to those clients that communicate with the object. Our notion of observable behavior is motivated by the similar notion described in [Milner 1989]; however, in this paper the machinery of CCS is used in specifying and reasoning about the observable behavior of individual objects, not systems of objects.

In dealing with concurrent objects, the relationship between the state of an object and the subset of methods which define its observable behavior is critical. This relationship is precisely what defines the behavior of a concurrent object. In order to understand concurrent object behavior, we must investigate this relationship.

3.1 Specifying Behavior

We may define the behavior of an object as a set of *behavior equations* which capture the states of an object and the subset of methods which are visible when the object is in a particular state. As an example, we define the behavior of an object that maintains some prescribed linear ordering over a collection of items and whose size is bounded. We call such an object a bounded linear ordering and its behavior is described by the following equations:

$$\begin{aligned} A_0 &\overset{\text{def}}{=} \mathtt{in}(x).A_1 \\ A_1 &\overset{\text{def}}{=} \mathtt{in}(x).A_2 + \overline{\mathtt{out}}(x).A_0 \\ &\vdots \\ A_n &\overset{\text{def}}{=} \overline{\mathtt{out}}(x).A_{n-1} \end{aligned}$$

This set of behavior equations is similar to an example in [Milner 1989]. The equations capture precisely the states that an object representing a bounded linear ordering may occupy during its lifetime. In the equations, we use only the Prefixing (.) and Summation (+) combinators of Milner's calculus. In each of the equations the name on the left-hand side denotes an agent whose behavior is defined by the right-hand side. Intuitively, agent A_i represents the behavior of the object when the size of the collection is i, where $0 \leq i \leq n$. One can verify, through recursive substitution, that this set of equations defines all possible behaviors of a bounded linear ordering.

In the behavior definition of the A_1 agent, the Summation combinator conveys that the agent offers both the `in` and $\overline{\texttt{out}}$ operations simultaneously to a client. If the `in` operation is chosen, the Prefix combinator requires that an agent accept an input value denoted by x and then become agent A_2. Similarly, if the $\overline{\texttt{out}}$ operation is chosen, the agent outputs a value denoted by x and then assumes the behavior defined by agent A_0.

In general, we say that agent A_i becomes agent A_{i+1} following an `in` operation and agent A_{i-1} following an $\overline{\texttt{out}}$ operation, with the behavior of agents A_0 and A_n being special cases. From this perspective, the behavior equations define the operations offered by an agent as well as a replacement behavior. The notion of replacement behavior is a fundamental aspect of the Actor model [Agha 1986]. Hence, it seems appropriate to use behavior equations as a formal means for specifying and reasoning about the behavior of individual actor-like objects.

Although we have specified a generic bounded linear ordering, the above set of behavior equations is isomorphic to a set of equations representing a bounded buffer accepting *put* and *get* operations, a stack accepting *push* and *pop* operations, or a queue accepting *enqueue* and *dequeue* operations. The isomorphism is realized through an application of the CCS Relabeling operator to yield the desired name substitution:

$$\begin{aligned}
\text{Buffer} &\equiv A_0[\texttt{in/put}, \texttt{out/get}], \ldots, A_n[\texttt{in/put}, \texttt{out/get}] \\
\text{Stack} &\equiv A_0[\texttt{in/push}, \texttt{out/pop}], \ldots, A_n[\texttt{in/push}, \texttt{out/pop}] \\
\text{Queue} &\equiv A_0[\texttt{in/enqueue}, \texttt{out/dequeue}], \ldots, A_n[\texttt{in/enqueue}, \texttt{out/dequeue}]
\end{aligned}$$

We can achieve the isomorphism because at this level of abstraction we are not concerned with the actual semantics of the `in` and $\overline{\texttt{out}}$ operations, e.g., whether or not the $\overline{\texttt{out}}$ operation returns values according to FIFO or LIFO semantics. To maintain generality, the equations describing a bounded linear ordering are used in the remainder of this paper with the understanding that we can apply the isomorphism at any time.

3.2 Object States and Behavior Sets

Behavior equations may be viewed as defining independent agents representing the various states of an object. In this section a model is defined which captures the essential elements used in developing a programming abstraction to represent a collection of behavior equations.

In our model, we associate with each A_i a state σ_i and an set β_i called the *observable behavior set*. For a given behavior equation A_i, the observable behavior set β_i is constructed from the non-restricted prefix operations on the right-hand side of behavior equations. By non-restricted prefix operations, we mean those operation names that do not appear within the scope of the CCS Restriction operator, thereby removing those operations from the set of observable behaviors. The collection of all states is given by the state set $S = \{\sigma_0, \sigma_1, \ldots, \sigma_n\}$. The set of all possible observable behavior sets is the powerset $B = \mathcal{P}(M)$, where $M = \bigcup_{i=0}^{n} \beta_i$. To complete our model, we also need a function which relates states to behavior sets.

3.3 Mapping States to Behavior Sets

We can define a function $f_\beta : S \to B$ which maps elements of the state set to elements of the powerset of observable behaviors. We might argue that in developing an abstract data type of a bounded linearly ordered structure in the standard model, a programmer implicitly defines a mapping from S to B. More precisely, each σ_i is *always* mapped to a single element in B, namely M, where $M = \{\mathtt{in}, \overline{\mathtt{out}}\}$. We call f_β the *behavior function* since it defines the observable behavior of an object in any given state. In the standard model f_β is defined as:

$$\begin{aligned} f_\beta(\sigma_0) &= M \\ f_\beta(\sigma_1) &= M \\ &\vdots \\ f_\beta(\sigma_n) &= M \end{aligned}$$

That is, objects in the standard model alway have the same observable behavior regardless of the object state.

We argue that the definition of f_β in the standard model is unnatural. Because $f_\beta(\sigma_0) = \{\mathtt{in}, \overline{\mathtt{out}}\}$, we are forced to write the method implementing the $\overline{\mathtt{out}}$ operation in such a way that an underflow condition is detected. A similar situation occurs for $f_\beta(\sigma_n)$. What is needed is a more natural mapping for f_β. A more natural mapping from S to B can be given as follows:

$$\begin{aligned} f_\beta(\sigma_0) &= \{\mathtt{in}\} \\ f_\beta(\sigma_1) &= M \\ &\vdots \\ f_\beta(\sigma_n) &= \{\overline{\mathtt{out}}\} \end{aligned}$$

This new mapping corresponds to our intuition about the observable behavior of a bounded linear ordering.

Consider that one wants to realize the behavior captured by the behavior equations by implementing an object in an appropriate object-oriented language which embodies the abstraction of a linear ordering. We observe from the mapping f_β that a useful abstraction is to partition the above behavior equations into three sets based on the notion of the state of the object. Such a partitioning appeals to our intuition about the behavior of a linear ordering. Furthermore, we only need to reason about three behaviors, not n; that is, we reason that the structure is either empty, full, or somewhere in between. When we implement the abstraction just formed, we will define a class which exports two methods, **in** and **out**, and which either explicitly or implicitly implements a synchronization mechanism which is consistent with the behavior equations previously formulated.

Suppose now that we wish to introduce a new constraint on the behaviors. For example, we distinguish the behavior given by the A_1 equation as being different from the $A_2, \ldots, A_{n-1}$ behaviors because a new operation is introduced which augments the behavior sets of agents $A_2, \ldots, A_{n-1}$. In distinguishing the A_1 behavior, we must define a new partitioning different than the one previously formed. We now distinguish the conditions empty, full, singleton, and somewhere in between.

If we attempt to specialize the previously implemented abstraction through inheritance we find that we have to redefine the mapping given by f_β. Redefining the mapping means that we have to change the domain S, the codomain B, and the mapping of elements in S to elements in B. If we have implemented the abstraction is such a way that these components are implicitly imbedded in the implementation, then we run into the inheritance anomaly. That is, because the components of the mapping f_β are implicitly imbedded in the implementation, the only way we can redefine the mapping to give us the new synchronization control we desire is by reimplementing the methods in which the mapping components are embodied. This renders the inheritance mechanism useless. The solution, as we shall see in the next section, is to dissassociate the components of the mapping f_β from the method implementations and make them explicit.

4 Inheriting Concurrent Behavior in ACT++

The types of concurrent object-oriented systems we are interested in are composed of actor-like objects with properties similar to those described in [Agha 1986]. Each object possesses its own thread of control and communicates with other objects via message passing. Concurrency in our system is limited to inter-object concurrency which is achieved using message passing and an actor-like *become* operation. The become operation results in a *replacement behavior* (object) with its own thread of control. Fine-grained intra-object concurrency is not a feature of objects in our system.

We are specifically interested in expressing and inheriting concurrent object behavior in ACT++ [Kafura 1990], [Lee 1990], a prototype object-oriented language based on the Actor model and C++ [Ellis 1990]. ACT++ is a collection of classes which implement the abstractions of the Actor model and integrates these abstractions with the encapsulation, inheritance, and strong-typing features of C++. The language falls in the heterogeneous category of concurrent object-oriented languages [Papathomas 1989] since we have both active and passive objects. Active objects are instances of any class derived from a special *Actor* class. Any instance of a class not derived from the Actor class is a passive object.

Concurrency is achieved using an actor-like become operation which is implemented in the Actor class. The become operation permits an object to specify a replacement behavior.

The notion of *behavior abstraction* was previously proposed in ACT++ [Kafura 1989] as a mechanism for capturing the behavior of an object. Upon initial examination, behavior abstraction seems powerful since synchronization can be achieved naturally by dynamically modifying the visibility of the object interface using the become operation. The efficacy of this mechanism and its degree of interaction with the ACT++ inheritance mechanism has been examined by others [Papathomas 1989], [Matsuoka 1990] and has been found to have serious limitations. The most serious limitation occurs because a behavior abstraction is not a first-class entity in the language and is thus subject to the effects of the inheritance anomaly.

Enabled sets [Tomlinson 1989] improve on the notion of behavior abstraction by promoting the control of the visibility of an object's interface to a dynamic mechanism which can be manipulated within the language; i.e., objects in Rosette are first-class entities.

The flexibility offered by enabled sets lead us to investigate the combination of behavior abstraction and enabled sets which resulted in the notion of a behavior set as introduced in the previous section. The ACT++ mechanism which captures the idea of a behavior set has the following properties:

- it is a natural extension of formal methods for specifying concurrent object behavior,
- it does not interfere with the ACT++ inheritance mechanism,
- it is free from known inheritance anomalies,
- it can be expressed entirely within ACT++, and
- it can be enforced efficiently at run time.

We use ACT++ in the following sections to illustrate how to express elements of the object state set S, elements of the observable behavior powerset B, and the behavior function f_β, such that concurrent behavior may be defined and inherited free from known anomalies

4.1 Expressing Concurrent Behavior

To represent concurrent object behavior within the ACT++ language, we rely on three first-class entities expressible within the language:

1. *state functions* representing some or all of the elements of the state set S,
2. a *next behavior function* representing the function f_β, and
3. *behavior sets* representing elements of the observable behavior powerset B.

In the example shown in Figure 1, we demonstrate how each of these entities is expressed and used in an ACT++ class definition of a bounded linear ordering.

In this example, we use two boolean functions `empty` and `full` to distinguish three states: empty, full, and neither empty nor full. Although not shown, the functions are computed based on implementation dependent instance variables representing the actual number of elements in the linear ordering. Both functions are used by the `nextBehavior` function which maps the current object state to a behavior set represented by an instance of the `BehaviorSet` class. There are three behavior sets defined: `Zero`, `N`, and `Other`. The `Zero` and `N` behavior sets correspond to the previously expressed behavior equations A_0 and A_n, respectively. The `Other` behavior set is used in this abstraction to collectively represent the observable behaviors of the intermediate behavior equations $A_1, A_2, \ldots, A_{n-1}$. Each behavior set is initialized in the class constructor. Instances of the `BehaviorSet` class are first-class objects and we have given an overloading to the binary `+` operator denoting set union when applied to two behavior sets; hence, `Other` is formed as the union of the behavior sets `Zero` and `N`.

4.2 Inheriting Concurrent Behavior

To substantiate our claim that the inheritance anomaly is avoided, we derive from `LinearOrd` a new class called `HybridLinearOrd`. The main feature of `HybridLinearOrd` is that a new method is introduced which forces us to change the mapping given by f_β. The new method allows a client of an instance of the `HybridLinearOrd` class to atomically read a pair of elements instead of a single element. The method cannot simply invoke the `out` method

```
 class LinearOrd : Actor {

     ... // private instance variables

 protected: // instance variables and methods visible to subclasses

     BehaviorSet Zero, N, Other;

     virtual bool empty () { ... }
     virtual bool full () { ... }

     virtual BehaviorSet nextBehavior () {
         if (empty ())
           return Zero;
         else if (full ())
           return N;
         else
           return Other;
     }

public:     // methods visible to subclasses and clients

    void in (int x) {
      ...
      become nextBehavior ();
    }

    int out () {
      ...
      become nextBehavior ();
    }

    LinearOrd () {      // construct initial empty object
      ...
      Zero = BehaviorSet (&in);
      N  = BehaviorSet (&out);
      Other = Zero + N;
      become nextBehavior() ;
    }
};
```

Figure 1: The LinearOrd Class Definition

twice since the **out** method executes a **become** operation after each invocation. Due to the concurrency in the system, another object may have its **out** request executed before the second **out** is processed. We specify the behavior of this new type of object with the following behavior equations:

$$\begin{aligned}
A_0 &\overset{\text{def}}{=} \mathtt{in}(x).A_1 \\
A_1 &\overset{\text{def}}{=} \mathtt{in}(x).A_2 + \overline{\mathtt{out}}(x).A_0 \\
A_2 &\overset{\text{def}}{=} \mathtt{in}(x).A_3 + \overline{\mathtt{out}}(x).A_1 + \overline{\mathtt{outpair}}(x,y).A_0 \\
&\vdots \\
A_n &\overset{\text{def}}{=} \overline{\mathtt{out}}(x).A_{n-1} + \overline{\mathtt{outpair}}(x,y).A_{n-2}
\end{aligned}$$

The behavior equations for a hybrid linear ordering differ from the equations specifying the behavior of a linear ordering only in the addition of the choice of an $\overline{\mathtt{outpair}}$ operation in the definitions of the A_2 through A_n behaviors. There are two effects of this refinement. First, we need to add the $\overline{\mathtt{outpair}}$ operation to the observable behavior set and compute a new powerset B'. Second, since we now distinguish the A_1 behavior and since $B' \supset B$, a new mapping f'_β is required:

$$\begin{aligned}
f'_\beta(\sigma_0) &= f_\beta(\sigma_0) \\
f'_\beta(\sigma_1) &= f_\beta(\sigma_1) \\
f'_\beta(\sigma_2) &= \{\mathtt{in}, \overline{\mathtt{out}}, \overline{\mathtt{outpair}}\} \\
&\vdots \\
f'_\beta(\sigma_n) &= \{\overline{\mathtt{out}}, \overline{\mathtt{outpair}}\}
\end{aligned}$$

Clearly there is cause to reuse the implementations of the `in` and `out` methods defined in the `LinearOrd` class. In addition, we can see from the new mapping f'_β that we can also reuse the `nextBehavior` function. In order to do so, however, we must redefine both the `Other` and `N` behavior sets to include the method representing the $\overline{\mathtt{outpair}}$ operation. We must also define a new instance of the `BehaviorSet` class containing the methods representing the `in` and $\overline{\mathtt{out}}$ operations defined in the A_1 behavior equation.

The definition of the `HybridLinearOrd` class shown in Figure 2 inherits from the `LinearOrd` class and introduces the following:

- a new state function `singleton`,
- a new behavior set `One`,
- a redefinition of the behavior function `nextBehavior`, and
- a new method `outPair`.

The `singleton` function corresponds to distinguishing the agent A_1 from the $A_0, A_2, \ldots, A_n$ agents, and the new instance of the `BehaviorSet` class corresponds to the behavior set associated with agent A_1. The `Other` and `N` behavior sets are augmented in the class constructor with the `outPair` method corresponding to the enlarged codomain B'. Thus, the inherited `nextBehavior` function can be trivially redefined to correspond to the new mapping f'_β by only adding a check for the state corresponding to agent A_1 and invoking the superclass behavior function `LinearOrd::nextBehavior` for all other states.

Inheriting concurrent behavior means that we can reuse the superclass methods *and* specialize the mapping given to f_β by the superclass. This means the elements of S, the elements of B, and the function f_β must be representable in the language and the representations must be:

1. first-class,
2. inheritable, and
3. mutable.

The inheritance anomaly occurs in previous formulations of this problem precisely because the behavior sets and the behavior function, as they occured in the superclass, were neither first-class nor mutable.

State functions representing elements of S, instances of the `BehaviorSet` class representing elements of B, and the `nextBehavior` function representing f_β have these properties. All are first-class language entities inheritable by a `public` subclass. Instances of the

```
 class HybridLinearOrd : public LinearOrd {

     ...

 protected:

     BehaviorSet One;

     bool singleton () { ... }

     BehaviorSet nextBehavior () {
         if (singleton ())
           return One;
         else
           return LinearOrd::nextBehavior ();
     }

 public:

     intPair outPair () {
       ...
       become nextBehavior ();
     }

     HybridLinearOrd () {
         BehaviorSet Tmp(&outPair);
         One = Other;
         Other = Other + Tmp;
         N = N + Tmp;
     }
};
```

Figure 2: The HybridLinearOrd Class Definition

`BehaviorSet` class are mutable by a subclass since they are within the scope of a `protected` clause. The `empty` and `full` functions representing object states and the `nextBehavior` function representing the behavior function are mutable because they have the `virtual` attribute. These functions are also within the scope of the `protected` clause; the purpose being to hide them from clients of the subclass.

5 Summary and Status

We have attempted to explicate the relationship between concurrent object behavior and inheritance. In doing so, we are forced to first define the meaning of concurrent object behavior as it occurs in our actor-based concurrent object-oriented language. We have offered a formalized approach for specifying and reasoning about concurrent object behavior based on CCS behavior equations. This approach emphasizes the relationship between the state of an object and subsets of the set of methods in the interface to the object, called behavior sets. This relationship is embodied in the mapping given by the behavior function. If the inheritance anomaly is to be avoided, behavior sets and the behavior function must be first-class, inheritable, and mutable. We have shown that the language mechanisms of ACT++ (and therefore C++) are sufficiently expressive in this regard.

We are continuing to refine our ideas about what it means to inherit concurrent object behavior. We are exploring the ideas presented here in the context of distributed object-oriented systems with a high degree of both intra-node and inter-node concurrency. In particular, we have developed an object-oriented structure for the upper layer ISO protocols and we are investigating concurrency issues.

We are also addressing the semantic issues in a more rigorous fashion than is presented here. We suspect that type-theoretic semantics currently applied to object-oriented languages are incapable of addressing the temporal nature of a changing object interface as captured by the behavior function. Interesting work in this area is [Nierstrasz 1990] which also uses CCS as a starting point.

We have not discussed in this paper the run-time enforcement of behavior sets. We are experimenting with a binary overloading of the C++ method invocation operator (`->()`)

and a reinterpretation of the message passing semantics of ACT++. A subject of our current research is to determine the relationship between our implementation approaches and others based on reflection. We do not currently know how the approach in this paper relates to approaches based on reflective languages.

The ACT++ prototype continues to evolve as we gain understanding about the semantic issues underlying concurrent object-oriented languages. We have implemented ACT++ over the Experimental Systems Kit (ES-Kit) [Leddy 1989], [Joshi 1990]. ES-Kit is a distributed object-oriented run-time system which allows us to experiment with ACT++ on a network of Sun workstations. We are currently implementing behavior sets in ACT++ and integrating the behavior set implementation with PRESTO [Bershad 1988]. PRESTO is a C++ based threads package which will allow us to experiment with ACT++ on the Sequent Symmetry, a shared memory multiprocessor.

Acknowledgements

The previous work of Keung Hae Lee on behavior abstraction in ACT++ and the work on enabled sets in Rosette by Chris Tomlinson and Vineet Singh set the stage for the results described here. More recently, the presentations and discussions during the Workshop on Object-based Concurrency at the 1990 joint ECOOP/OOPSLA conference motivated us to explore exactly what we mean by object behavior and to investigate the relationship between inheritance and concurrent object behavior in our actor-based language. We thank Satoshi Matsuoka for originally formulating the inheritance anomaly in the context of behavior abstractions in ACT++.

References

[Agha 1986] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[America 1987] Pierre America. "Inheritance and subtyping in a parallel object-oriented language," *ECOOP'87 Proceedings*, pp. 234-242, Springer-Verlag, 1987.

[Bershad 1988] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. "PRESTO: a system for object-oriented parallel programming," *Software Practice and Experience*, 1988.

[Briot 1990] Jean-Pierre Briot and Akinori Yonezawa. "Inheritance and synchronization in object-oriented concurrent programming," in *ABCL: An Object-Oriented Concurrent System*, (ed. A. Yonezawa), MIT Press, 1990.

[Ellis 1990] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[Joshi 1990] Nandan Joshi. *A Distributed Implementation of ACT++*, M.S. Thesis (in preparation), Department of Computer Science, Virginia Tech, Blacksburg, Virginia, 1990.

[Kafura 1989] Dennis G. Kafura and Keung Hae Lee. "Inheritance in actor based concurrent object-oriented languages," *ECOOP'89 Proceedings*, pp. 131-145, Cambridge University Press, 1989.

[Kafura 1990] Dennis Kafura and Keung Hae Lee. "ACT++: building a concurrent C++ with actors," *Journal of Object-Oriented Programing*, Vol. 3, No. 1, pp. 25-37, May/June 1990.

[Leddy 1989] Bill Leddy and Kim Smith. "The Design of the Experimental Systems Kernel," *Proceedings of the Conference on Hypercube and Concurrent Computer Applications*, Monterey, CA, 1989.

[Lee 1990] Keung Hae Lee. *Designing a Statically Typed Actor-Based Concurrent Object-Oriented Programming Language*, Ph.D. Dissertation, Department of Computer Science, Virginia Tech, June 1990.

[Matsuoka 1990] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. "Analysis of inheritance anomaly in concurrent object-oriented languages," extended abstract presented at the ECOOP/OOPSLA'90 Workshop on Object-based Concurrency, October 1990, to appear in *SIGPLAN Notices*.

[Milner 1989] Robin Milner. *Communication and Concurrency*, Prentice-Hall, 1989.

[Nierstrasz 1987] Oscar Nierstrasz. "Active objects in hybrid," *OOPSLA'87 Proceedings*, pp. 243-253, 1987.

[Nierstrasz 1990] Oscar Nierstrasz and Michael Papathomas. "Towards a type theory for active objects," in *Object Management*, pp. 295-304, (ed. D. Tsichritzis), Centre Universitaire D'Informatique, Université De Geneva, 1990.

[Papathomas 1989] M. Papathomas. "Concurrency issues in object-oriented languages," in *Object Oriented Development*, pp. 207-245, (ed. D. Tsichritzis), Centre Universitaire D'Informatique, Université De Geneva, 1989.

[Tomlinson 1989] Chris Tomlinson and Vineet Singh. "Inheritance and synchronization with enabled-sets," *OOPSLA'89 Proceedings*, pp. 103-112, 1989.

[Wegner 1990a] Peter Wegner. "Concepts and paradigms of object-oriented programming," *OOPS Messenger*, Vol. 1, No. 1, pp. 7-87, August 1990.

[Wegner 1990b] Peter Wegner. Discussion Panel on Issues in Object-based Concurrency, held in conjunction with ECOOP/OOPSLA'90, October, 1990.