Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*

— Draft: Not for Wide Distribution —

Satoshi Matsuoka and Akinori Yonezawa†
Department of Information Science, The University of Tokyo

Abstract

It has been pointed out that *inheritance* and *synchronization constraints* in concurrent object systems often conflict with each other, resulting in *inheritance anomaly* where re-definitions of inherited methods are necessary in order to maintain the integrity of concurrent objects. The anomaly is serious, as it could nullify the benefits of inheritance altogether. Several proposals have been made for resolving the anomaly; however, we argue that those proposals suffer from the incompleteness which allows room for counterexamples. We give an overview and the analysis of inheritance anomaly, and review several proposals for minimizing the unwanted effect of this phenomenon. In particular, we propose (partial) solutions using (1) *computational reflection*, and (2) *transactions* in OOCP languages.

1 Introduction

Inheritance is the prime language feature in *sequential* OO (*Object-Oriented*) languages, and is especially important for code re-use. Another important feature is concurrency; although many OO languages in use today (such as C++ and Smalltalk) are sequential, it is natural to consider objects as being a unit of concurrency. A recent breed of OOCP (*Object-Oriented Concurrent Programming*) languages attempt to provide maximum computational and modeling power through concurrency of objects; in particular, our current prototype ABCL/onEM-4 language exhibits a real-life message passing latency of a mere 6 μseconds for two concurrent objects located on a separate physical node of a multicomputer[40].

Several researchers, however, have pointed out (albeit fragmentarily) the conflicts between inheritance and concurrency in OO languages[3, 17, 32, 35, 9]. More specifically, concurrent objects and inheritance seemingly have conflicting characteristics, thereby inhibiting their simultaneous use without heavy breakage of encapsulation. We have coined such a phenomenon as *inheritance anomaly* in OOCP. Its 'inauspicious' presence has persuaded OOCP languages *not* to support inheritance as a fundamental language feature. Some of the examples are families of Actor languages[18], POOL/T[3], Procol[37], and also, ABCL/1[42, 41]. There are other OOCP languages that do provide inheritance, yet are not concerned with the problems of conflicts — for those languages, we believe that the difficulties presented in this paper are unavoidable in practice.

*To be published in a forthcoming book on concurrent OO-computing edited by Gul Agha.

†Physical mail address: 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan. Phone 03-3812-2111 (overseas +81-3-3812-2111) ex. 4108. E-mail: {matsu,yonezawa}@is.s.u-tokyo.ac.jp

Inheritance anomaly entails a severe drawback for the development of large-scale and complex systems in OOCP languages, because there, the greatest benefits of using the OO framework are inheritance and encapsulation. It is therefore essential that clean amalgamation of inheritance and concurrency be achieved in order for large-scale systems to be constructed with OOCP languages. Unfortunately, previous work have largely neglected the proper analysis of the problem, and merely proposed ad-hoc solutions that are applicable for certain types of problems, but as we will see, are inapplicable for others. Instead, we argue that we must first analyze and categorize the conflicts, and based on the analysis, explore if an ideal solution is in fact possible.

The remainder of the paper is organized as follows: First, we give an overview of inheritance anomaly. We will then present non-trivial examples where the (rather simplistic) previous proposals for solutions are limited in their applicability. Next we will analyze and categorize the cause inheritance anomaly more generally. Finally, we examine some latest proposals for either solving or controlling the inheritance anomaly problem; in particular, we ourselves propose a scheme which utilizes a special form of *computational reflection* in OOCP.

2 Inheritance Anomaly in OOCP

One of the prime concerns in OOCP is *synchronization* of concurrent objects: when a concurrent object is in a certain state, it can accept only a subset of its entire set of messages in order to maintain its internal integrity. We call such a restriction on acceptable messages the *synchronization constraint* of a concurrent object. For example, consider a bounded buffer with methods `put()` and `get()`, where `put()` stores an item in the buffer and `get()` removes the oldest one; then, the synchronization constraint is that one cannot `get()` from a buffer whose state is *empty* and cannot `put()` into a buffer whose state *full* is likewise prohibited.

In most OOCP languages, the programmer explicitly programs the methods to control the set of acceptable messages for each object, in order to *implement* the object *behavior* that satisfy the synchronization constraint. *Synchronization code* is the term we use to refer to the portion of the method code where object behavior with respect to synchronization is controlled. The synchronization code must always be *consistent with* the synchronization constraint of an object; otherwise the object might accept a message that it really should not accept, resulting in a semantical error during program execution[1]. Here, in order to program the synchronization code, the programming language must provide some primitives for object-wise synchronization, such as semaphores, guards, etc.; we refer to the scheme for achieving object-wise synchronization using those primitives in the language as the *synchronization scheme* of the language.

Unfortunately, it has been pointed out that *synchronization code cannot be effectively inherited without non-trivial class re-definitions.* This conflict, which we have coined as *inheritance anomaly* in OOCP, has been identified by several researchers[17, 32, 35], although a comprehensive analysis has not been given yet to our knowledge. Inheritance anomaly is more severe than the violation of class encapsulation in sequential OO-languages that has been pointed out by Snyder[34], because in some of the schemes it is possible to create a general counterexample where NONE of the parent methods can be inherited. We will defer the more detailed analysis of inheritance anomaly until the latter sections; here, we identify the following situations where the benefits of inheritance is lost:

[1] Such a distinction between the synchronization constraints as a specification versus the behavior of the actual code that implements it, have not been clearly addressed in the previous literatures to our knowledge; in fact, the term 'synchronization constraints' has been confusingly used to mean both in various contexts.

1. Definition of a new subclass K' of class K necessitates re-definitions of methods in K as well as those in its ancestor classes.

2. Modification of a new method m of class K within the inheritance hierarchy incur modification of the (seemingly unrelated) methods in both parent and descendent classes of K.

3. Definition of a method m might force the other methods (including those to be defined at the subclasses in the future) to follow a specific protocol which would not have been required had that method not existed. Encapsulated definition of *mix-in classes* would thus be very difficult.

One notable fact is that the occurrence of inheritance anomaly *depends on* the synchronization scheme of the language; in other words, re-definitions would be required for classes in an OOCP language that adopted a certain synchronization scheme, while the (semantically identical) classes could be safely inherited in another language that provids an entirely different synchronization scheme. This implies that the heart of the problem is the semantical conflicts between the descriptions of object-wise synchronization and inheritance within the language, and not on how the language features are implemented underneath. Moreover, it is not immediately obvious whether previous techniques developed in concurrent/distributed languages and systems are applicable.

3 Inheritance Anomalies in the Previous Proposals

Recently, several proposals have been made for effectively allowing synchronization code to be inherited based on various synchronization schemes (examples are [2, 12, 17, 35, 30], among others). Some (although not all) of these proposals emphasized strong control over the conflicts a.k.a. inheritance anomaly, effectively claiming that synchronization code can be inherited for all common and/or necessary cases. Unfortunately, it is possible to show that such proposals still suffer from inheritance anomaly — in this section, we fortify this claim by presenting the actual (counter)examples of anomaly occurrence.

Before we proceed, however, we make a point that the proposals selected here are considered to be representative of certain classes of synchronization schemes, and the intention of the (counter)examples is to illustrate what type of inheritance anomaly would occur for such schemes. We do NOT intend to claim that a particular proposal is useless — as a matter of a fact, some do embody good ideas that could be used as a basis of a more complete solution.

3.1 Simple Examples of Inheritance Anomaly — Caused by 'Body's, Explicit Message Reception within Methods, Path Expressions, Direct Key Specifications

In order to gain the reader's insight into the problem, we first present simple examples of inheritance anomalies occurring in OOCP languages. Some of these cases have already been pointed out by the previous researchers.

3.1.1 Bodies

Some OOCP languages allow each object to have a so called '*body*', an internal method with its own thread of control. The body thread remains active irrespective of the external message

reception. The body is typically used to control message receptions, usually in the fashion of Ada's **select** statement. After receiving a message, the body thread takes on the responsibility of invoking the method corresponding to the message. In some languages, the body thread suspends during method processing, while in others the body thread runs independently of the threads for message processing.

America[3] discusses the difficulty of integrating inheritance with languages that allow bodies: On defining a subclass from another class, the definition of the subclass usually require *total re-definition* of the body. This is rather obvious, because otherwise the newly added features cannot be used. America points out that this poses difficulty in programming because having a different body means that the dynamic behavior of such a new object may be totally different from the old ones, thus severely interfering with formal reasoning about the program. America states that, after initial experiments with inheritance in the OOCP language POOL/T, it was decided not to adopt inheritance as a primitive language feature[2]. Another related difficulty we point out is that such re-definitions require total knowledge of and access to the synchronization code of the ancestor classes. Thus, not only that they cannot be inherited, but also encapsulation of class implementation is broken with respect to synchronization constraints.

As an example of the 'body' anomaly, consider a first-in first-out bounded buffer class as illustrated in Figure 1. It has two public methods, `put()` and `get()`; `put()` stores an item in the buffer and `get()` removes the oldest one. Two instance variables `in` and `out` count the total numbers of items inserted and removed, respectively, and act as indices into the buffer — the location of the next item to be put is indexed by (`in` *mod* `size`) and that of the oldest item by (`out` *mod* `size`). Upon creation, the buffer is in the empty state, and the only message acceptable is `put()`; arriving `get()` messages are not accepted but kept in the message queue un-processed. When a `put()` message is processed, the buffer is no longer empty and can accept both `put()` and `get()` messages, reaching a 'partial' (non-empty and non-full) state. When the buffer is full, it can only accept `get()`, and after processing the `get()` message, it becomes partial again.

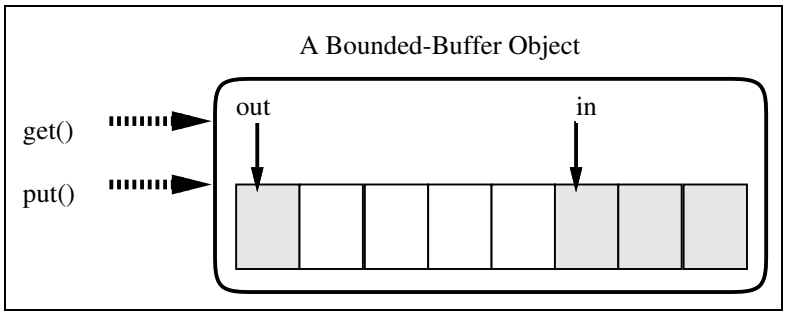

Figure 1: A Bounded Buffer Object

Figure 2 is a definition of class `b-buf` which implements the above described behavior, given with an extended syntax of C++ for reader familiarity. (Note that, some liberty is taken with the syntax and semantics — for instance, C++ does not provide the Smalltalk-80 style `super` pseudo variable, whose meaning should be obvious to those familiar in OO programming.). Explicit message reception is made within the body using the `select` and `accept` statements. The `get()` message is accepted by the first `accept` statement in the body if the buffer is not empty; then the actual `process_get()` method is invoked with the `start` statement. Upon its

[2]The recent version of POOL called POOL/I incorporates inheritance. Proper body re-definition is left as the responsibility of the programmer.

termination, the result of the method invocation is directly returned to the caller. Here, it is quite obvious that in any subclasses of `b-buf`, the entire `body()` must be re-defined in order to account for the newly added method definitions.

```
Class b-buf: ACTOR {// b-buf is an Actor
   int in, out, buf[SIZE];
public:
   void b-buf() {
      in = out = 0;
   }
   void process_put() { //store an item
      in++;
   }        //the argument of the call is omitted
   int process_get() {  //remove an item
      out++;
   }        //the return value of the call is omitted
   void body() {
      loop {
         select {
            accept get() when (!(in == out))
               start process_get();
         or
            accept put() when (!(out = in + SIZE))
               start process_put();
}}}}
```

Figure 2: Definition of Bounded Buffer Class with Body (The code related to accessing the local array storage for insertion and removal is omitted for brevity.)

There are several languages that allow body within objects ([7, 12]. [13] also essentially allows bodies when the 'low level' scheme is utilized).

3.1.2 Explicit Message Receptions

An analogous situation occurs if a language allows explicit (interior) reception of messages within a method, in that the newly added method definitions cannot be entirely accounted for. Therefore it would be difficult to incorporate inheritance into languages that allow interior message receptions. Examples of such languages are ABCL/1[42, 41] and CSSA[29].

There are also languages that extends existing sequential OO languages with explicit message reception statements in order to achieve inter-object concurrency, such as Concurrent C++[15], Buhr et. al.'s extension to C++[10], or Tuple Space Smalltalk[22]. For these languages, however, the messages explicitly received are not processed via the normal method dispatch mechanism of the base language. As a result, inheritance and communication are totally separated from the beginning, causing heavy breakage of encapsulation.

3.1.3 Path Expressions

Again, a similar problem occurs for languages with synchronization schemes expressed in variants of Path Expressions[11]. Additionally, the original path expression suffers the limitation

that is imposed by the expressive power of Path Expressions with respect to complex synchronization constraints of objects. For instance, the textual length of the path expression of the above bounded buffer example would be enormous for a large `SIZE`, because one must account for every possible combinations of interleaved `put`s and `get`s; more specifically, the expressive power of the original Path Expressions is limited to the regular expression, whereas the bounded buffer require a more powerful language class for concise description. This can be resolved with augmenting the terms in the path expression with guards and thereby allowing conditional synchronization[6]. An example OOCP language with augmented Path Expression is Procol[37]. Nevertheless, the original problem is not resolved, because one still cannot account for the newly added methods in the subclass unless the entire path expression is re-defined.

3.1.4 Direct Key Specifications

One very important classification of inheritance anomaly is its occurrence in the synchronization schemes involving operations with message keys. We refer to this as the *direct key specification anomaly*. The primary reason for anomaly is that the newly added keys in the subclasses cannot be accounted for in the synchronization scheme of the methods inherited from the parent methods. Languages employing this type of synchronization schemes such as SINA[36][3] or OTM[16] would suffer from the inheritance anomaly if they were to be extended to incorporate inheritance. (For the example of the anomaly occurring with bounded buffers, see[17].)

3.2 Problems with *Behavior Abstractions*

Kafura et. al's proposal called the *behavior abstraction*[17] attempts to solve the above problems, especially the problem with direct key specifications, in the context of their language ACT++. The essence of their proposal is to assign identifiers to *accept sets*, namely, the set of keys of messages that can be accepted by an object.

Figure 3 is the definition of the bounded-buffer object with behavior abstractions. We basically adopt a simple Actor-like language, whereby:

- Each object is single threaded i.e., an object can only accept one message at a time.
- Message passing is asynchronous, and pending messages are placed in the message queue.
- The next 'behavior' of the object is specified with the `become` primitive (see below).

The `behavior` statements declare three sets of keys named `empty`, `partial`, and `full` assigned to {`put`}, {`put`,`get`}, and {`get`}, respectively. The synchronization scheme employs the `become` statement to designate a set of method keys acceptable in the next state. We call such a set the *next accept set*. This set is not a first-class value; rather, another *key* is designated to each next accept set at the first part of a class definition.

Kafura describes in[17] how behavior abstraction serves as a clean solution to the anomaly exhibited in the `x-buf` example; there, `x-buf` has one additional method `last` that is similar to `get` — the difference is that it removes the last item previously put into the buffer instead of the first. In Figure 3, neither `put` nor `get` need to be re-defined in `x-buf`, whereas re-definitions of all the methods were necessary for the comparative language that could only specify the method keys.

[3] Although SINA does not support inheritance, there is an extension called Sina/ST[2] which employs pattern matching of method names and arguments in the similar manner as the path expression. Inheritance and delegation are simulated using this scheme. The path expression anomaly we have discussed in Section 3.1.3 would occur for this scheme.

```
Class b-buf: ACTOR {// b-buf is an Actor
   int in, out, buf[SIZE];
behavior:
   empty   = {put};
   partial = {put, get};
   full    = {get};
public:
   void b-buf() {
      in = out = 0;
      become empty;
   }
   void put() {
      in++; //store an item
      if (in == out + size) become full;
      else                  become partial;
   }
   void get() {
      out++; //remove an item
      if (in == out) become empty;
      else           become partial;
   }
}

Class x-buf: b-buf {// extends b-buf
behavior:
   x_empty   =               renames empty;
   x_partial = {put,get,last} redefines partial;
   x_full    = {get,last}     redefines full;
public:
   void x-buf() {
   }
   int last() {
      in-- ; //remove the last item
      if (in == out) become x_empty;
      else           become x_partial;
   }
}
```

Figure 3: `B-buf` and `x-buf` with Behavior Abstractions

Unfortunately, it is possible to create a non-trivial counterexample of inheritance anomaly with behavior abstractions. Consider creating a class `x-buf2`, a subclass of `b-buf`. `x-buf2` has one additional method `get2`, which removes the two oldest items from the buffer simultaneously. (Notice that this cannot be done with successive messages sends of `get`, because `get` messages from different objects may be interleaved.) The corresponding synchronization constraint for `get2` requires that at least two items exist. As a consequence, the partial state must be partitioned into two — the state in which exactly one item exists, and the remaining states. To maintain consistency with the new constraint, we need another accept set `x-one` that represents the former state (the `behavior` definitions in Figure 4). Then, the methods `get` and `put` *must be re-defined* (Figure 4). Here, notice that NONE of the methods (except the initializer) in `b-buf` can be inherited — the anomaly has occurred again[4].

```
Class x-buf2: b-buf { // x-buf2 is a subclass of b-buf
behavior:
   x_empty   =                renames empty;
   x_one     = {put,get};
   x_partial = {put,get,get2} redefines partial;
   x_full    = {get,get2}     redefines full;
public:
   void x-buf2() { in = out = 0; become x-empty; }
   void get2() { out += 2; //definition of get2
      if (in == out)          become x_empty;
      else if (in == out + 1) become x_one;
      else                    become x_partial;
   }
   //The following re-defines the methods in b-buf.
   void get() { out++;
      if (in == out)          become x_empty;
      else if (in == out + 1) become x_one;
      else                    become x_partial;
   }
   void put() { in++;
      if (in == out + size)   become x_full;
      else if (in == out + 1) become x_one;
      else                    become x_partial;
   }
}
```

Figure 4: Inheritance Anomaly with Behavior Abstractions

3.3 Problems with First-Classing of Accept Sets — *Enabled Sets*

Tomlinson and Singh[35] propose a scheme that enhances Kafura's in their Actor-based reflective language called Rosette. In Rosette, the accept sets can be treated as first-class objects called *enabled sets*. We show that this difference is essential, because their proposal can localize (although not eliminate) the method re-definitions in some cases. We also show, however, that there are still other cases that would require a considerable amount of re-definitions.

[4]Recently, they have proposed a more advanced scheme called *behavior sets*, which is similar in essence to Tomlinson and Singh's enable sets we discuss next.

Here is a brief overview of Rosette with respect to synchronization schemes: although its original syntax is based on S-expressions, we will continue to use our C++ based syntax with the following extensions:

- The `become` statement now specifies the next state and the next enabled set of the object:

  `become(`⟨*enabled-set*⟩`, (`⟨*new-state*⟩`))`

- An enabled set is an instance of class `Enable`; here the constructor adopts a special syntax whereby a set of message keys to be enabled are specified:

  `Enable(`⟨*message keys*⟩`)`

  There are several operations defined for the enabled set, such as union (`+`), intersection (`&`), etc.

- In order to specify the next enabled set for an object, we typically define a private method for each enable-set:

  `Enable` ⟨*method*⟩`() {return Enable(`⟨*message keys*⟩`))`

- There are two kinds of methods, *public* and *private*. The public methods are invoked as a result of a message reception from an external object. Message sending is asynchronous, and only those messages whose corresponding methods are currently 'enabled' by the enabled set can be accepted. On the other hand, the private methods are internal to the object and can be only invoked from within the public and private methods of the same object as a function call.

Now, consider defining, in addition to `get2`, method `empty?` which checks whether the buffer is empty or not. The method is in effect stateless, that is, it does not affect the state of the buffer. Thus, this message should always be acceptable irrespective of object state (as long as other methods are not executing). Then, in principle its definition should be independent of definitions of other methods, since the effects on the object state by other methods are irrelevant to `empty?`. But this is not the case — in the definitions of `b-buf` and `x-buf2` (Figure 5), we can observe the followings:

- We must override every single private methods that returns an enabled set so that it enables the `empty?` method (all the private methods of `x-buf2` in Figure 5).

- We must perform extensive case analysis of object state for the newly added method — this is necessary even if the method itself does not affect the state of the object.

The advantage of enabled-sets over behavior abstractions is that re-definition of the *parent methods*, although unavoidable, can sometimes be confined within private methods by inheritance. This is seen in Figure 5, where only (all the) private methods such as `empty` and `full` are re-defined. This possibility is due to the first-class nature of enabled sets derived naturally from the reflective language architecture of Rosette. We feel that reflective architecture is essential in OOCP languages[43], and this is one example of how it can be used to enhance the descriptive power of OOCP languages. For enabled-sets in particular, however, there are non-trivial cases where such re-definitions, even though confined, would nevertheless be overwhelming in comparison to method guards we discuss in the next section.

```
Class b-buf: ACTOR { // b-buf is an Actor
  int in, out, buf[SIZE];
private:
  Enable empty() { return enable([put]) };
  Enable partial() { return enable([put,get]) };
  Enable full() { return enable([get]) };
public:
  void b-buf() { in = out = 0;
    become(empty(),(in,out,buf));
  }
  void put() {
    if (in == out + size) become(full(),(in,out,buf));
    else                  become(partial(),(in,out,buf));
  }
  void get() { // Similar to put()...
}

// The entire private methods must be re-defined
Class x-buf2: b-buf {
private:
  Enable empty() {
    return Enable(empty?) + super empty()};
  Enable one() {
    return Enable(get,put,empty?) };
  Enable partial() {
    if (in == out + 1)
       return super partial + Enable(empty?);
    else
       return super partial() + Enable(get2,empty?) };
  Enable full() {
    return super full() + enable(empty?) };
public:
  void x-buf2() { in = out = 0; become x-empty; }
  void get2() { out += 2; // addition of get2()
    if (in == out)          become(empty(),(in,out,buf));
    else if (in == out + 1) become(one(),(in,out,buf));
    else                    become(partial(),(in,out,buf));
  }
  // Painstaking case analysis is necessary
  int empty?()[] { // addition of empty?()
    if (in == out)          become(empty(),(in,out,buf));
    else if (in == out + 1) become(one(),(in,out,buf));
    else if (in == out - 1) become(full(),(in,out,buf));
    else                    become(partial(),(in,out,buf));
  }
}
```

Figure 5: `X-buf2` with Enabled-Sets

Here, let us illustrate this by generalizing the method re-definitions of the enabled-sets. The private methods returning an enable set correspond to the 'states' distinguished at class K. On defining a method m at class *K-sub*, a subclass of K, the user needs to check, for each 'state', whether addition of m incurs partitioning of that state. If so, the predicate which determines the state may need to be partitioned. In Figure 6, this is done for the private methods `state_1` through `state_n`. In our `empty?` example, since the method was always acceptable, `Enable(...,empty?,...)` had to be added to ALL the private methods of `x-buf2`. Furthermore, on specifying the next behavior of m, the programmer must judge which of the states among those labeled `state_1` through `state_n` is appropriate, depending on the current state of the object (Figure 6).

```
Class K-sub: K { //K-sub is a subclass of K
  ⟨Instance Variable Definitions⟩
private:
  Enable state_1() {
    if (⟨method⟩ is acceptable)
      return Enable(⟨method⟩) + super state_1()
    else
      return super state1_() };
  //Repeat for state_2 through state_n
      ...
public:
  ⟨type⟩ ⟨method⟩(⟨args⟩...) {
    return ⟨value⟩;
    if      (Object is in state 1) become(state_1(),⟨new state⟩);
    else if (Object is in state 2) become(state_2(),⟨new state⟩);
          ...
    else if (Object is in state n) become(state_n(),⟨new state⟩);
  }
}
```

Figure 6: General Analysis of Enabled Set

Despite its limitations, we do strongly acknowledge the significance of Rosette in pointing out that the first-classing technique provides the possibility of enlarging the class of synchronization schemes that can be safely inherited. Later on, we will describe a more elaborate synchronization scheme intended for (partially) resolving the inheritance anomaly.

3.4 Problems with Method Guards

A natural synchronization scheme is to attach a predicate to each method as a guard, thus making each object a conditional critical region (for example, [14, 23] and indivisible objects in [19]). We illustrate this for `b-buf` and `x-buf2` in Figure 7. Here, we employ the following syntax:

⟨*method name*⟩(⟨*formal arguments*⟩) `when` (⟨*guard*⟩) `{` ⟨*body of method definition*⟩ `}`

where *guard* is a boolean expression whose terms are either constants or instance variables bound to primitive values. Method `m` is invoked only when *guard* evaluates to `True`. For instance, in

class b-buf, the guard (in < out + size) attached to put() assures that put() is not invoked when the buffer is full. As shown in Figure 7, all the methods defined at b-buf are inherited by x-buf2 without any changes to the methods or the guards.

```
Class b-buf: ACTOR {
  int in, out, buf[SIZE];
public:
  void b-buf()  { in = out = 0; }
  void put() when (in < out + size) { in++; }
  void get() when (in >= out + 1)   { out++; }
}

// x-buf is a subclass of b-buf
Class x-buf2: public b-buf {
public:
  void x-buf2()
  void get2() when (in >= out + 2) { out += 2; }
  void empty?() when (true) { return in == out; }
}
```

Figure 7: B-buf and x-buf2 with Method Guards

This scheme does provide an elegant solution to the get2/empty? example. Furthermore, although a naive implementation of guards is not usually very efficient, it can be improved with the use of program transformation[23] and other optimization techniques; and since they are usually invisible to the programmer, the full benefit of inheritance can be attained without sacrifices in efficiency.

However, the problem is that the occurrence of inheritance anomaly still cannot be prevented. This is a different kind of anomaly from the ones we have so far discussed in this paper. We will give two examples: one is the definition of the gget() method, and the other is the definition of the class Lock as a *mix-in class*.

First we consider defining gb-buf, a subclass of b-buf, adding a single method, gget(). The behavior of gget() is almost identical to that of get(), with the sole exception that it cannot be accepted immediately after the invocation of put. Such a condition for invocation cannot be distinguished with method guards and the set of instance variables available in b-buf alone; we need to define an extra instance variable after-put. As a consequence, both get() and put() must be re-defined as in Figure 8. We note that the analogous situation also occurs for accept set based schemes.

The reason for the anomaly occurrence is that we cannot judge the state for accepting the gget message with the guard declarations in b-buf. To be more specific, gget is a *trace-only or history-only sensitive* methods with respect to instances of b-buf; we will defer the discussion until the next section.

We next consider the Lock class, which is an abstract *mix-in* class[8]. Direct instances of Lock are not created; rather, the purpose of Lock is to be 'mixed-into' other classes in order to add the capability of locking an object. In Lock, a pair of methods lock and unlock have the following functionality: an object, upon accepting the lock message, will be 'locked', i.e., will suspend the reception of further messages until it receives and accepts the unlock message. Its synchronization constraint is *localized* i.e., it is not affected by methods of the class it is being

```
// gb-buf is a subclass of b-buf with gget()
Class gb-buf: b-buf {
   bool after-put;
public:
   void gb-buf() { after-put = False};

   // Definition of gget()
   void gget() when (!after-put && (in >= out + 1))
       { out++; after-put = False; }

   // The following methods must be re-defined
   void put() when (in < out + size) { in++; after-put = True; }
   void get() when (in >= out + 1) { out++; after-put = False; }
}
```

Figure 8: Inheritance Anomaly with Guards — the `gget` method

mixed into.

When `Lock` is 'mixed-into' the definition of `b-buf` to create the class `lb-buf`, we are likely to assume that it would not affect the definition of other methods, since the state of the object with respect to `lock` and `unlock` is totally orthogonal to the effect of other messages. However, this is not the case — first, we must add an instance variable `locked` which indicates whether the object is currently 'locked' or 'unlocked'; this is obviously necessary since it is impossible to distinguish between the two states otherwise. Then, the inherited methods such as `put` or `get` must be overridden in order to account for `locked` (Figure 9). Furthermore, all methods which would be defined in the subclasses of `lb-buf` must also account for `locked`. This would not have been necessary if we were to be defining exactly the same methods in the subclass of `b-buf`. To summarize, the effect of mixing-in `Lock` cannot be localized in `b-buf`.

Why has anomaly occurred here? Again, `lock` and `unlock` are history-only sensitive methods. In addition, although neither of them cause partitioning of states, they modify the synchronization constraints of the methods that are already defined, in this case both `put` and `get`. Thus, method guards of `b-buf` had to be modified in order to maintain consistency with the new constraints.

4 Analysis of Inheritance Anomaly

We have seen through examples that the previous proposals are not sufficient for avoiding the inheritance anomaly. We believe that their shortcomings are due to insufficient analysis of the situation; that is to say, the conflict we treat here is deeply rooted in the semantics of synchronization constraint/schemes verses semantics of inheritance, and analysis is first necessary for achieving a sufficiently clean solution.

There are three reasons why inheritance anomaly occurs, depending on what the subclass definition entails on how the *state* of the object upon which the messages are acceptable are modified:

- **Partitioning of Acceptable States** — `get2`, `gget`

```
Class Lock: ACTOR {
   bool locked;
public:
   void Lock() {locked = False};
   void lock() when (!locked} {lock = True};
   void unlock() when (locked) {lock = False};
}

// lb-buf is a subclass of b-buf with Lock mix-in
Class lb-buf: b-buf,Lock {
public:
   void lb-buf();
   // The following methods must be re-defined
   void put() when (!locked && (in < out + size)) { in++; };
   void get() when (!locked && (in >= out + 1)) { out++; };
}
```

Figure 9: Inheritance Anomaly with Guards — the `Lock` class

- **History-only Sensitiveness of Acceptable States** — `gget`, `lock`
- **Modification of Acceptable States** — `lock`

The three causes are relatively independent; for example, the `gget` partitions the states as well as being history-only sensitive.

4.1 Partitioning of States

The `x-buf2` example in Section 3.2 is a anomaly caused by *partitioning of acceptable states*. In object-oriented languages, an object is said to have some 'state'. Then, one can consider the 'set of states' an object can have. This set can be partitioned into disjoint subsets according to the synchronization constraint of the object; in the bounded buffer examples in Section 3, there are three distinguishable set of states, under which respective sets of acceptable messages can be defined: *empty*, *partial*, and *full*. This is conceptually illustrated by the left rectangle of Figure 10.

Now, when a new method is added in the definition of the subclass, the partitioning of the set of states in the parent class may need to be further partitioned in the subclass; this is because the synchronization constraint of the new method may not be properly be accounted for in the partitioning of the parent class. In our example, when the `get2()` method was added in `x-buf2`, a partitioning of x-partial into x-one and x-partial was necessary in order to distinguish the state at which only one element is in the buffer.

For accept set based synchronization schemes, this state partitioning is usually distinguished at the termination of the methods with some conditional statements, upon which the objects 'become' that state. This is seen for example in the definitions of `put` and `get` methods in Figure 3. Requirement for method re-definitions follows naturally, as we have illustrated in Figure 4, because the new partitioning must be accounted for in all the methods. Note that, this is not resolved by making accept sets first-class values, because this partitioning cannot be affected by the operations upon the accept-set data.

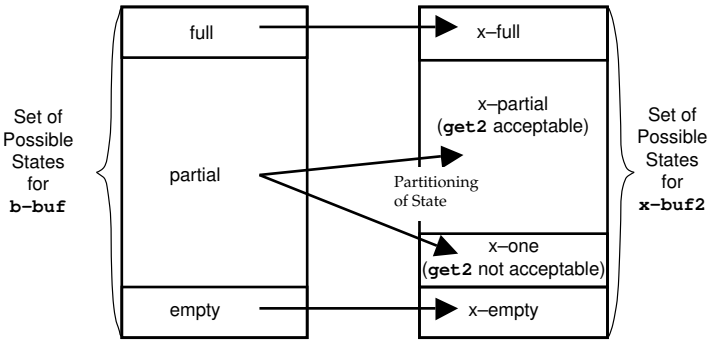

Figure 10: Conceptual Illustration of the State Partitioning Anomaly

This partitioning is not a problem for method guards, because they are able to directly judge whether the message is acceptable or not under the current state. Thus, even if the new methods were added, the guards would not need be re-defined, provided that it would not affect the partitioning of the methods in such a way that the condition denoted by a guard in a certain method would no longer be valid; this certainly holds for most cases of inheritance.

4.2 History-only Sensitiveness of States

When two different views in modeling the 'state' of objects. One is the *external view*, where the state is captured indirectly by the external observable behavior of the object. This view is taken by the models of parallelism based on process calculi, such as CCS[28] and Actors[1]; there, the equivalence of two objects are determined solely with how they respond to external experiments, and not with how their internal structures are constructed[5]. Another is the *internal view*, where the state is captured by the valuation of the state variables in the implementation of the object; for example, a Cartesian point object can have a valuation such that its x-coordinate is 3, and its y-coordinate is 5. (The actual semantics is more complicated by the fact that the valuation could be another object, and that objects have methods with **self** and **super** references.)

The two views on state are not identical; there are set of states whose elements can be distinguished under the external view, but is indistinguishable under the internal view. With method guards, in particular, only the latter states are distinguishable, because guards are usually boolean expressions consisting of constant object values, instance variables of the object, and various arithmetic/logical operators (other syntactic categories such as message keys are usually not allowed). Then, it follows that there exist some synchronization constraint that cannot be specified with a given set of instance variables and method guards: this is precisely the history information that do not manifest itself in the values of the instance variables.

When such a distinction becomes necessary, the state of the object under the internal view must be 'refined' in order to match the state of the external view. For this purpose, the methods in a parent class must be modified; that is to say, the state of the object is *history-only sensitive* with respect to the internally distinguishable ones. This is illustrated in our previous `gget` example in Section 8, where the state "immediately after accepting `put`" cannot

[5]To be more precise, the equivalence relation of objects are typically defined by the *bisimulation* relation. One could define several classes of bisimulation relations, yielding weaker or stronger equivalences according to his requirements, e.g. whether object congruence is required, etc.

be distinguished with the set of instance variables available in `b-buf`, requiring the addition of an instance variable `after-put`. Since the proper valuation of this variable must be done in all the methods, the requirements of method modification arose (The situation is similar for accept set based schemes in this respect, in that the `gget` example would require considerable re-definitions.). Also notice that `gget` partitions the state as well.

4.3 Modification of Acceptable State

The methods in the `Lock` example in Section 9 are *history-only sensitive* in a similar manner as `gget`. The difference from `gget` is that the execution of the methods in `Lock` modifies the set of states under which the methods inherited from the parent could be invoked (Figure 11). That is to say, mixing-in of `Lock` introduces finer-grained distinction for the set of states under which `get` (or `put`) in `lb-buf` can be invoked. This would naturally require the modification of the method guards to account for the new synchronization constraint. (Note that, although the history-only sensitive characteristics did not come into play for `Lock`, we could easily generate a case that does so; for example, we could define a mix-in class `Glock`, which would only allow locking of an object immediately after the acceptance of `put`.)

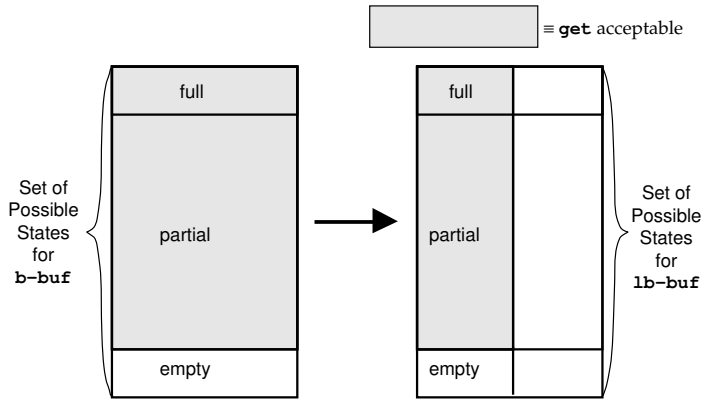

Figure 11: Conceptual Illustration of the State Modification Anomaly

4.4 Examples of Analysis of Anomaly Occurrence

Given the above categorizations, we could analyze the effectiveness of the synchronization schemes, and create an example of anomaly occurence. See Appendix A for brief analysis on *Synchronizing Actions* which was recently proposed[30].

5 Proposals for Solutions to the Inheritance Anomaly

Recently, there has been much research that have proposed to minimize the effect of inheritance anomaly in OOCP languages, effectively allowing inheritance of synchronization code in various situations. We will briefly review them as well as provide our own proposals.

5.1 Shibayama's Proposal

Shibayaba proposes a scheme based on fine-grained inheritance of synchronization schemes, so that the amount of code that must be re-defined can be minimized. In the proposed extension of ABCL/1 [33] to incorporate inheritance, methods are categorized into *primary*, *constraint*, and *transition* methods. A method of one category may have its counterparts with identical keys in other categories, and each of them can be separately defined/inherited/overridden. The categorization of methods are as follows:

- A *primary method* is responsible for the task other than object-wise synchronization.
- A *constraint method* acts as a method guard. Since it can be re-defined independently of the primary methods, only the constraint methods need to be overridden in the event that the guards of the methods of the parent class must be changed (the corresponding primary methods are unaffected).
- A *transition method* determines how the messages are delegated. Its re-definition allows dynamic modification of the delegation path.

By separating the synchronization code from other parts of method definitions, the amount of re-definitions is minimized. Shibayama also shows in [33] that history-only sensitivity can also be handled with a modest amount of code re-definitions in the concurrent implementation of a 2-3 tree.

5.2 Meseguer's Proposal

Meseguer proposes a new formalism[27] for modeling concurrent systems, and an OOCP language called Maude, which is based on this formalism. The language possesses the flexibility to provide cleans solutions for (some of the) anomalous examples we have presented in this paper.

Meseguer's formalism is a logic called the (concurrent) rewriting logic, which (Meseguer states that) most models of concurrent computation can be regarded as its special instantiations. A concurrent system is derived from (instantiations of) *modules*, that are composed of *terms* and rewrite rules. Computation proceeds by simultaneous simplification of terms when there are applicable rewrite rules. There are two types of modules, *functional* and *system*. The rewriting in system modules are not equational, i.e., does not exhibit the Church-Rosser property. This allows the modeling of phenomenon specific to concurrent computations, such as non-deterministic choice. The Maude language[27], based on this framework, provides *object-oriented modules* for ease of programming in concurrent object-oriented style. For actual execution, object-oriented modules are first translated into system modules; then, computation proceeds with concurrent rewriting according to the rewrite rules of the translated module. Inheritance is also supported in object-oriented modules directly with Maude's order-sorted type structure.

Inheritance anomaly is avoided in Maude in the following way: the conditions placed on the rewrite rules can serve as a guard; thus, state-partitioning anomaly does not arise. In addition, rewrite rules can be very flexible, operating on the term structures as first class values[26]. Thus, there is (albeit implicit) 'reflective' capability in Maude, which allow history information to be encoded within the term structure in a straightforward way. For example, it is simple to define a parametric class which adds the locking capability to arbitrary classes. There is still work needed to be done, however, to see the extend of applicability of Maude to other classes of inheritance anomaly.

5.3 Frølunds's Proposal

Frølund proposes a framework in which mostly concentrates on synchronization code for the derived (i.e., overridden) methods[20]. He proposes a design in which synchronization constraints get increasingly restrictive in subclasses. Basically, one specifies a guard that gives the condition under which the method *cannot be accepted*, i.e. a *negative* guard. Furthermore, the guard expressions are accumulated along the inheritance chain so that, given a method with the name `m`, all the guards for the methods in the ancestor classes with the name `m` and were thus overridden must evaluate to *false* in order for the message `m` to be accepted. Thus, the re-use only works in the way to restrict the conditions under which the messages are acceptable. Frølunds points out that this is reasonable given that it should be possible for superclass operations to work on (all) subclass state, i.e., if an ancestor operation is not enabled in a particular state, then a derived operation with extended behavior will also incur inconsistency in that state.

In addition, one could refer to other methods within the guard expressions; in this case, the method itself is not invoked, but instead, its guard(s) are evaluated and the resulting boolean value is returned. One is also able to describe synchronization constraints that should hold uniformly for all methods to be defined in subclasses, except for a set of certain exception methods. This allows one to program the `Lock` mixin-class in a similar strategy (albeit hard-codedly) as we present below.

By all means, the problem in inheritance anomaly is that it is not only the derived operations but also seemingly unrelated methods that might inadvertently require re-definitions. Nevertheless, Frølund's work is valuable in pointing out that reuse may or may not be possible without certain assumptions about how operations are derived.

5.4 Our Proposals

5.4.1 The Use of OOC-Reflective Architectures

The above proposals have identified that (1) separation of synchronization code from the method code, and (2) 'first-classing' of synchronization schemes keeps code re-definitions small. Our first proposal is along this line — by employing a reflective language, we encapsulate the different synchronization schemes in the meta-level. Efficiency is maintained by employing the lazy reification mechanism of the reflective language.

As an example, we present a scheme where we make the guards first-class objects in the meta-level. Since guards are immune to state partitioning, we attempt to either avoid or minimize the anomaly for history-only sensitive case such as `Lock`. The main idea is to manipulate the guards in a homogeneous way except for special methods in the subclasses where the exceptions occur. We also encapsulate meta-operations on the guards in the meta-level of the object.

ABCL/R2[24], an OOCP language with a *Hybrid Reflective Architecture*, is employed for this purpose. The *metaobject* of an object x, denoted as `[meta` x`]`, is a meta-level representation of the structure and computation of x. Here, `[meta` x`]` is itself a (concurrent) object[6] Given x, we can manipulate the guards as a first class-object with reflective operations via `[meta` x`]`. Figure 12 illustrates how the `Lock` mixin class can be programmed with this strategy[25]. (Here, we adopt an ABCL-family syntax: '`(...)`' indicates a LISP-like expression, and '`[... <== ...]`' indicates a message transmission. `Me` is identical to `self` in Smalltalk.)

In order to retain the efficiency of direct message dispatching, a technique which we call the

[6]Note that this aspect of reflective architecture — the *individual-based* architecture — was first introduced with ABCL/R[39]. ABCL/R2 also features reflection of object *groups*. See [24] for details.

```
(class Lock
  [state (Saved-Guards)]

  (=> [:lock]
     (Saved-Guards := [[meta Me] <== [:get-all-guards]])
     ;; below causes the metaobject to be coerced to {Meta-Object}.
     [[meta Me] <== [:set-all-guards! '#f]]
     [[meta Me] <== [:set-guard! 'unlock '#t]]]

  ;; not invoked until the guard is set to #t with :lock
  (=> [:unlock] when #f
     [[meta Me] <== [:set-each-guard! Saved-Guards]]))
```

Figure 12: Definition of Lock Mixin Class

dynamic progression of degree of reflectivity[7][25] is applied; basically, we modify the reflective architecture of an object on demand when a more elaborate reflective operation is necessary. In our example, the default metaobject of objects is an instance of the class `Lite-Meta-Object`. In this case, message dispatching would be very efficient, because the dispatching mechanism is hardwired into the system with various optimization schemes. The problem, however, is that only *reification* (i.e., to obtain the computational state of the object from the meta-level as a first-class object) of the guards is possible, and it is impossible to modify and *reflect* (i.e., to affect the computational state by 'reflecting' the data into the meta-level) the guards. This is not sufficient, because the definition of `Lock` in Figure 12 requires that the guards be modified.

To achieve the capability to reflect the guards, we have the class `Meta-Object`, which has the capability at the cost of efficiency of execution. Then, `:set-guard` and other methods which have side-effects on the guards in `Lite-Meta-Object` dynamically *coerces* the metaobject from `Lite-Meta-Object` to `Meta-Object` on demand. This allows us to maintain the efficiency provided by `Lite-Meta-Object` for most instances of subclasses of `Lock`. In addition, the coercion need to be performed only once; therefore, subsequent reflective operations on the guards (e.g., `:lock`) will require little overhead. For details, see [25].

5.4.2 Syntactic Elimination of Synchronization Codes

The second proposal is a totally different one, in which we attempt to **syntactically minimize the (amount of) synchronization code itself**. Much of the per-object localized synchronization code is employed to solve the inter-object coordination problem, where multiple objects compete for resources encapsulated within objects (the bounded buffer is a classic example). Here, if the inter-object consistencies are maintained *transparently* without the necessity for per-object localized synchronization code, anomaly does not occur in the first place.

In our prototype OOCP language HARMONY[38], we take the approach of *syntactically reducing or eliminating* the need for synchronization code with the embedded transaction feature of the system, instead of inventing various new synchronization schemes which might cause yet another anomaly. Distributed transactions facility supplemented with method guards are the basic synchronization scheme of objects. One need not employ guards as often, because inter-

[7] This technique is also employed in optimizing compiler for ABCL/R2: for details, see[21]

object synchronization for maintaining integrity is now implicit in the transaction facility of the system. With such a strategy, inheritance anomaly is less likely to occur, since there is less requirement for describing object-wise synchronization.

With a bounded buffer, for example, only the essential (guarded) `put` and `get` methods are necessary; compound methods known to cause anomalies, such as `get2`, are no longer necessary. This is because with HARMONY one can perform successive `get`s to the buffer with guarantee of atomicity without any programming of synchronization code such as locking. We refer the readers to a separate paper for the details of HARMONY[38].

(We make a note that, since the above two approaches are not contradictory, we can use both approaches to either eliminate or at least minimize the inheritance anomaly.)

6 Conclusion and Future Work

The prime objective of OOCP languages is to provide maximum computational power through concurrency of objects. At the same time, OOCP languages allow the system to be flexible and dynamically configurable. This effectively captures the essential properties of concurrent computational systems, which are highly complex and must change and evolve to adapt to the requirements of the user. Some ideas that have flourished in the sequential OO world, particularly inheritance, have similar objectives; but unfortunately, as we have shown, synchronization constraints and inheritance have conflicting characteristics, and thus it is difficult to combine them in a clean way. We have analyzed various types of inheritance anomaly and discussed several approaches to its solution. Our two proposals in particular were (1) first-class construction of synchronization schemes with *reflection* in OOCP languages, and (2) incorporation of *transactions* as a basic synchronization scheme which syntactically reduces or eliminates the need for synchronization code with the embedded transaction feature of the system.

In conclusion, in order for OOCP languages to be usable for large-scale programming, the inheritance anomaly needs more thorough theoretical analysis, plus derivation of a good solution. We need to strive on the followings:

- Establishing a more precise and formal definition of classification of inheritance anomaly. Although [23] made some preliminary formal analysis, it is still incomplete in that it only treats the anomaly that occurs with state partitioning. The work towards type theory for active objects[31], and recent work by America et. al. to separate the subtyping hierarchy from the inheritance hierarchy in the POOL family of OOCP languages[5, 4] could serve as a basis of more comprehensive formalism.

- Further identification of a general class of synchronization schemes with respect to anomaly classifications. Although we tried to be as comprehensive as possible by categorizing and selecting representative synchronization schemes, formal analysis might provide more insights for further sub-categorization.

- Proper development of languages (features) that either totally avoid or minimize inheritance anomaly, based on the above two analysis.

References

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

[2] Mehmet Aksit and Anand Tripathi. Data abstraction and mechanism in Sina/ST. In *Proceedings of OOPSLA'88*, volume 23, pages 267–275. SIGPLAN Notices, ACM Press, September 1988.

[3] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of ECOOP'87*, volume 276 of *Lecture Notes in Computer Science*, pages 234–242. Springer-Verlag, 1987.

[4] Pierre America. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of OOPSLA'90*, volume 25, pages 161–168. SIGPLAN Notices, ACM Press, October 1990.

[5] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands, May,1990*, number 489 in Lecture Notes in Computer Science, pages 60–90. Springer-Verlag, February 1991.

[6] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[7] J. P. Bahsoun, L. Feraud, and C. Betourne. A "two degrees of freedom" approach for parallel programming. In *Proceedings of the 1990 IEEE International Conference on Programming Languages*, pages 261–270, 1990.

[8] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of OOPSLA'90*, volume 25, pages 303–311. SIGPLAN Notices, ACM Press, October 1990.

[9] Jean-Piere Briot and Akinori Yonezawa. Inheritance and synchronization in concurrent OOP. In *Proceedings of ECOOP'87*, volume 276 of *Lecture Notes in Computer Science*, pages 33–40. Springer-Verlag, 1987.

[10] P. A. Buhr, Glen Ditchfield, and C. R. Zarnke. Adding concurrency to a statically type-safe object-oriented programming language. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 18–21. SIGPLAN Notices, ACM Press, April 1989.

[11] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In *Lecture Notes in Computer Science*, volume 16, pages 89–102. Springer-Verlag, 1974.

[12] Denis Caromel. A general model for concurrent and distributed object-oriented programming. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 102–104. SIGPLAN Notices, ACM Press, April 1989.

[13] Antonio Corradi and Letizia Leonardi. Parallelism in object-oriented programming languages. In *Proceedings of the 1990 IEEE International Conference on Programming Languages*, pages 271–280, 1990.

[14] D. Decouchant et. al. A synchronization mechanism for typed objects in a distributed system. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 105–107. SIGPLAN Notices, ACM Press, April 1989.

[15] Narain Gehani and William D. Roome. *The Concurrent C Programming Language*. Prentice Hall, 1989.

[16] John Hogg and Steven Weiser. OTM: applying objects to tasks. In *Proceedings of OOPSLA'87*, volume 22, pages 388–393. SIGPLAN Notices, ACM Press, October 1987.

[17] Dennis G. Kafura and Keung Hae Lee. Inheritance in Actor based concurrent object-oriented languages. In *Proceedings of ECOOP'89*, pages 131–145. Cambridge University Press, 1989.

[18] Henry Lieberman. Concurrent object-oriented programming in Act/1. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. The MIT Press, 1987.

[19] Steven E. Lucco. Parallel programming in a virtual object space. In *Proceedings of OOPSLA'87*, volume 22, pages 26–34. SIGPLAN Notices, ACM Press, October 1987.

[20] Svend Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings of ECOOP'92*, 1991. (to appear).

[21] Hidehiko Masuhara, Satoshi Matsuoka, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Submitted to OOPSLA'92*, 1992.

[22] Satoshi Matsuoka and Satoru Kawai. Using Tuple Space communication in distributed object-oriented languages. In *Proceedings of OOPSLA'88*, volume 23, pages 276–283. SIGPLAN Notices, ACM Press, September 1988.

[23] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. Synchronization constraints with inheritance: What is not possible — so what is? Technical Report 10, Department of Information Science, the University of Tokyo, 1990.

[24] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of ECOOP'91*, number 512 in Lecture Notes in Computer Science, pages 231–250. Springer-Verlag, 1991.

[25] Satoshi Matsuoka and Akinori Yonezawa. Metalevel solution to inheritance anomaly in concurrent object-oriented languages. In *Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[26] José Meseguer. (personal communication).

[27] José Meseguer. A logical theory of concurrent objects. In *Proceedings of OOPSLA'90*, volume 25, pages 101–115. SIGPLAN Notices, ACM Press, October 1990.

[28] Robin Milner. *Communication and Concurrency*. Prentice Hall, Engle Cliffs, 1989.

[29] Jürgen Nehmer, Dieter Haban, Friedemann Mattern, Dieter Wybraniez, and H. Dieter Rombach. Key concepts of the INCAS multicomputer project. *IEEE Transactions on Software Engineering*, 13(8):913–923, August 1987.

[30] Christian Neusius. Synchronizing actions. In *Proceedings of ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 118–132. Springer-Verlag, 1991.

[31] Oscar Nierstrasz and Michael Papathomas. Towards a type theory of active objects. In *Proceedings of the 1990 ECOOP-OOPSLA Workshop on Object-Based Concurrent Programming, Ottawa, Canada, Oct. 1990*, volume 2 of *OOPS Messenger*, pages 89–93. ACM Press, April 1991.

[32] M. Papathomas. Concurrency issues in object-oriented programming languages. In D. Tsichritzis, editor, *Object Oriented Development*, chapter 12, pages 207–245. Université de Geneve, 1989.

[33] Etsuya Shibayama. Reuse of concurrent object descriptions. In B. Meyer, J. Potter, M. Tokoro, and J. Bezivin, editors, *Proceedings of TOOLS 3, Sydney*, pages 254–266, November 1990.

[34] Alan Snyder. Encapsulation and inheritance. In *Proceedings of OOPSLA'86*, volume 21, pages 38–45. SIGPLAN Notices, ACM Press, September 1986.

[35] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with Enabled-Sets. In *Proceedings of OOPSLA'89*, volume 24, pages 103–112. SIGPLAN Notices, ACM Press, October 1989.

[36] Anand Tripathi, Eric Berge, and Mehmet Aksit. An implementation of object-oriented concurrent programming language SINA. *Software—Practice and Experience*, 19(3):235–256, March 1989.

[37] Jan van den Bos and Chris Laffra. PROCOL: a parallel object language with protocols. In *Proceedings of OOPSLA'89*, volume 24, pages 95–102. SIGPLAN Notices, ACM Press, October 1989.

[38] Ken Wakita and Akinori Yonezawa. Linguistic supports for development of organizational information systems. In *Proceedings of ACM COCS*, November 1991.

[39] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA'88*, volume 23, pages 306–315. SIGPLAN Notices, ACM Press, September 1988.

[40] Masahiro Yasugi, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/onEM-4: A new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *Proceedings of 6th ACM International Conference on Supercomputing*. ACM, 1992. (to appear).

[41] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. Computer Systems Series. The MIT Press, 1990.

[42] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings of OOPSLA'86*, volume 21, pages 258–268. SIGPLAN Notices, ACM Press, September 1986.

[43] Akinori Yonezawa and Takuo Watanabe. An introduction to object-based reflective concurrent computations. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, volume 24, pages 50–54. SIGPLAN Notices, ACM Press, April 1989.

A Appendix — Analysis of the Synchronizing Actions

Synchronizing Actions[30] was recently proposed which extends behavior abstractions. It also supports method guards in the form of preconditions. Figure 13 is the definition of a bounded buffer in Synchronizing Actions. The four keywords in the method definition are as follows: `matching`, `action`, `pre`, and `post` specify the guard, the pre-actions, the method body, and the post-actions, and respectively. Synchronizing Actions supports intra-object concurrency, and behavior abstractions is used to exclude mutually interfering operations.

Since Synchronizing Actions utilize behavior abstractions, one could conjecture that the anomalies presented in Section 3.2 could occur. It is a little bit difficult, because most of the partitioning is absorbed in the guards; it is however, possible to create a mutual exclusion condition that cannot be reflected to the guards, thus requring re-definitions. As an example, we define a class `extended-bounded-buffer2`, which extends the `bounded-buffer` class with a method `read-middle`, which returns the middle elements of the buffer excluding the head and tail. Thus, `read-middle` should not be invoked when the buffer consists of less than three elements. Furthermore, suppose that the implementation details require that there exist five or more elements for `read-middle` to be mutually independent from both `put` and `get` (this alternative partitioning cannot be reflected to the guards). Figure 13 is the resulting subclass definition of `extended-bounded-buffer2`. Here, notice that not only the `concurrency-control` part have to be extended, but also the methods themselves must also be re-defined (the precondition part). The required re-definition occurs for the same reason as the original behavior abstractions — the partitioning of states.

```
class bounded-buffer;
private:
   const SIZE = 64;
   int in = 0, out = 0, buf[SIZE];
concurrency-control:
   int N = 0;          //counts queued elements
   behavior-abstraction
      op-on-head = { get }
      op-on-tail = { put }
public:
   method put(int elem);
      matching   ( N < SIZE );
      pre        { exclude op-on-tail; }
      action     { in++; /* add element to tail of buf */ }
      post       { N++; }
   method int get();
      matching   ( N > 0 );
      pre        { exclude op-on-head; }
      action     { /* return element from head of buf */ out++; }
      post       { N--; }
end bounded-buffer;

class extended-bounded-buffer2 inherits bounded-buffer;
concurrency-control:
   behavior-abstraction // new exclusion sets for read-middle
      op-on-head-and-tail-and-middle = { get, put, read-middle-elements }
      op-on-head-and-middle = { get, read-middle-elements }
      op-on-tail-and-middle = { put, read-middle-elements }
public:
   method int[] read-middle-elements();
      matching   ( N >= 3 );
      pre        { exclude op-on-head-and-tail-and-middle; }
      action     { /* return the middle elements of buf excluding two
                elements from both head and tail */ }
      post       { N = 2; }
   // re-definitions of both put and get methods in bounded-buffer
   method put(int elem);
      matching   ( N < SIZE );
      pre        { if (N >= 5) exclude op-on-tail
                   else exclude op-on-tail-and-middle; }
      action     { in++; /* add element to tail of buf */ }
      post       { N++; }
   method int get();
      matching   ( N > 0 );
      pre        { if (N >= 5) exclude op-on-head
                   else exclude op-on-head-and-middle; }
      action     { /* return element from head of buf */ out++; }
      post       { N--; }
end bounded-buffer;
```

Figure 13: Inheritance Anomaly in Synchronizing Actions