

A SOLUTION TO INHERITANCE ANOMALY BASED ON REUSABLE BEHAVIOR DEFINITIONS

ANDREW EDWARD SANTOSA

YASURO KAWATA

MAMORU MAEKAWA

Graduate School of Information Systems
University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, Tokyo 182-8585
Japan

ABSTRACT

In this paper we present a solution to inheritance anomaly. Our solution is based on the principle of separation of concerns; i.e., in our solution, concurrent behavior is defined and manipulated as a separate entity from classes, unlike conventional concurrent object-oriented languages.

Our approach has advantages over existing solutions. First, our approach frees programmers from the burden of having always to think in pairs the bodies of methods and their invocation constraints. Secondly, since concurrent behavior once defined can also be reused, it can also reduce the cost of building concurrent applications significantly. Thirdly, rather than directly program a complex concurrent behavior into a class, several simpler concurrent behavior can be integrated into a class. This makes program codes easier to reason about.

Our solution even takes care of the case of so-called history-only sensitiveness, which is usually the hardest-to-solve case of inheritance anomaly.

Keywords: inheritance anomaly, concurrent object-oriented programming

INTRODUCTION

Inheritance anomaly [4] is often encountered in programming programming with concurrent object-oriented programming languages (COOPLs). It is the nontrivial redefinitions of parent class code in its child class that are caused by the difference in *concurrent behavior* between the parent and child class. Here we define concurrent behavior as the message accepting policy of an object¹.

In a class definition of conventional COOPLs, there are two kinds of code:

- the *concurrency code* which express the concurrent

¹Concurrent behavior is often referred to as *synchronization* in most references. However, since the intended meaning is often larger (e.g. includes mutual exclusion) than those found in operating system textbooks, we chose the term concurrent behavior instead.

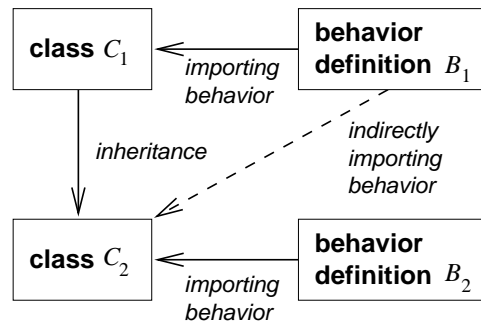


FIG. 1: THE OUTLINE OF OUR SOLUTION

behavior, and

- the *functional code* which express how to accomplish the service requested by a client.

Inheritance anomaly occurs when the concurrency code cannot be effectively inherited without non-trivial class redefinitions.

In this paper we will show a solution to inheritance anomaly which is based on separation of both kinds of code. The concurrency code is defined in a *behavior definition*, while the functional code is defined in a class. A programmer codes the relation of both in the class side to obtain the desired functional and concurrent behavior. This is shown in Figure 1. A class C_1 imports concurrent behavior from behavior definition B_1 to specify the concurrent behavior of its objects. In this importing of behavior, the programmer associates C_1 's methods with *abstract methods* (to be explained later) defined in behavior definition B_1 .

A child class C_2 is derived from C_1 . Assume that the required concurrent behavior for C_2 differs from those of the C_1 . To code this new concurrent behavior of C_2 , the concurrent behavior specified in a behavior definition B_2 is imported into C_2 . Since we only need to associate methods of C_2 with abstract methods of B_2 , there would be no redefinitions of parent class code. We can thus avoid inheritance anomaly in this case.

```

class BBuf: Concurrency {
protected:
    buf b; // Sequential bounded buffer.
public:
    void put(int x) when (!b.full());
    int get() when (!b.empty());
behavior:
    behavior bbufmutex is Mutex;
    put, get behave_as bbufmutex.excl;
};

```

FIG. 2: BBUF

Note that a child class inherits the concurrent behavior imported by its ancestors. As in Figure 1 with C_2 indirectly importing behavior from behavior definition B_1 , whose concurrent behavior imported by C_1 .

By separating functional and concurrency code, we will benefit from separation of concerns; a programmer can concentrate first on coding the functional part, and later integrating concurrency to it by reusing already defined behavior definitions or defining new ones.

Separation of functional and concurrency code also increases reusability because it avoids replication of code [1]. We would not need to repeatedly program the same concurrent behavior for all classes that require it.

By dividing the code that describes concurrent behavior of a class into several behavior definitions, we can also program complex concurrent behavior that are easier to reason about.

Our approach depends largely on whether we are able to localize the concurrent behavior that cause inheritance anomaly into a behavior definition. To effectively program behavior definitions, we introduce a new *events history* mechanism. This mechanism has a very high expressiveness, which is good enough to solve the most difficult-to-solve case inheritance anomaly known as *history-only sensitiveness of acceptable states* [4].

BOUNDED BUFFER EXAMPLE

We will base the examples on a COOPL where an object may have internal concurrency; i.e. multiple threads can be executed simultaneously on an object. In our language a thread is assigned to a method execution.

We will use C++-like syntax unless otherwise noted. However, we are assuming a dynamic binding semantics, which is different from C++'s default. In C++, variables are statically bound to their defining class.

```

behavior Mutex {
    excl when (!isactive(excl));
};

```

FIG. 3: MUTEX

The traditional example of inheritance anomaly is based on a bounded buffer problem [4]. A bounded buffer class BBuf is shown in Figure 2. Inheriting from a class Concurrency signifies the concurrent nature of BBuf's instances. Note that we implemented BBuf as concurrent wrapper of a sequential bounded buffer b for simplicity.

BBuf has two methods *put* and *get*. *Put* puts an element to the buffer while *get* retrieves an element from it. Both methods have their guards, that are written in a **when** clause. Guards are boolean expressions. They may contain references to instance variables and the corresponding method's parameters. A method can begin execution only when its guard evaluates to true. A method without guard can be considered as having guard that always yields true. We programmed the guards such that *put* cannot be executed when the buffer is full while *get* cannot be executed when the buffer is empty.

A special **behavior:** section is for integrating concurrent behavior defined in behavior definitions into a class. The

```

behavior bbufmutex is Mutex;

```

statement imports a concurrent behavior from behavior definition Mutex (shown in Figure 3) and give it a local name *bbufmutex*. Mutex is a behavior definition that provides mutual exclusion to its importers. We need mutual exclusion because we do not want *put* or *get* to be executed concurrently with other *puts* or *gets*². The

```

    put, get behave_as bbufmutex.excl;

```

statement declares that *put* and *get* are associated *abstract method excl* defined in Mutex. Associating methods to an abstract method adds further constraint to the methods. The methods now can only be executed when their own guards and the abstract method's *activation condition* are satisfied.

The activation condition of an abstract method is defined in a **when** clause. Activation conditions, similar to guards, are boolean expressions that are used to write concurrency constraints, but they may only call *history functions*. History functions are parts of our events history mechanism. They read an events history table, which records all events (message arrival, method activation, and method finalization) that has occurred on an object together with their order of occurrence, then transforms the data into a more useful

²A rather strong constraint because *put* can actually be executed concurrently with *get*.

```

behavior Blocking {
    blocker;
    unblocker;
    blockable when (
        lastfin(blocker) < lastfin(unblocker) &&
        lastact(blocker) < lastfin(blocker));
};

```

FIG. 4: BLOCKING

form that is needed by the programmer. *Excl*'s activation condition *!isactive(excl)* calls a history function *isactive*, that returns true when the abstract method supplied as its parameter is currently executing. The activation condition thus evaluates to true only when *excl* itself is not currently executing.

Suppose we want to define BBuf's child class called GBBuf. In GBBuf, a new method *gget* is defined. *Gget* has the same function with *get*, but it is required that *gget* can be executed only if the last completed execution is not *put*. Providing only guards as a mean of coding concurrency constraints could cause the inheritance anomaly that is categorized as *history-only sensitiveness of acceptable states* [4]. Programmers are forced to provide a new state variable whose value is set inside methods. Methods that are otherwise inherited from parent class are thus need to be redefined, and inheritance anomaly occurs.

To solve the problem, we first consider the concurrent behavior of GBBuf. There is a method (*put*) that when executed, blocks the further execution of another method (*gget*). The second method can be executed again if the third method (*get*) has been executed. We abstract this concurrent behavior in the behavior definition Blocking shown in Figure 4. In Blocking, there are three abstract methods *blocker*, *unblocker*, and *blockable*. Each abstracts the behavior of the first to the third method, respectively. figure *Blockable*'s activation condition calls the *lastfin* and *lastact* history functions. *Lastfin* returns the occurrence number of the last finalization of an abstract method, while *lastact* returns the occurrence number of the last finalization of an abstract method. *Gget*'s activation condition makes sure that *unblocker* was finished later than *blocker*, and *blocker* is not currently executing. Note that an abstract method that has not yet been executed is assumed to be last executed at an indefinite past.

After we have defined Blocking, we will now define the child class GBBuf of BBuf and import the behavior from Blocking. The child class GBBuf is shown in Figure 5. Note that *bbufmutex*, which was defined in BBuf can be referenced from GBBuf, and thus we may incrementally modify the concurrency code. In GBBuf, we associate *gget* with *excl*. In Figure 5, no

```

class GBBuf: public BBuf {
public:
    int gget() when (!b.empty());
behavior:
    gget behave_as bbufmutex.excl;
    behavior block is Blocking;
    put behave_as block.blocker;
    get behave_as block.unblocker;
    gget behave_as block.blockable;
};

```

FIG. 5: GBBUF

```

behavior ReadersWriters {
    reader when (!isactive(writer));
    writer when (!isactive(reader) &&
        !isactive(writer));
};

```

FIG. 6: READERSWRITERS

redefinition of parent class code is required, and thus we have solved the inheritance anomaly.

In this section we have also shown how we may separate functional code from concurrency code. This frees the programmer from the difficulties in programming them at once.

DISK INTERFACE EXAMPLE

In this section we will show how we can program more complex concurrent behavior.

Consider a class CachedDisk, which provides interface for cached disk I/O. Because the cache is located in volatile memory, we would still want to provide interfaces for direct I/O, in case safety is at a higher priority than speed. CachedDisk will thus have the following methods:

- *read* which reads directly from disk,
- *write* which writes directly to disk,
- *cread* which reads from disk through cache,
- *cwrite* which writes to disk through cache,
- *updatedisk* which flushes the contents of the cache to disk, and,
- *updatecache* which updates the contents of the cache when changes directly applied to disk is newer.

Suppose that we already have ReadersWriters (Figure 6) and Blocking behavior definitions in our library, then we may define CachedDisk as in Figure 7. Since all methods that access the disk can be executed only one at a time, we use *diskmutex* which imports Mutex to specify the constraint between such methods. Since the methods that read from the cache can be executed concurrently while those that modify the cache

```

class CachedDisk: Concurrency {
public:
    int read(int addr);
    void write(int data, int addr);
    int cread(int addr);
    void cwrite(int data, int addr);
    void updatedisk();
    void updatecache();
behavior:
    behavior diskrw is ReadersWriters;
        cread, updatedisk behave_as diskmutex.read;
        cwrite, updatecache behave_as diskmutex.write;
    behavior diskmutex is Mutex;
        write, read, updatedisk, updatecache
        behave_as diskmutex.excl;
    behavior block1 is Blocking;
        cwrite behave_as block1.blocker;
        updatedisk behave_as block1.unblocker;
        read, write behave_as block1.blockable;
    behavior block2 is Blocking;
        write behave_as block2.blocker;
        updatecache behave_as block2.unblocker;
        cread, cwrite behave_as block2.blockable;
};

```

FIG. 7: CACHEDDISK

cannot, we use *diskrw* which imports ReadersWriters to specify the constraint between them. If the client writes to cache memory, then the cache must be flushed first before direct read/write can be performed. This is the behavior expressed by *block1*. *Block2* specifies that if the client writes to disk through direct write, then the cache must be updated first before cached read/write can be processed.

As shown by *CachedDisk*, we will benefit from reusability of behavior definitions. *CachedDisk* also showed that we may specify complex concurrent behavior by integrating concurrent behavior of simple behavior definitions. This results in a code that is easier to reason about.

COMPARISONS

Our behavior definition has much resemblance to DRAGOON's *behavioral class* [1]. DRAGOON employs the calling of history functions in behavioral class' activation conditions. These functions are based on *synchronization counters*. They return the number of message arrivals, activations, and finalizations of methods. Unfortunately, DRAGOON's history functions are not expressive enough to solve history-only sensitiveness of acceptable states.

Solutions to inheritance anomaly by Mitchell and Wellings [5], Holmes [3], and Cugola [2] require attention. In such proposals, concurrent behavior is coded

using method guard-like constructs. Redefinitions in a child class are avoided by separately referencing to parent class method's body and the corresponding guard with the keyword **super** from the child class' method code. We think that such solution only eliminate the need to rewrite parent class code in a child class and not actually avoiding redefinitions. Our proposal provides a higher level of abstraction and therefore easier to use.

CONCLUSION

There has been many solutions proposed to solve inheritance anomaly. However, none of the solutions having enough flexibility and ease of use to be actually implemented. In this paper we had shown a solution to inheritance anomaly that has a high level of abstraction and flexibility. We believe that our solution can be applied to most COOPLs that support guarded methods.

Because the core problem of inheritance anomaly is the inability of language designers to foresee all the programmers' needs, a total solution to inheritance anomaly would be very hard to find, if not impossible. A potential limitation of our approach is the inability to mix history functions with state variables and method parameters in a single guard expression. This could reduce flexibility. However, we think that such requirements are relatively rare, and chose not to treat them in this paper.

REFERENCES

- [1] Colin Atkinson. *Object-Oriented Reuse, Concurrency and Distribution: An Ada-Based Approach*. Addison-Wesley, 1991.
- [2] Gianpaolo Cugola and Carlo Ghezzi. CJava: Introducing concurrent objects in Java. In *Proceedings of the 4th International Conference on Object-Oriented Information Systems (OOIS '97)*, 1997. Brisbane (Australia), 10–12 November 1997.
- [3] David Holmes. Java™: Concurrency, synchronization and inheritance. 1998. Available from <http://www.mri.mq.edu.au/~dholmes/java-concurrency.html>.
- [4] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 4, pages 107–150. MIT Press, 1993.
- [5] S. E. Mitchell and A. J. Wellings. Synchronisation, concurrent object-oriented programming and the inheritance anomaly. *Computer Languages*, 22(1):15–26, April 1996.