The inheritance anomaly: ten years after

Giuseppe Milicia
Chi Spaces Technologies ltd.
Cambridge, UK
milicia@chispaces.com

Vladimiro Sassone
University of Sussex, UK
vs@susx.ac.uk

ABSTRACT

The term *inheritance anomaly* was coined in 1993 by Matsuoka and Yonezawa [15] to refer to the problems arising by the coexistence of *inheritance* and *concurrency* in *concurrent object oriented languages* (COOLs). The quirks arising by such combination have been observed since the early eighties, when the first experimental COOLs were designed [3]. In the nineties COOLs turned from research topic to widely used tools in the everyday programming practice, see e.g. the Java [9] experience. This expository paper extends the survey presented in [15] to account for new and widely used COOLs, most notably Java and C♯ [19]. Specifically, we illustrate some innovative approaches to COOL design relying on the *aspect oriented programming* paradigm [13] that aim at better, more powerful abstraction for concurrent OOP, and provide means to fight the inheritance anomaly.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures, Inheritance*; D.1.3 [**Programming techniques**]: Concurrent Programming

Keywords

Inheritance Anomaly, Concurrent Programming

1. INTRODUCTION

In the early eighties the first attempts to mix *object oriented programming* and *concurrency* showed that the two concepts do not mix gracefully [3, 6]. In a concurrent program, the set of messages objects can handle is not uniform over time. Instead, it depends on the actual *state* of the object. To enforce such '*synchronization constraints*' we are forced to write specific *synchronization code*.

To exemplify the situation let us consider the following pseudo-code implementing a concurrent bounded buffer:

```
class Buffer {
  ...
  void put(Object el) { if ("buffer not full") ... }
  Object get() { if ("buffer not empty") ... }
}
```


SAC'04, March 14–17, 2004, Nicosia, Cyprus

Clearly we must make sure that no object is removed from an empty buffer and that no object is inserted into a full buffer. In a sequential setting, the burden of ensuring such constraints resides with the buffer's user. Indeed the buffer is created and used by *one* thread only, which is responsible for the state of the object. To facilitate usage, the buffer's methods might return certain error codes in case of misuse. This approach is not feasible in a *concurrent* setting. The buffer will be used concurrently by multiple clients, leaving each of them no idea on the buffer's current state. Although it is possible to envisage ad-hoc protocols among clients to keep track of the buffer's state, such a solution is complex and hardly feasible in practice. The burden of enforcing the *synchronization constraints* must ultimately lie with the buffer itself. Unfortunately, mixing *behavioural* and *synchronization* code in class definitions represents an obstacle to code inheritance. The quirks arising from the coexistence of inheritance and concurrency were considered so severe as to suggest the removal of inheritance from concurrent object oriented languages (COOLs) [3]. In 1993 Matsuoka and Yonezawa coined the now widely known term *inheritance anomaly* to refer to the problem.

Since then, we have seen a significant change in the way COOLs have been used. In the early nineties, COOLs were a matter of research rather than programming practice. Concurrency was a very specific requirement for a small niche of the programming community. This is not longer the case. The trend changed with the introduction of Java [9] which is to be considered the first COOL to be widely accepted by the mainstream programming community. Nowadays concurrency is essential to application development, consequently it is treated as a fundamental part of language design. Newly introduced languages, meant for everyday programming, do provide concurrency, as for instance C♯ [19].

It is often unclear how the inheritance anomaly affects concurrent programming in modern COOLs. In this paper we extend the survey presented in [15] to include popular languages such as Java, C♯ and Eiffel. We move on describing some of the most interesting approaches to the problem which have emerged in the last ten years. We concentrate on programming languages based on the *aspect oriented* philosophy, a new and promising programming paradigm [16].

2. THE INHERITANCE ANOMALY

As briefly mentioned in the Introduction, interweaving the behavioural and synchronization code of class definitions is an obstacle to inheritance. We refer to this problem as the *inheritance anomaly*.

In this section we shall use the classic bounded buffer example to show where and how the anomaly occurs. Consider the following pseudo-code:

```
class Buffer {
  ...
  void put(Object el) { ... }

  Object get() { ... }
}
```

The bounded buffer provides the methods `get` and `put` respectively to remove and insert an element. In a concurrent setting we need to refine the code above with suitable *synchronization code*, so as to make sure that no `get` is executed on an empty buffer and, dually, that no `put` is executed on a full buffer.

It is generally agreed that the inheritance anomaly shows itself in different guises depending on the *synchronization mechanism* provided by the language. In [15] the authors give a taxonomy of the problem and classify the difference occurrences of the inheritance anomaly in three broad classes as follows.

History-sensitiveness of acceptable states. Synchronization constraints can be enforced by providing a *guard* for each method. A guard is a boolean expression which must be true for the method to be executable:

```
class Buffer {
  ...
  void put(Object el) when "not full" { ... }

  Object get()  when "not empty" { ... }
}
```

The semantics is self-explanatory. Before executing a method its guard is evaluated, if the latter evaluates to false the calling thread must '*wait*' for the condition to become true. Such form of synchronization has been very popular, and it survives to this day (under several superficially different forms) in widely used languages such as Java [9] and C♯ [19]. The *wait* primitive is not immune to the inheritance anomaly. When method enabling rather than depending only on the object's state, depends on its past history, the inheritance anomaly occurs. Suppose for instance that we want to refine our buffer with a method `gget` that works like `get` but that cannot be executed immediately after a `get`:

```
class HistoryBuffer extends Buffer {
  ...
  Object gget() when "not empty" and "not after-get" {
     ...
  }
}
```

Clearly, this can only be achieved adding code to `get` to keep track of its invocations. That is, we have to rewrite the entire class. We will revisit this problem later on.

Partitioning of states. Inspired by the example above, one may disentangle code and synchronization conditions by describing the enabling of methods according to a partition of the object's states. To describe the behavior of class `Buffer`, for instance, the state can be partitioned in three sets: `empty`, `partial`, and `full`, the former containing the states in which the buffer is empty – so that `get` is inhibited – the latter those in which it is full – so that is `put` to be disallowed. One can then specify

```
put: requires not full
get: requires not empty
```

and refine the code of `get` and `put` to specify the state transitions. For instance, `get` would declare the conditions under which the buffer becomes `empty` or `partial`:

```
Object get() {
  ...
  if ("buffer is now empty") become empty;
  else become partial;
}
```

The inheritance anomaly surfaces again here, as derived classes may force a refinement of the state partition. As an example, consider adding a method `get2` that retrieves two elements at once. Alongside `empty` and `full`, it is necessary to distinguish those states where the buffer contains exactly one element. Clearly, the state transitions specified in `get` and `put` must be re-described accordingly.

Modification of acceptable states. A third kind of anomaly happens with mix-in classes, that is classes whose purpose is to be mixed-into other classes to add to their behavior. The typical situation arises when one wishes to enrich a class with a method that influences the acceptance states of the original class' methods. For instance, from an 'object-oriented' perspective, it is perfectly legitimate to expect to be able to design a

```
class Lock {
  ...
  void lock()   { ... }
  void unlock() { ... }
}
```

to be used to add lock capabilities to clients classes purely by means of the standard inheritance mechanism. But, clearly enough, (multiple) inheritance of `Lock` and `Buffer` does nothing towards creating a lockable buffer, unless we completely recode `get` and `put` to keep into account the state of the `Lock` component of the object.

3. THE INHERITANCE ANOMALY IN MODERN COOLS

In this section we shall analyze the problem of the inheritance anomaly in two widely used COOLs: Java and C♯. The two languages are naturally concurrent and used as such in the everyday programming practice.

3.1 Java

Java [9] can be considered the first object oriented language with native support for concurrency that was widely accepted by the programming community. Concurrency, at first considered a specialized need for a restricted audience, has become with Java a central feature of new programming languages.

Java is an imperative object oriented language. Its syntax is similar to that of C++ [24]. Java programs can be *multi-threaded*, where the concurrent access to shared objects by threads is regulated by a variation of the *monitor* primitive [10]. Every object possesses a *lock*. The lock is *indirectly* accessed when accessing *synchronized* methods or using the `wait` and `notify/notifyAll` primitives.

The classic bounded buffer example can be programmed in Java as in Figure 1. The code is self-explanatory; it is however useful to comment on the use of the `wait` primitive. A while loop ensures that the method will wait till the guard becomes false. At the end of the methods, having modified the state of the object, we signal all the waiting threads using the primitive `notifyAll`.

Java monitors are clearly a variation of the method guard synchronization primitive, and as such they suffer from the *history-dependent* variety of the anomaly. Figure 2 illustrates this. We are forced to do some bookkeeping to verify whether or not the last method to be executed was a `get` or not. We implement it using a boolean flag, `afterGet`, which forces us to redefine the methods

```
public class Buffer {

  protected Object[] buf;
  protected int MAX;
  protected int current = 0;

  Buffer(int max) {
      MAX = max;
      buf = new Object[MAX];
  }
  public synchronized Object get()
    throws Exception {
      while (current<=0) { wait(); }
      current--;
      Object ret = buf[current];
      notifyAll();
      return ret;
  }
  public synchronized void put(Object v)
    throws Exception {
      while (current>=MAX) { wait(); }
      buf[current] = v;
      current++;
      notifyAll();
  }
}
```

Figure 1: Buffer in Java

```
public class HistoryBuffer extends Buffer {
  boolean afterGet = false;

  public HistoryBuffer(int max) { super(max); }

  public synchronized Object gget()
    throws Exception {
      while ((current<=0)||(afterGet)) {
          wait();
      }
      afterGet = false;
      return super.get();
  }
  public synchronized Object get()
    throws Exception {
      Object o = super.get();
      afterGet = true;
      return o;
  }
  public synchronized void put(Object v)
    throws Exception {
      super.put(v);
      afterGet = false;
  }
}
```

Figure 2: HistoryBuffer in Java

`get` and `put` which we would rather inherit. A clear-cut example of inheritance anomaly.

3.2 C♯

C♯ [19] is a recently introduced COOL. It borrows considerably from the Java experience, which however it improves in a number of ways. From a concurrent programming point of view, C♯ relies on the notion of *monitor* just like Java. Its monitor interface is, however, slightly more flexible. In contrast to Java, monitors are programmed in C♯. The `Monitor` class provides the methods `Enter()` and `Exit()` to delimit code which must be executed in mutual exclusion. The C♯ instruction `lock()` can be used for the same purpose and is just syntactic sugar over the monitor mechanism. If a whole method is to be executed in mutual exclusion the *attribute* `MethodImplOptions.Synchronized` can be used instead of the explicit monitor. Attributes allows programmers to specify meta-data on the program which can be interpreted and treated appropriately by the compiler. The C♯ synchronization mechanism is based on the monitor primitives `Wait()`, `Pulse()` and `PulseAll()`, which are essentially equivalent to Java `wait()`, `notify()` and `notifyAll()`.

The additional flexibility given by explicit monitors does not help against the history-sensitive variety of the inheritance anomaly. C♯ is indeed subject to the same anomalies as in Java. In Figure 3 we show a possible implementation of the `Buffer` in C♯. Note that we used the `MethodImplOptions.Synchronized` attribute for the method `get` and the explicit monitor for `put`. The two techniques are equivalent and we show both for completeness only. Figure 4 shows that implementing the `HistoryBuffer` class in C♯ gives rise to an instance of the inheritance anomaly just like it was the case for Java. The method redefinition abstraction provided by the two languages does not provide the means to finely adjust the method behavior.

3.2.1 *Polyphonic C♯*

Polyphonic C♯ is a dialect of C♯ developed at Microsoft Research, Cambridge [4]. In Polyphonic C♯ , the monitor synchronization mechanism of C♯ is replaced by a *pattern-based* mechanism inspired by the Join calculus [8]. Precisely, Polyphonic C♯ is based on two main notions: *asynchronous methods* and *chords*. Asynchronous methods, as opposed to *synchronous* ones, do not return a result, and their caller thread does not wait blocked for method completion, but can proceed immediately. In standard C♯ method calls are always synchronous: they return a value and the caller blocks until method completion. Chords are code blocks associated with a set of synchronous and asynchronous method identifiers. Every chord can contain at most a synchronous method. The body of a chord is executed only if and when *all* of its method identifiers have been called. If more than one chord can be executed at any time, one is chosen randomly. To exemplify the notion of chord, consider the following (simplistic) implementation of a buffer:

```
public class Buffer {
  public String get() & public async put(String s) {
     return s;
  }
}
```

The method `put` is asynchronous, consequently it does not block the callers. The method `get` is synchronous, callers will be blocked until the chord can be executed, i.e. until there is an element in the buffer.

From the existing literature on Polyphonic C♯ and the Join Calculus, it appears that the language is subject to the state partitioning variety of the anomaly. Indeed its synchronization mechanism is closely related to *behavior abstractions* [12]. An analysis of Polyphonic C♯ in this direction has been carried out in [20].

3.3 Eiffel

Eiffel [17] is a popular OO language designed by Meyer and Nerson. Currently, concurrency is not part of the Eiffel language. Multithreading finds its way inside the language through additional threading libraries, just as it happens for C and C++. The 'official' language specification, however, describes a concurrency model: SCOOP (Simple Concurrent Object-Oriented Programming) [18].

```
public class Buffer {
   protected Object[] buf;
   protected int max;
   protected int current = 0;

   public Buffer(int max) {
      this.max = max;
      buf = new Object[max];
   }

   public virtual void put(Object v) {
      Monitor.Enter(this);
      while (current>=max) { Monitor.Wait(this); }
      buf[current] = v;
      current++;
      Monitor.PulseAll(this);
      Monitor.Exit(this);
   }

   [MethodImpl(MethodImplOptions.Synchronized)]
   public virtual Object get() {
      while (current<0) { Monitor.Wait(this); }
      current--;
      Object ret = buf[current];
      Monitor.PulseAll(this);
      return ret;
   }
}
```

Figure 3: Buffer in C♯

```
public class HistoryBuffer : Buffer {
  bool afterGet = false;

  public HistoryBuffer(int max) : base(max) {}

  [MethodImpl(MethodImplOptions.Synchronized)]
  public Object gget() {
     while ((current<=0)||(afterGet)) {
        Monitor.Wait(this);
     }
     afterGet = false;
     return base.get();
  }

  [MethodImpl(MethodImplOptions.Synchronized)]
  public override Object get() {
     Object o = base.get();
     afterGet = true;
     return o;
  }

  [MethodImpl(MethodImplOptions.Synchronized)]
  public override void put(Object v) {
     base.put(v);
     afterGet = false;
  }
}
```

Figure 4: HistoryBuffer in C♯

```
class BBUF[G]  creation make
 feature ...
  put(x : G)  is
      require not full
      do ...  ensure not empty  end
  get : G  is
      require not empty
      do ...  ensure not full  end
end
```

Figure 5: (Sequential) Bbuf in Eiffel

SCOOP is currently available only through a proof-of-concept implementation [7], yet it is likely to be the future of concurrency in Eiffel.

Syntactically SCOOP's impact is minimal, only one keyword needs to be introduced: `separate`. Semantically the changes are more profound and interesting. The `separate` keyword can be applied to classes, e.g. `separate class NAME`, and to object instances, e.g. `o:separate TYPE`. Intuitively the keyword means that operations on the entity it refers to might be executed *concurrently* in their own process. A method call is said to be separate if it refers to a separate object. For instance, the call `x.f()` is separate if `x` is a separate object. In contrast with Java and C♯, separate calls in SCOOP can be either *asynchronous* or *synchronous*. In the first case the syntax is simply `x.f()`, in the second case we write `y:=x.f()`, called '*wait by necessity*' in SCOOP terminology. Every method of a separate object is executed in mutual exclusion, to avoid the difficulties arising from race conditions.

Eiffel is based on the *design by contract* paradigm. Methods are usually associated with a precondition, introduced by the keyword `require`, and a post-condition, denoted by `ensure`. Additionally, classes can be associated with an *invariant*, which they must respect at any time. Consider the Eiffel class implementing a (sequential) bounded buffer in Figure 5. The method `get` has a precondition ensuring a non-empty buffer. Similarly its post-condition guarantees that after its execution the buffer will not be full. Although deceivingly similar to method guards, assertions have a different semantics. Recall that we are still in a *sequential* world. Assertion violation does not result in a client waiting for the assertion to become true; rather, a run-time exception is thrown.

The assertion mechanism breaks down in a concurrent setting, as well known both in practical and theoretical communities. Meyer calls the resulting situation the *concurrent precondition paradox*. Indeed, the design-by-contract methodology is based on the notion that if a precondition is satisfied, the client calling a routine is guaranteed a result in line with the routine's post-condition. If we consider a concurrent execution of the bounded buffer above (add the `separate` keyword to the class definition), we see that this is clearly not the case. No matter how hard a client tries to satisfy the method's preconditions, another client can modify them *concurrently*. This problem leads to modified the precondition semantics for separate objects. Preconditions for separate objects, called *wait conditions*, are essentially equivalent to method guards. The semantic of a wait condition is:

> *'Before being executed a separate call must wait till it gains exclusive ownership of the separate object it refers to and till its wait-conditions are all satisfied.'*

Wait-conditions are prey to the history-dependent variety of the anomaly. Let us note that the bounded buffer in Figure 5 can be made concurrent simply by changing the first line of the class definition from: `class BBUF[G]` to: `separate class BBUF[G]` From this modified definition we can derive a class implementing an `HistoryBuffer` as shown in Figure 6. Again, we are forced to add a boolean flag, `after_get`, to check whether the last method to be executed was a `get` or not. This brings forth the need to redefine the inherited methods `get` and `put`. Despite a significantly different concurrency model, we face the same scenario seen for both Java and C♯. The core of the matter is that the inheritance anomaly depends *only* on the language's synchronization mechanisms, and wait conditions are intimately related to method guards.

```
separate  class HISTORY_BUFFER[G]
inherit BOUNDED_BUFFER[G]  redefine put, get
feature ...
  put(x : G)  is
      require not full
      do ... after_get := false
      ensure not empty
      end
   get : G  is
       require not empty
       do ... after_get := true
       ensure not full
       end
   gget : G  is
       require not empty
       do ... after_get := false
       ensure not full
       end
feature NONE
   after_get : BOOLEAN
end -- HISTORYBUFFER
```

Figure 6: HistoryBuffer in Eiffel

4. MODERN APPROACHES TO THE INHERITANCE ANOMALY

Since the survey presented in [15], new approaches to the problem of the inheritance anomaly have been pursued. Research led towards language features which are not only resilient to known forms of the anomaly but offer powerful abstractions to deal with concurrent programs. The four approaches we shall present have as a common basis the decoupling of synchronization code from the 'business code' of class definition. Such *separation of concerns* has been brought forth by a promising novel approach to program development: *aspect oriented programming* (AOP) [16].

4.1 Synchronization Patterns

The application of aspect oriented techniques to decouple synchronization from actual business code in class definitions was first explored by Lopes and Lieberherr in [14].

The authors do not propose a new language implementing their methodology; they rather describe a rich meta-language which can be mapped down to chosen *target languages*. The mapping may cause the loss of some of the benefits of the meta-language, depending on the nature target language. Languages based on *active objects* are ill-suited as target languages: the best support for the mapping are target languages which provide different abstractions for objects and processes.

The methodology of synchronization patterns helps to decouple business and synchronization code. Synchronization constraints are expressed in terms of synchronization patterns, by the following syntax:

```
sync_pattern name
   add_structure ...  // additional data structures
   add_func ...       // additional operations
   mutex ...          // Locks
   sync ...           // Synchronization scheme
```

Synchronization patterns contain four blocks which specify data structures (`add_structure`) and operations (`add_func`) additionally needed by the synchronisation scheme, the locks (`mutex`) and the synchronization strategy to be used (`sync`). The synchronization scheme is specified in terms of mutual exclusion, pre and post conditions. From the definition of the synchronization patterns and the 'normal' class definition, code for the target language is generated.

```
sync_pattern BufferSync
   add_structure          //empty
   add_func               //empty
   mutex per_object x1
   sync
      operation Object get()
        at Buffer exclusive x1
           requires (@ ! empty @)  false (wait)
      operation void put(Object o)
        at Buffer exclusive x1
           requires (@ ! full @)  false (wait)
```

Figure 7: Buffer using Synchronization Patterns

```
sync_pattern HistoryBufferSync : inherit BufferSync
 add_structure after-get : boolean
 sync
   operation Object get()
     at HistoryBuffer exclusive x1
        requires (@ ! empty @)  false (wait)
        on_exit (@ after-get=true @)
   operation void put(Object o)
     at HistoryBuffer exclusive x1
        requires (@ ! full @)  false (wait)
        on_exit (@ after-get=false @)
   operation Object gget()
     at HistoryBuffer exclusive x1
        requires (@ (!empty) && (!after-get) @)
            false (wait)
        on_exit (@ after-get=false @)
```

Figure 8: HistoryBuffer using Sync. Patterns

The synchronization scheme for the classic bounded buffer example can be specified using synchronization patterns as shown in the self-explanatory Figure 7. A guard is provided using the `requires` keyword, if the guard evaluates to false the calling thread will wait. The code between `(@` and `@)` is written in the target language (Java-like, in this case). Observe that the synchronization constraints are specified separately from the class behaviour, and in an ad-hoc language.

In Figure 8 we see the `HistoryBuffer`. The introduction of a history-sensitive method (`gget`) results in the need to redefine the synchronization constraints of the inherited methods. However, there is no need to redefine the *behaviour* of the methods. Code rewriting is limited to the synchronisation code, where it should belong, if anywhere. As we shall see, this *modularization* of code rewriting is typical of the aspect-oriented approach to the anomaly.

4.2 Composition Filters

Composition Filters [5] extend standard object-oriented modeling techniques by explicitly programmed *filters* which apply to method calls to and from an object. Depending on the method invoked, the filters can take actions which extend/modify the original semantics of the object. The path followed by a message is shown in Figure 9.

Messages sent to the object go through a set of *input filters*, each filter looks at the message and if necessary modifies it according to its definition. The message is then passed to the next filter till it reaches the object. Output filters behave similarly, but they work on messages sent *from* the object. A filter consists of three parts: a *condition*, the filter is applied only if it holds; a *pattern*, the filter is applied to matching methods; a *substitution* describing the message

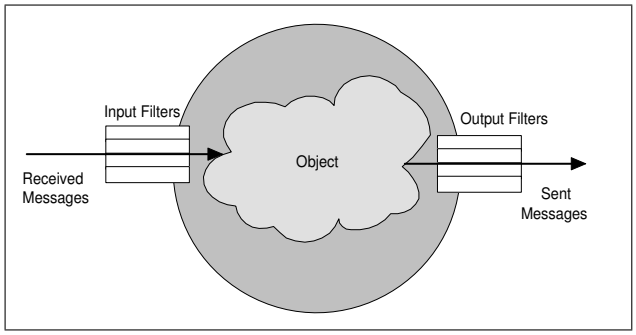

Figure 9: Message path through filters

```
class BoundedBuffer(limit:Integer) interface
  conditions
   Empty; Partial; Full;
  methods
   put(Any) returns Nil;
   get returns Any;
  inputfilters
   bufferSync : Wait =  Empty, Partial=>put,
                        Partial, Full=>get ;
   disp : Dispatch = inner.* ;
end
```

Figure 10: Buffer interface with Comp. Filters

processing performed by the filter.

Another feature of the composition filter model is the *Abstract Communication Types* (ACTs), first introduced in [1] and expressed as composition filters in [2]. An ACT is a class which can *coordinate* the behaviour of different objects by accomplishing some computation depending on the messages they receive.

Synchronization constraints are specified using a *wait* filter. A wait filter can either accept a message and forward it to the next filter or reject it and store it in a *queue* where it will stay till it can be accepted. Wait filters are essentially equivalent to method guards. Filters are specified in *interfaces*, implementations are oblivious to them. Synchronization constraints are thus specified at interface level, cf. e.g. the interface for a bounded buffer of Figure 10.

The interface provides the signatures for the class methods in the *methods* section. The parts of interest are the *condition* section – where the relevant states of the object are named – and the definition of the *input filters*, particularly the definition of the `bufferSync` wait filter. The `bufferSync` filter specifies the synchronization constraints of the object with respect to its conditions. So the method `put` can be executed only if the object is in the state `Empty` or `Partial`, and similarly for the `get` method. The anomaly shows itself in the case of history-sensitive methods. To program the `HistoryBuffer` class in the composition filter model, an ACT is used to administer the history information. A possible implementation, adapted from [5] can be seen in Figure 11.

Unsurprisingly, a new condition `admin.NoRecentGet` is introduced to guarantee that the last method to be executed is not a `get`. This condition is however implemented not in the `HistoryBuffer` class but rather in the `HistoryACT`. The `register` filter takes care of passing every message to the `admin` ACT which will record it and update the `NoRecentGet` accordingly. ACTs can treat messages as first class objects, it is thus easy for the ACT to see what message was received and act accordingly. Note that only the interface of the `HistoryBuffer` class is concerned with synchronization issues. The anomaly has been modularized and confined to interfaces and the ACTs, rather than solved. However, it is clearly a significant improvement over traditional method guards.

```
class HistoryBuffer  interface
   internals
    buf : Buffer;
    admin : HistoryACT;
   methods
    gget returns Any;
   conditions
    admin.NoRecentGet;
   inputfilters
    sync : Wait={ NoRecentGet=>gget, True~>gget };
    buf.bufSync;
    register : Meta= { [*]admin.register };
    disp : Dispatch= { inner*, buf.* };
 end;

class HistoryACT interface
  conditions
   NoRecentGet;
  methods
   register(Message) returns Nil;
  inputfilters
   disp : Dispatch= inner.* ;
end;

class HistoryACT implementation
  instvars
   justGet : Boolean;
  conditions
   NoRecentGet
    begin return justGet.not; end;
  methods
   register(meta:Message) returns Nil;
     begin
      justGet := (meta.selector='get'); meta.fire;
     end;
end;
```

Figure 11: HistoryBuffer with Composition Filters

4.3 Synchronization Rings and Syral

Synchronization rings [11] isolate the following main aspects of the synchronization problem.

Exclusion: The *mutual exclusion* aspect, whereby multiple methods can or cannot be executed concurrently.

State: This aspect specifies which methods are executable when the object is in a certain state.

Coordination: Is the caller *allowed* to execute the method at this time?

These aspects are further refined in two *sub-aspects*:

Response: It specifies what happens if a message cannot be accepted. The options are *blocking*, *balking* (return with an error code) and *timed wait* (wait only a fixed time).

Scheduling: It specifies the order in which waiting messages must be served.

Synchronization rings are based on the abstraction illustrated in Figure 12. The core object implements the behaviour, while the rings around it take care of the synchronization aspects. Incoming messages enter the rings through *entry ports*, reach the core object and leave through *exit* port, on successful execution, or *rollback* ports otherwise. Each ring can have multiple entry ports. Although similar to composition filters, synchronization rings provide low-level hooks to additional aspects for instance scheduling

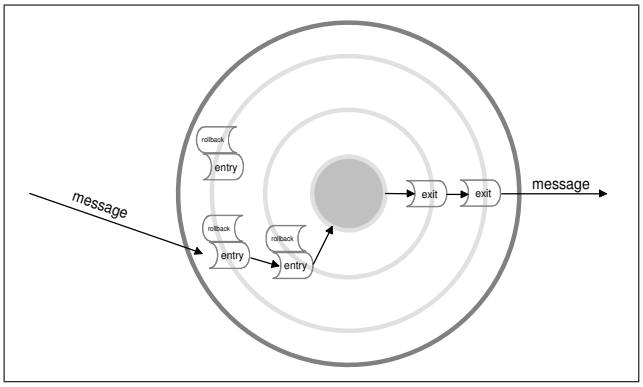

Figure 12: Synchronization Rings Model

```
synchronisation SyncBuffer {
   constructor(maxCapacity);
   core Buffer buf(maxCapacity);
   ring MutualExclusion mutex;
   ring FixedDoubleBoundedSlidingScale
               count(0, maxCapacity);
  put(item) {
     exclusion: uses mutex.mutex;
     hasSpace: uses count.increment;
  }
  get() {
     exclusion: uses mutex.mutex;
     hasItem: uses count.decrement;
  }
}
```

Figure 13: Buffer in Syral

and response behaviour. The synchronization rings model has been implemented in the programming language Syral (**Sy**nchronization **R**ings **A**spect **L**anguage).

In Figure 13 we can see a Syral synchronization definition for a bounded buffer. We need to specify the *core object*, `Buffer` in this case, and the *rings* surrounding it. In our example, we have a *mutual exclusion* ring and a *FixedDoubleBoundedSlidingScale* provided by the standard Syral ring library. The first guarantees that `get` and `put` will be executed in mutual exclusion, the second keeps track of the number of elements in the buffer. The last part of the specification maps messages to ports, in this case both messages must pass through the mutual exclusion ring and the sliding scale ring (through the increase port in the case of `put`, through the decrease one for `get`).

Let us consider the case of history sensitive methods. In Figure 14 we can see an implementation of the `HistoryBuffer` class.

To deal with the new history-sensitive method we are forced to introduce a new *event* ring which will keep track of the history of the object. Note that we must *redefine* the synchronization constraints for the inherited methods to force the inherited messages to pass the event ring. Clearly, this is an instance of the inheritance anomaly. However, as for other AOP approaches, code rewriting affects only the synchronization specification. Further details on Syral and synchronization rings can be found in [11].

4.4 Jeeg

Jeeg [21] is a dialect of Java related to the AOP paradigm. Synchronization constraints, expressed declaratively as guards, are totally decoupled from the body of the method, so as to enhance separation of concerns. A typical Jeeg program has the following form:

```
synchronization SyncHistoryBuffer like SyncBuffer {

   constructor(maxCapacity);
   ring Event lastOpWasGet;

   put(item) { lastOp: uses lastOpWasGet.clearEvent; }

   get() { lastOp: uses lastOpWasGet.setEvent; }

   gget() like get {
      lastOp: uses lastOpWasGet.waitForSetAndClear;
   }
}
```

Figure 14: HistoryBuffer in Syral

```
public class MyClass {
   sync { m : φ; ...  }

   // Standard Java class definition
   public ... m(...) { ... }
  ...
}
```

Intuitively, `m` : ϕ means that at a given point in time a method invocation `o.m()` can be executed if and only if the guard ϕ evaluated on object `o` yields *true*. Otherwise, the execution of `m` is *blocked* until ϕ becomes *true*. The `Buffer` class can be implemented in Jeeg as shown in Figure 15. The novelty of the approach is that guards are expressed in (a version of) Linear Temporal Logic [23] (LTL), so as to allow expressing properties based on the history of the computation. Exploiting the expressiveness of LTL, Jeeg is able to single out situations such as the `HistoryBuffer` class, thus ridding the language from the corresponding anomalies. The syntax of the LTL variation used by Jeeg is the following:

$$\phi ::= \texttt{AP} \mid !\phi \mid \phi \,\&\&\, \phi \mid \phi \,|\, \phi \mid \mathit{Previous}\ \phi \mid \phi\ \mathit{Since}\ \phi$$

Using a meta-variable `event` bound to the name of last method executed, we can code the `HistoryBuffer` example as follows.

```
public class HistoryBuffer extends Buffer {
  sync { gget: (Previous(event != get)) && (!empty); }

  public HistoryBuffer(int max) { super(max); }

  public Object gget() throws Exception { ... }
}
```

In general, the guards can refer to a rich variety of information about methods and fields values.

Due to the nature of the anomaly it is generally not possible to claim formally that a language avoids the problem or solves it. The matter depends on the synchronization primitives of the language of choice, and new experience in OOP may at any time unveil novel shortcomings and new kinds of anomalies. Nevertheless, since the expressive power of LTL is formally understood, a pleasant features of Jeeg is to come equipped with a precise characterization of the situations it can address. More precisely, all anomalies depending on sensitivity to object histories expressible as star-free regular languages can, in principle, be avoided in Jeeg [21].

The current implementation of Jeeg relies on the large body of theoretical work on LTL, that provides powerful model checking algorithms and techniques. Currently, each method invocation incurs an overhead that is linear in the size of the guards appearing in the method's class. Also, the evaluation of the guards at runtime

```
public class Buffer {
    sync {
        put : (! full);
        get : (! empty);
    }
    Buffer(int max) { ... }
    public Object get() throws Exception { ... }
    public void put(Object v) throws Exception { ... }
}
```

Figure 15: Buffer in Jeeg

requires mutual exclusion guarantees that have a (marginal) computational cost. Benchmarks in support of the feasibility of the Jeeg approach, have been presented in [22].

4.5 Advantages of the AOP methodology

As we can see from the four proposals in this section, decoupling behaviour from synchronization has clear advantages. The chief advantage, common to all proposals, is a *modularization* of the synchronization code to be rewritten. This is due to the clear separation of concerns brought forth by the aspect oriented paradigm. Modularization is, however, not enough to truly eliminate the problem. Synchronization code has to be rewritten in certain cases, chiefly due to the history-dependent variety of the anomaly. This issue is even more limited in the case of Jeeg: the use of past-tense linear temporal logics effectively confines the occurrences of the anomaly to a well-defined set of cases: the constraints which cannot be described as star-free regular languages over the objects' traces.

When compared to more traditional COOLs, the AOP approach is thus advantageous. Decoupling synchronization from behaviour seems to fully deliver the promises of the AOP paradigm.

5. CONCLUSION

Ten years after the seminal paper of Matsuoka and Yonezawa the situation of COOLs has changed significantly. Concurrent programming has become a matter of daily practice, at the same time, new programming paradigms have emerged.

In this survey we illustrated the inheritance anomaly problem in the context of the popular programming languages Java and C♯, with the aim of clarifying and putting into perspective the impact of the anomaly. We proceeded to analyze new approaches to the specification of synchronization, devoting particular attention to languages based on the aspect oriented programming paradigm. Our examples showed that regarding the inheritance anomaly, such approaches provide significant advantages over more traditional ones. Our analysis clearly highlights the consequences of well-organized separation of concerns. We believe it marks a strong point of the aspect oriented approach to software development, which clearly goes well beyond the inheritance anomaly problem.

6. REFERENCES

[1] M. Aksit. *On the Design of the Object-Oriented Language Sina*. PhD thesis, University of Twente, Enschede, 1989.

[2] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*. *LNCS*, 791:152–184, 1994.

[3] P. America. POOL: Design and experience. *OOPS Messenger*, 2(2):16–20, Apr. 1991.

[4] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C♯. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*. *LNCS*, 2374:415–440, 2002.

[5] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994.

[6] J.-P. Briot and A. Yonezawa. Inheritance and synchronization in concurrent OOP. In *European Conference on Object-Oriented Programming (ECOOP'87)*. *LNCS*, 276:32–40, 1987.

[7] M. Compton and R. Walker. A run-time system for SCOOP. *Journal of Object Technology*, 1(3):119–157, 2002. special issue TOOLS USA 2002 proceedings.

[8] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM, 1996.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Sun Microsystems, June 2000.

[10] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[11] D. Holmes. *Synchronization Rings – Composable Synchronization for Object-Oriented Systems*. PhD thesis, Macquarie University, 1999.

[12] D. G. Kafura and K. H. Lee. Inheritance in actor-based concurrent object-oriented languages. In S. Cook, editor, *Proceedings of ECOOP'89, Nottingham, UK*, pages 131–145. Cambridge University Press, 1989.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming*, *LNCS*, 1241:220–242, 1997.

[14] C. V. Lopes and K. J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP)*, *LNCS*, 821:81–99, 1994.

[15] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming language. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. 1993.

[16] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales. Aspect-oriented programming. *LNCS*, 1357:483–496, 1998.

[17] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[18] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[19] *C♯Language Specification*. Microsoft Press, 2001.

[20] G. Milicia. *Applying formal methods to programming language design and implementation.* PhD thesis, BRICS – University of Aarhus, 2003.

[21] G. Milicia and V. Sassone. Jeeg: A programming language for concurrent objects synchronization. In *Proceeding of JavaGrande/ISSCOPE 2002*, pages 212–221, 2002.

[22] G. Milicia and V. Sassone. Jeeg: Temporal constraints for the synchronization of concurrent objects. Technical Report RS-03-6, BRICS, February 2003.

[23] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, 1977.

[24] B. Stroustrup. *The C++ Programming Language 2nd edition*. Addison-Wesley, 1991.