CJava: a proposal to circumvent the inheritance anomaly in true concurrent object-oriented languages

Gianpaolo Cugola

cugola@elet.polimi.it
Dipartimento di Elettronica e Informazione
Politecnico di Milano
P.za Leonardo da Vinci 32
20133 Milano (Italy).
Tel.: +39-2-23993666
Fax: +39-2-23993411

Abstract

In order to combine concurrency with object orientation and reuse, the so-called "inheritance anomaly" has to be faced and solved. A large number of solutions have been proposed in the literature to solve the inheritance anomaly in the contest of quasi-concurrent object-oriented languages (i.e., concurrent object-oriented languages where only one activity at a time is allowed to execute in each object). In this paper we survey such proposals and describe a solution to the problem of inheritance anomaly in true concurrent OO languages (i.e., languages where more than one activity at a time are allowed to execute in each object) by presenting the language CJava, an extension to Java featuring true concurrent objects. We also discuss how CJava can be mapped into Java.

1 Introduction

In recent years, there has been great interest in object-oriented programming methodologies. The object-oriented paradigm has been proven to be effective in reducing the effort in designing, developing, and maintaining complex software applications. In the area of programming languages, this lead to a great effort in developing new languages that support the object-oriented paradigm. As a result, most modern programming languages support the basic object-oriented abstractions: encapsulation, classification, inheritance, polymorphism, and late binding.

Another direction of evolution has been the increasing availability of parallel and distributed architectures that may be used for executing software applications structured as a set of parallel, cooperating activities.

The consequence of these two trends was the development of several *Concurrent Object-Oriented Languages* (COOLs) [1,2,3] that support both the object-oriented paradigm and concurrency. Among COOLs we distinguish between two classes of languages:

- The class of *integrated COOLs* is composed of languages that try to tightly integrate concurrent and object-oriented features. They treat objects as active entities that evolve in parallel exchanging messages. Concurrent execution and synchronization are expressed through the creation of new objects and through their interaction. Examples of integrated COOLs will be given in the remainder of the paper.
- The class of *non-integrated COOLs* is composed of languages that keep separate object-oriented and concurrency features. In such languages, concurrent execution is expressed explicitly through the creation of new control threads that communicate and synchronize by invoking operations on shared objects. Examples of non-integrated COOLs are Ada95 [4], Java [5, 6] and Smalltalk [7].

Several studies addressed the issue of determining the interactions between concurrent and object-oriented features in COOLs [8, 9]. They shown that the main conflict between these two classes of features is the difficulty of inheriting the portion of code that expresses the synchronization constraints of objects. This phenomenon is known in the literature as *inheritance anomaly*. Its unfortunate consequences lead to breaking object encapsulation and reducing the expressive power of the language.

In this paper, we are interested in examining more closely the problem of inheritance anomaly in the context of integrated COOLs. As shown in [9], the inheritance anomaly affects all the non-integrated COOLs, which use ad-hoc mechanisms (such as monitors, semaphores, or select statements) to synchronize concurrent threads of execution. This is the reason why these languages usually do not provide inheritance. Conversely, integrated COOLs were developed to solve this and other similar problems. Unfortunately, as we will show in Section 3, most of them do not fully reach this goal.

In the remainder of this paper, we will use the term "COOL" to implicitly refer to an "integrated COOL", thus following the conventional terminology. In such languages, there is no need for separate constructs to create and handle threads (or processes). Objects are intrinsically active entities (we will refer to such objects as *concurrent objects* to distinguish them from traditional sequential objects) that receive messages and answer them by asynchronously executing their methods.

The different COOLs presented in the literature so far may be classified in two classes: *quasi-concurrent object-oriented languages* and *true concurrent object-oriented languages*. Quasi-concurrent object-oriented languages are COOLs in which each object may answer only one message at a time. Conversely, in true concurrent object-oriented languages, objects may answer several messages in parallel. In other words, in true concurrent object-oriented languages, concurrency is not constrained to occur between different objects that execute their methods in parallel, but may occur within the same object that answers to several messages in parallel.

It is easy to imagine that the inheritance anomaly problem is more difficult to solve in true concurrent object-oriented languages. In such languages, the complexity of synchronization constraints tends to be greater than in quasi-concurrent object-oriented languages. This increases the risk of interference between inheritance and concurrency control. Thus, it is not surprising that most proposals that can be found in the literature try to solve the inheritance anomaly problem in

the context of quasi-concurrent object-oriented languages. Only a few proposals address it in the context of true concurrent object-oriented languages.

This paper surveys such proposals and describes a solution to the problem of inheritance anomaly in true concurrent object-oriented languages by presenting an extension to Java [5,6] that introduces true concurrent objects in the language.

The paper is organized as follows: Section 2 describes the inheritance anomaly in detail. Section 3 surveys several COOLs, analyzing their ability to overcome the inheritance anomaly. Section 4 describes the language CJava, an extension to Java providing true concurrent objects. The synchronization primitives of the language are examined to show how they reduce the need for redefining inherited code. A translation scheme from CJava into Java is also sketched. Finally, Section 5 compares CJava with the other COOLs surveyed in Section 3 drawing some conclusions and describing future work.

2 The inheritance anomaly

One of the prime concerns of COOLs (both quasi-concurrent and true concurrent) is the synchronization of concurrent activities. When an object is in a certain state, it can accept only a subset of its set of messages in order to maintain its internal integrity.

As an example, consider a bounded stack of integers `BStack` with methods `push` to add an integer on top of the stack, `pop` to remove the topmost integer, and `top` to read the value of the topmost integer without removing it. The clients of this stack have to honor the following obligations: the method `push` cannot be called if the stack is full and both `pop` and `top` methods cannot be called if the stack is empty.

In a sequential environment (i.e., in an environment in which a single thread of execution exists) these obligations represent correctness conditions. If, prior to a call to the `put` method the `BStack` is full, this simply indicates a bug in the client containing the call. Consequently, the implementation of the class `BSTack` using a sequential language does not have to include any special code to manage these obligations: it is a responsibility of the clients to guarantee their respect.

Conversely, in a parallel environment, a client of the class `BStack` does not have any means to ensure that the above mentioned obligations are honored before calling one of the `BStack` methods. Testing the state of the stack to avoid errors does not solve the problem. Between the time the test is performed and the time of the call another client running in parallel with the first may change the object's state. As a result, in a parallel environment, the above mentioned obligations have to be treated as *synchronization constraints*. Issuing a `push` call to a full `BStack` should not result in an error, but in delaying such call until the stack state changes to "not full". It is a responsibility of the `BStack` to satisfy these synchronization constraints by delaying the calls to its methods, if needed.

Observe also that if the language used to implement the class `BStack` is a true concurrent object-oriented language, other synchronization constraints have to be added to take care of the fact that, potentially, each `BStack` object may answer several calls in parallel. Such new synchronization constraints have to guarantee that while a `BStack` is answering a message `push` or `pop` (by executing its `push` or `pop` methods, respectively) it does not answer other messages. In fact, methods `push` and `pop` change the internal state of the stack and executing them in parallel would possibly result in an inconsistency.

Several research studies pointed out that the *synchronization code* (i.e., the portion of the code of a concurrent object used to guarantee the respect of its synchronization constraints) can only be inherited in the case of trivial class extensions. As we mentioned in Section 1, this conflict between concurrency and inheritance in COOLs is called *inheritance anomaly*.

In order to analyze the inheritance anomaly, it is important to distinguish the different COOLs with respect to the *synchronization scheme* they adopts, that is, with respect to the mechanism they use to implement the synchronization constraints of concurrent objects. In particular, we distinguish among four classes of COOLs:

- *Languages based on method preconditions*. COOLs belonging to this group allow the programmer to provide a boolean expression for each method of each concurrent object. It specifies the conditions that have to be satisfied in order for the corresponding message to be accepted.
- *Languages based on active objects with bodies*. Concurrent objects implemented using such languages have their own thread of control that executes a particular method, often called "the body" of the object. The body describes the order in which messages can be accepted and the methods that have to be called to answer them.
- *Languages based on accept sets*. Each method of each concurrent object implemented using such languages specifies, often as its last instruction, the next *accept set*. Each accept set is a set of message names (or message signatures). The messages that belong to the current accept set may be accepted, while the other messages are delayed until they become part of the current accept set.
- *Languages based on mixed approaches*. This group includes the languages whose synchronization scheme is based on a combination of the previous approaches.

This distinction is important because, as pointed out in [10, 11], the occurrence of the inheritance anomaly depends on the *synchronization scheme* of the language. The same class may be inherited with no problems if it is implemented in a particular language, while it may require rewriting of one or more of its methods if it is implemented in other languages which use a different synchronization scheme. An example will support this claim.

Consider a quasi-concurrent language whose synchronization scheme is based on active objects with body[1]. Figure 1 shows the implementation of the class `BStack` using such language. Each concurrent object has its own control thread that executes the special method named `body`. The body uses the statement `answer` to describe the set of messages that may be answered. To answer them, the thread executing the body calls the corresponding method. Suppose now that we need to create a subclass `BStack2` of `BStack` to implement a bounded stack exporting a `pop2` method. When invoked, `pop2` pops the first and the second integer off the stack as a single atomic operation. As shown in Figure 2, the additional synchronization constraints imposed by the introduction of the method `pop2` require the body of the class `BStack` to be entirely rewritten in class `BStack2`.

Conversely, consider a quasi-concurrent language whose synchronization scheme is based on explicit method preconditions. As shown in Figure 3, the introduction of the class `BStack2` using such language does not require a redefinition of any of the parent class methods: we only need to add the `pop2` method with its own precondition.

[1] To reduce the effort required to understand a completely new language, we used a hypothetical language whose syntax resembles C++.

```
class BStack {
 protected:
  int next, size, *buf;
  void body() {
    while(1) {
      if (next<size && next>0) answer(push, pop, top);
      else if(next==0) answer(push);
      else if(next==size) answer(pop, top);
    }
  }
 public:
  BStack(int s) {
    next=0; size=s;
    buf=new int[size];
  }
  void push(int i) {
    buf[next++]=i;
  }
  int pop() {
    return buf[--next];
  }
  int top() {
    return buf[next-1];
  }
}
```

Figure 1: The class BSTack implemented using a quasi concurrent language based on active objects with body

```
class BStack2 : public BStack {
 protected:
  void body() {
    while(1) {
      if(next<size && next>1) answer(push, pop, pop2, top);
      else if(next==1) answer(push, pop, top);
      else if(next==0) answer(push);
      else if(next==size) answer(pop, pop2, top);
    }
  }
 public:
  void pop2() {
    next=next-2;
  }
}
```

Figure 2: The class BStack2 implemented using a quasi concurrent language based on active objects with body

```
class BStack {
 protected:
  int next, size, *buf;
 public:
  BStack(int s) {
    next=0; size=s;
    buf=new int[size];
  }
  void push(int i)
   precondition next<size;
  {
    buf[next++]=i;
  }
  int pop()
   precondition next>0;
  {
    return buf[--next];
  }
  int top()
   precondition next>0;
  {
    return buf[next-1];
  }
}
```

```
class BStack2 : public BStack {
public:
  void pop2()
   precondition next>1;
  {
    next=next-2;
  }
}
```

Figure 3: Classes BSTack and BStack2 implemented using a quasi concurrent language based on method preconditions

This example shows that the ability of inheriting the synchronization code of concurrent objects depends on the synchronization scheme adopted by the language. In [10] several other examples may be found, which show how the inheritance anomaly affects different kinds of COOLs depending on the synchronization scheme they adopt.

Section 3 surveys several COOLs with particular attention to the synchronization scheme they adopt in order to understand if (and how) they solve the inheritance anomaly.

3 A survey of existing COOLs

In this section, we analyze some of the existing COOLs (both true and quasi-concurrent) focusing on the features that may have an impact on the inheritance anomaly. In particular, we focus on the synchronization scheme they adopt to understand if it solves the inheritance anomaly.

This survey does not have the ambition of being exhaustive. A large number of languages exist that support both concurrency and object-orientation and analyzing all of them would be impossible. We chose the languages that are most often cited in the literature and whose documentation may be obtained more easily. Moreover, being interested in analyzing the synchronization scheme adopted by these languages in order to verify if it solves the inheritance anomaly, we discarded the COOLs that do not support inheritance.

The subsections that follow survey the chosen languages, distinguishing them with respect to the synchronization scheme they adopt.

3.1 Languages based on method preconditions

A method precondition is a boolean expression that specifies the conditions that have to be satisfied in order for a method to be executed (i.e., in order for the corresponding message to be answered). As shown by the examples in Figure 3, by using method preconditions, it is possible to easily describe the synchronization constraints of concurrent objects.

Unfortunately, even languages based on method preconditions suffer of a particular form of inheritance anomaly. This form of inheritance anomaly manifests itself when one or more attributes have to be added to a subclass in order to implement its synchronization constraints. In particular, the inheritance anomaly occurs if such new attributes have to be modified also by the methods inherited from the parent class in order to correctly implement the subclass synchronization constraints. The anomaly occurs also if the methods added in the subclass impact on the synchronization constraints of the methods inherited by the parent class. Several examples of such kind of inheritance anomaly were given in [10] and they are included in Section 4.3. There we show that the impact of this form of inheritance anomaly can be greatly reduced if the synchronization scheme of the language allows method preconditions and method bodies to be inherited separately and to be specified incrementally.

As suggested by its name, CEIFFEL [12] is a concurrent extension of Eiffel [13]. All the objects declared in CEIFFEL are concurrent in the sense described in Section 1. They are active entities that wait for messages and answer them by asynchronously invoking their methods. To express the synchronization constraints of concurrent objects, CEIFFEL modifies the semantics associated to Eiffel method preconditions. In Eiffel, preconditions are used to describe the constraints that the caller has to satisfy in order for the callee to produce the correct results. Violating a precondition results in raising an exception. In CEIFFEL method preconditions are split in two parts: the *checker* and the *guard*. The checker part maintains the standard Eiffel semantics i.e., violating it results in raising an exception. The guard is used to describe the synchronization constraints of the method. It has to be satisfied in order for the corresponding message to be accepted and answered. Each message received while the guard of the corresponding method is false is delayed until the guard becomes true.

In CEIFFEL asynchronous invocation of methods is not confined to procedures, the invocation of a function (i.e., a routine returning a value) also immediately returns, delivering a *proxy* for the result. By means of proxies CEIFFEL implements the *wait-by-necessity* approach to synchronization[2]. According to this approach, the caller and the callee synchronizes only when the caller need to access the value returned by the call. When this happens, the caller suspends itself waiting for the termination of the callee method that have to calculate the result.

CEIFFEL is one of the few examples of true concurrent object oriented language that may be found in literature. Each CEIFFEL concurrent object may answer several messages in parallel. Ad hoc annotations (*compatibility annotations*) are added to method declarations to describe the fact that the corresponding messages may be answered in parallel with themselves or with other messages.

CEIFFEL does not offer any particular feature to overcome the particular form of inheritance anomaly that affects the languages based on method preconditions.

DistributedConcurrentSmalltalk (DCST) [15] extends ConcurrentSmalltalk (an extension to Smalltalk that adds concurrent objects to SmallTalk [16]) by introducing new features to program the synchronization constraints of concurrent objects and by introducing true concurrency. Two mechanisms are provided by DCST to program the synchronization constraints of concurrent objects: *method relations* and *guarded methods*. Method relations are used to serialize several activities into the same object. If a method relation is specified for two methods of the same class then these

[2] The *wait-by-necessity* approach to synchronization of callers and callees was described for the first time in [14].

methods cannot be executed concurrently. Moreover, DCST methods may provide a guard (i.e., a precondition). It is a boolean expression used to control if messages may be immediately answered or if they have to be delayed. When a message is received, the corresponding method is executed only if its guard evaluates true. Otherwise, the message is delayed until its guard becomes true. With respect to inheritance, DCST suffers of the limitations of the languages based on method preconditions. In particular, inherited methods have often to be rewritten in order to extend their guards.

In BLAZE 2 [17] methods are either serial or parallel. Serial methods are executed in mutual exclusion whereas parallel methods may be executed concurrently with other parallel methods of the same object. Declaring a method to be serial or parallel is a particular form of precondition. This explains why we decided to classify BLAZE 2 among the languages whose synchronization scheme is based on method preconditions. Concurrent access to class attributes is controlled by explicitly locking them. The language provides both read and read-write lock primitives. BLAZE 2 does not offer any mechanism to overcome the inheritance anomaly.

In [18,19] an innovative approach to describe the synchronization constraints of concurrent classes is described in the context of the language DRAGOON, a quasi-concurrent extension to Ada. Synchronization constraints for DRAGOON concurrent classes are specified separately from the sequential features of the classes themselves by mean of *behavioral classes* (b-classes). Each b-class is an abstract, generic specification of a behavior. Multiple inheritance, called *behavioral inheritance* is exploited to associate synchronization constraints, expressed via b-classes, to *free classes* (i.e., concurrent classes that do not provide synchronization constraints). A b-class abstractly specifies synchronization constraints to be imposed to method activations independently from the actual methods of any free class by using formal method names that are bound to actual method names at behavioral inheritance time. For each formal method name, the b-class specifies a precondition expressed using deontic logic.

The most relevant limitation of the approach described in [18,19] is that neither b-classes nor the *controlled classes* obtained by combining (via inheritance) the synchronization constraints expressed by a b-class with a free class, can be sub-classed. The authors claim [18] that: "composing the behavior of two classes is not essential, because behaviors are seldom so complex to justify an inheritance taxonomy".

3.2 Languages based on active objects with bodies

Languages belonging to this class use a separate thread of control for each concurrent object, which is in charge of managing the messages sent to the object. Such thread of control executes a special method (the object's "body") that specifies, usually by using guarded commands, the order in which messages have to be answered and the methods that have to be called to answer them. An example of how these languages may be used to implement the synchronization constraints of a concurrent object has been given in Figure 1 and in Figure 2.

In POOL-I [20,21] (a quasi-concurrent object oriented language) parallelism is introduced by giving every object a body which is in charge of controlling inter-objects communication. The communication is based on synchronous message passing. The object that sends the message is blocked until the receiver answers the message by executing the corresponding method. Both sending and receiving a message take place explicitly. In POOL-I, the body is not inherited in subclasses and has to be redefined in order to account for the new methods introduced into the subclass.

Eiffel// [22] is a quasi-concurrent object-oriented language that extends Eiffel adding concurrent objects to the language. Each Eiffel// concurrent object (called *process*) is provided with a special method (the `live` method) that specifies the messages that can be accepted depending on the

state of the object. The `live` method plays the role of the active object's body and, as mentioned in Section 2, it has to be redefined in subclasses if new methods are added. This have to be done even if the synchronization constraints related to the other (i.e., inherited) methods do not change.

Eiffel// is worthy of being mentioned not for the way it overcomes the inheritance anomaly (as mentioned it does not succeed in this goal) but because historically it was the first language using the wait-by-necessity approach to synchronization [14]. When a process `P` sends a message `M` that requires an answer to a process `P1`, it does not need to wait for P1 to compute the answer. P may continue its execution in parallel with P1 (which is computing the answer) until it needs to access the value included in the answer to the message M. When this happens, P suspends, waiting for P1 to finish computing the answer to M. The wait-by-necessity implements an efficient and easy to use form of synchronization driven by data.

In [23] an Eiffel library is described, which provides concurrency abstractions. The form of concurrency implemented is very similar to that implemented in EIFFEL// and suffers of the same problems. Both provide quasi-concurrency and allow synchronization constraints to be expressed through a particular method whose role is of explicitly accepting incoming messages. The core of the parallel library described in [23] is the class `CONCURRENCY` that embodies the methods to accept messages and send back results to the callers. Each class inheriting from the class `CONCURRENCY` have to provide the method `scheduler` which plays the role of the body (i.e., it plays the same role of the method `live` in EIFFEL//). To extend a concurrent class the user needs to completely redefine the `scheduler` method even if only a single method has to be added.

μC++ [24] extends C++ by introducing *coroutines* and *tasks*. Each task provides a special method (the `main`) which is concurrently started when the task is created. The `main` controls the way messages are accepted and answered by using `accept` statements similar to that provided by Ada. Like other languages based on a "body" which is in charge of controlling message reception, μC++ suffers of the inheritance anomaly.

3.3 Language based on accept sets

An accept set is a set of message names or message signatures. Accept sets are used to specify the synchronization constraints of concurrent objects by describing the set of messages that can be accepted after the execution of each method (i.e., the new accept set). In fact, only messages that belong to the current accept set may be accepted. The other messages are delayed until they become part of the current accept set. As an example, Figure 4 shows the implementation of the class `BStack` described in Section 2 using a language based on accept sets[3].

Languages based on accept sets suffer of a particular form of inheritance anomaly whose cause is the need of partitioning the accept sets to implement the synchronization constraints of the methods added in a subclass.

[3] To reduce the effort required to understand a completely new language, we used a hypothetical language obtained by extending C++ with the minimal set of new syntactical elements needed to describe accept sets and the operations to manage them.

```
class BStack {
 accept-set:
  empty={push}; partial={push, pop, top};  full={pop, top};
 protected:
  int next, size, *buf;
 public:
  BStack(int s) {
    next=0; size=s;
    buf=new int[size];
  }
  void push(int i) {
    buf[next++]=i;
    if(next==size) become full; else become partial;
  }
  int pop() {
    if(next==1) become empty; else become partial;
    return buf[--next];
  }
  int top() {
    return buf[next-1];
  }
}
```

Figure 4: The class BSTack implemented using a quasi concurrent language based on accept sets.

As an example, consider the class `BStack` described in Figure 4. Its accept sets are: `empty={push}`, `partial={push,pop,top}`, and `full={pop,top}`. Suppose we have to create the subclass `BStack2` of `BStack` described in Section 2. As shown in Figure 5, in order to implement the synchronization constraints of the method `pop2` we have to partition the accept set `partial` into two different accept sets: `one={push,pop,top}` and `more_that_one={push,pop,pop2,top}`[4]. By doing so we will be able of distinguishing the state in which the stack contains only one element (and `pop2` messages cannot be accepted) from the states in which the stack contains two or more elements. The main consequence of such partitioning is the need of redefining the inherited methods `push` and `pop` in order to cope with the new accept sets that partitioned the original accept set `partial`.

ACT++ [25] is a true concurrent extension to C++ based on the actor computation model of Agha and Hewitt [26]. Each concurrent object (called *Actor* in ACT++) provides a set of methods and a set of behaviors. Each behavior is represented by a name associated to a set of methods. Only the messages corresponding to the methods that belong to the current behavior may be accepted. The statement `become` is used to specify the next behavior.

In ACT++ each behavior may be redefined in subclasses. This reduces the occurrence of the inheritance anomalies. As an example, it is possible to create a subclass of the bounded stack described in Section 2 that adds the method `popFirst()`. It returns the first element put into the buffer (as opposite to the inherited method `pop()` that returns the last). To create such subclass the programmer needs to redefine the behaviors `full` and `partial` as follow:

```
full={pop,top,popFirst()};
partial={push,pop,top,popFirst()};
```

[4] The accept set `full` have to be modified also (i.e., becoming `full={pop,pop2,top}`) but this change does not impact in the code of the inherited methods.

```
class BStack2 : public BStack {
accept-set:
 // accept set empty is inherited
 one=partial; //inherited from superclass
 more_than_one={push, pop, pop2, top};
 full={pop, pop2, top};
public:
 void push(int i) {
   buf[next++]=i;
   if(next==size) become full;
   else if(next>1) become more_than_one;
   else become one;
 }
 int pop() {
   if(next==1) become empty;
   else if(next==2) become one;
   else become more_than_one;
   return buf[--next];
 }
 void pop2() {
   next=next-2;
   if(next==0) become empty;
   else if(next==1) become one;
   else become more_than_one;
 }
}
```

Figure 5: The class BSTack2 implemented using a quasi concurrent language based on accept sets.

Unfortunately, the example of the class `BStack2` made in Section 2 shows that this approach does not fully solve inheritance anomaly.

ConcurrentSmalltalk-90 [27] is a quasi-concurrent language that extends ConcurrentSmalltalk [16] by introducing new features to program the synchronization constraints of concurrent objects. ConcurrentSmalltalk-90 is not directly based on accept sets but uses a similar approach that has the same pros and cons. Concurrency in ConcurrentSmalltalk-90 is based on *atomic objects*. They are concurrent objects capable of answering messages by asynchronously invoking their methods. Atomic objects methods have two modes: *active* and *inactive*. Messages for methods in active mode are served as soon as possible. Conversely, messages for methods in inactive mode are postponed until their mode is changed to active. Two methods are provided to change the mode of a method from inactive to active and vice versa. This approach presents pros and cons of accept sets. To extend an existing atomic object by introducing new methods, the ConcurrentSmalltalk-90 programmer often needs to redefine the inherited methods in order to add the statements that make the added methods active or inactive depending on the object's synchronization constraints.

In Hybrid [28] each method may be associated to a *delay queue*. It is an instance variable of the object that may be in one of two states: *open* or *closed*. If the delay queue associated to a method is open, calls to that method are accepted, otherwise they are suspended until the queue opens again. Delay queues' state may be changed only by the methods of the object they belong to. The resulting synchronization scheme is similar to that provided by ConcurrentSmalltalk-90 and suffers of the same limitations.

Concurrent object-oriented C (cooC) [29] introduces concurrency into Objective-C. It is a true-concurrent object-oriented language providing a simplified form of method preconditions to

implement synchronization constraints. The cooC programmer may only specify those methods that cannot be served concurrently (*exclusive methods*). Moreover, cooC methods are usually served by respecting the FIFO order but the programmer may explicitly change this behavior by specifying the next message to process. This last approach is similar to that taken by ConcurrentSmalltalk-90 and has the same drawbacks with respect to the ability of inheriting the synchronization code.

3.4 Languages based on mixed approaches

In [11] an extension to the ABCL language is proposed, which shows how the inheritance anomaly may be entirely solved in quasi-concurrent object-oriented languages. The approach combines the advantages of both method preconditions and accept sets. The programmer may choose the synchronization scheme she/he prefers and each subclass may use a synchronization scheme different from that of their superclasses. This is possible because the two synchronization schemes chosen are orthogonal. Moreover, synchronization code is separated from the body of methods so that they can be inherited and redefined separately. This approach eliminates the need of redefining of method bodies when only the synchronization code has to be redefined.

4 CJava: extending Java with true concurrent objects

This section describes CJava, an extension to Java providing true concurrent objects. As mentioned in Section 1, CJava has been developed to show how the inheritance anomaly can be solved in the context of a true concurrent object-oriented language. As a consequence, we focused on the synchronization scheme adopted by the language, leaving out of account the problems related to efficiency.

The section is organized as follow: Subsection 4.1 motivates our choice of using Java as a base language for our COOL. Subsection 4.2 describes CJava. Subsection 4.3 demonstrate, through several examples, that CJava does not suffer of the inheritance anomaly that affects the other true concurrent object-oriented languages surveyed in Section 3. Finally, Subsection 4.4 describes the approach used to implement CJava.

4.1 Why Java

Released by Sun at the end of 1994, Java is rapidly becoming one of the most popular object-oriented languages. One of the main reasons of this success is that Java syntax is similar to C++ syntax. This made it easy moving from C++ to Java and promoted the adoption of Java by the large community of C++ programmers. On the other hand, Java semantics is more similar to other, more sophisticated, object-oriented languages, like Eiffel, Objective C, Cedar/Mesa, and Smalltalk and this further supported Java adoption. According to many of its users, Java can be viewed as a "more elegant and simple C++".

Another essential feature of Java is its portability. The Java compiler generates *bytecodes*, an architecture-neutral, intermediate format, designed to transport the code efficiently to multiple hardware and software platforms. Java bytecode is interpreted by the Java interpreter that realizes the *Java Virtual Machine*. The Java Virtual Machine abstracts from the hardware and software characteristics of existing "concrete machines", allowing the Java bytecode to be portable to all the platforms that provide a Java interpreter[5].

[5] The Java Virtual Machine is available for a great number of platforms including Sun Solaris on SPARC architectures, Microsoft Windows95/NT, Linux, and IBM OS/2 on Intel X86 architectures, MacOS and AIX on PowerPC architectures and others.

Popularity and portability were not the only reasons that motivated our choice of using Java as the base language for our COOL. The most important reason is that Java supports multithreading at the language level as well as some synchronization primitives that allow us to easily provide a translation from our concurrent extension to pure Java. By using such approach based on translation, we gave our extension the same characteristics of portability that Java provides.

Java supports concurrency by means of the class `Thread` provided as part of the Java standard library. The class `Thread` provides a rich collection of methods to start a thread, stop a thread, and check the status of a thread. To control the access to data shared by different threads, the Java language provides monitors [30] and condition lock primitives.

To create a new thread, the Java programmer creates a subclass of the class `Thread`, redefining its `run` method. Such a new subclass is then instantiated to create a thread object that may be started by calling its `start` method. After being started, a thread object executes its `run` method in parallel with the thread that called the `start`[6].

Threads communicate among themselves by means of *condition variables*. Condition variables are passive objects defined by a class exporting one or more `synchronized` methods. Methods within a class that are declared `synchronized` run under control of *monitors* to ensure that the internal state of the called object remains consistent. As defined by Hoare in [30], a monitor associated with a specific condition variable works as a lock on that variable. When a thread holds the monitor for some condition variable other threads are locked out and cannot inspect or modify its contents.

In Java, every class and instantiated object has its own monitor that comes into play if required. In particular, to execute a synchronized method on an object `O`, a thread has to acquire the monitor associated with `O`. The monitor is automatically released when the synchronized method returns.

Java provides also the methods `notify` and `wait` that can be invoked from within a synchronized method to coordinate the activities of multiple threads that need to use the same resources. The `wait` method causes the current thread to release the monitor it holds and to wait (possibly forever) until another thread notifies it that some condition has changed (by calling the `notify` method).

An example will give an idea of how a multi-threaded program can be implemented in Java. Suppose we need to develop a concurrent version of the `Printer` class described in Figure 6a. Suppose, it has to be implemented in such a way that when the `print` method is called, a new thread is started to print the document in parallel with the caller. The new thread has to wait the other printing jobs to complete before printing the document to avoid conflicts while accessing the printer driver. Figure 6b shows the Java implementation of such concurrent printer.

The example suggests that programming concurrent applications in Java is not a trivial task. Ad-hoc thread classes (e.g., the `PrintThread` class) have to be created, threads have to be explicitly instantiated and started, and the code to guarantee that shared resources are accessed in mutual exclusion has to be added (e.g., the `synchronized` statement in method `print_impl`). The matters become more and more complex when more than one method may run concurrently into the same object. In such case, synchronization between the different threads has to be carefully programmed by using the `wait` and `notify` methods (this feature is not present in our simple example). Conversely, COOLs like CJava make concurrency transparent to the programmer. CJava programmers are not required to take care of the details of concurrency related issues and they might concentrate on application programming.

[6] Java provides also a different approach to create new threads with no need of subclassing the class `Thread`, but it also requires the identification of a particular method (i.e., the `run` method) which is the only method that runs concurrently with the caller after thread creation.

```
class Printer {
  protected boolean onLine=false;
  public void online() {
    onLine=true;
  }
  public void offline() {
    onLine=false;
  }
  public void print(Document d) {
    if(online) {
      ... // print the document
    } else return;
  }
}
```

a) the class `Printer`

```
class ConcurrentPrinter {
  protected boolean onLine=false;
  public void online() {
    onLine=true;
  }
  public void offline() {
    onLine=false;
  }
  public void print(Document d) {
    PrintThread pt=new PrintThread(this,d);
    pt.start();
  }
  public synchronized void print_impl(Document d) {
    if(online) {
      ... // print the document
    } else return;
  }
}

class PrintThread extends Thread {
  private Document doc;
  private ConcurrentPrinter printer;
  public PrintThread(ConcurrentPrinter p, Document d) {
    printer=p;
    doc=d;
  }
  public void run() {
    printer.print_impl(doc);
  }
}
```

b) the class `ConcurrentPrinter`

Figure 6: A concurrent printer in Java

4.2 Language description

CJava is a true concurrent object-oriented language based on Java. It extends Java by providing three main constructs: *concurrent classes*, *method preconditions* and *futures*.

Concurrent classes

Concurrent classes are classes whose objects are able to answer asynchronously the messages they receive by invoking new threads of execution that run in parallel with the thread of the caller. When a concurrent object (i.e., an object that belongs to a concurrent class) receives a message, a new thread is started to execute the corresponding method. Thereafter, the caller and the callee run in parallel.

A concurrent class is defined in CJava by placing the keyword `concurrent` before the class definition. Figure 7 shows the CJava implementation of a concurrent queue[7] (Figure 7a) and the way it could be used by a client (the class `ConcurrentQueueClient` in Figure 7b). As shown in Figure 7b, a concurrent object may be instantiated by using the standard Java statement `new`. Figure 7b shows also that messages are sent to concurrent objects by using the standard dot notation (e.g., the statement `cq.enqueue(5)` that adds the integer 5 to the queue `cq`).

[7] The concurrent queue reported in figure lacks of synchronization constraints. They will be introduced later, when method preconditions will have been described.

```
concurrent class ConcurrentQueue {
  protected Node head, tail;
  protected int nElements;
  public void enqueue(int item) {
    if(isEmpty())
      head=tail=new Node(item,null);
    else tail.next=new Node(item,null);
    nElements++;
  }
  public void dequeue() {
    if(nElements==1) {head=tail=null;}
    else head=head.next;
    nElements--;
  }
  public int top() {
    return head.data;
  }
  public boolean isEmpty() {
    return nElements==0;
  }
}

class Node {
  int data;
  Node next;
  Node(int i, Node n)
    {data=i; next=n;}
}
```

a) ConcurrentQueue class

```
class ConcurrentQueueClient {
  public void main(String args[]) {
    ConcurrentQueue cq=new ConcurrentQueue();
    cq.enqueue(5);
    cq.enqueue(7);
    System.out.println(cq.top());
    cq.dequeue();
  }
}
```

b) ConcurrentQueueClient class

Figure 7: A concurrent queue in CJava (without synchronization constraints)

As already mentioned, message answering in CJava is asynchronous. This means that, as an example, after sending the message `cq.enqueue(5)` the `main` method of the class `ConcurrentQueueClient` may continue executing without the need of waiting the completion of the method `enqueue`. To distinguish methods of concurrent classes (that run concurrently with the caller method) from methods of standard Java classes, we will call the former *concurrent methods* and the latter *non-concurrent methods*.

Concurrent classes may inherit from other classes by using the standard Java statement `extends`. Like Java, CJava supports single inheritance only. The exact semantics of class inheritance in presence of concurrency will be detailed later, after describing method preconditions. By now, it is enough to say that a concurrent class inherits all the attributes and methods of its superclass obeying to the standard visibility rules defined in Java. By default, a concurrent class that does not extend any other class is considered to extend the concurrent class `ConcurrentObject` that is part of the CJava run-time system. It exports the same methods of the Java class `Object` but they are executed asynchronously.

The main constraint regarding inheritance in CJava is that concurrent classes cannot be subclasses of non-concurrent classes and vice versa. As a consequence, the inheritance hierarchy of concurrent classes (whose root is represented by the `ConcurrentObject` class) and the inheritance hierarchy of non-concurrent classes (whose root, according to the Java language specification, is represented by the `Object` class) are kept separate. This avoids any interference between concurrent and non-concurrent methods within the same class.

Methods of concurrent classes (hereafter "concurrent methods") are specified as standard Java methods with three modifications:

1. they may provide preconditions (see next subsection);

2. they cannot be synchronized in the Java sense, i.e., via `synchronized` methods and `wait` and `notify` primitives. Consequently, synchronization constraints have to be expressed using method preconditions, only. If a concurrent method would be allowed to be synchronized, the synchronization constraints stated by its precondition could conflict with those resulting from the use of the Java synchronization primitives;
3. they cannot throw exceptions. In fact, the standard behavior of exception handling cannot be naturally integrated with the asynchronous semantics of concurrent method calls. In languages supporting exception handling like Java, Ada, and C++, when an exception is thrown the runtime stack is unwound by following the dynamic chain of allocated frames until a unit providing a suitable exception handler is found; or, if none is found, the program is terminated [31]. The implicit assumption is that unit activation is synchronous. Unfortunately, this assumption is not valid in COOLs. If a concurrent method could throw an exception, it would be unclear if and how the exception should be propagated to the unit that generated the asynchronous call. Thus, for simplicity, we decided to forbid exception throwing by concurrent methods.

As a final assumption, the local method calls issued by concurrent methods (i.e., calls to the same object) are executed synchronously. No precondition evaluation is performed. This reduces the chance of deadlock and makes the design of concurrent classes simpler and more natural.

Method preconditions

CJava's synchronization scheme is based on a particular form of method preconditions that, combined with the ability of separately redefine methods' preconditions and methods' bodies, overcome the inheritance anomaly, as we will show in Subsection 4.3.

Each concurrent method may provide its own precondition. It is a boolean expression placed after the keyword `precondition` attached to the method's signature. A precondition may refer to object's attributes and to method parameters, and it may include calls to other methods[8].

When a message `M` is sent to a concurrent object `CO`, the message is added to the `CO`'s *list of pending messages* (LPM). Then, a new thread is created to evaluate the precondition associated with the method corresponding to message `M`. If the precondition is true, the message `M` is removed from LPM, the method corresponding to `M` is added to `CO`'s *list of running methods* (LRM) and its body is executed. If the precondition is false, the thread waits until it becomes true. When the execution of the body ends, the method is removed from the `CO`'s LRM.

As mentioned in the previous subsection, this rule (i.e., asynchronous invocation of methods with preconditions used to enforce synchronization constraints) is broken for local calls only. Actually, a local call, is executed synchronously (i.e., using a standard procedure call) and the precondition of the called method (if any) is ignored.

The operation that adds a message to the list of pending messages is executed synchronously with message passing. This means that the order of the pending messages in LPM reflects the order of the calls. Similarly, the procedure that: (1) evaluates the precondition of a method whose message is pending, (2) deletes the corresponding message from LPM, and (3) adds the method to LRM, is executed atomically. Figure 8 formalizes this semantics (i.e., the semantics of CJava message passing and preconditions evaluation) by using an ER Petri net.

[8] The number of times each precondition is evaluated cannot be determined in advance. As a consequence, preconditions should only include calls to methods that do not have side effects.

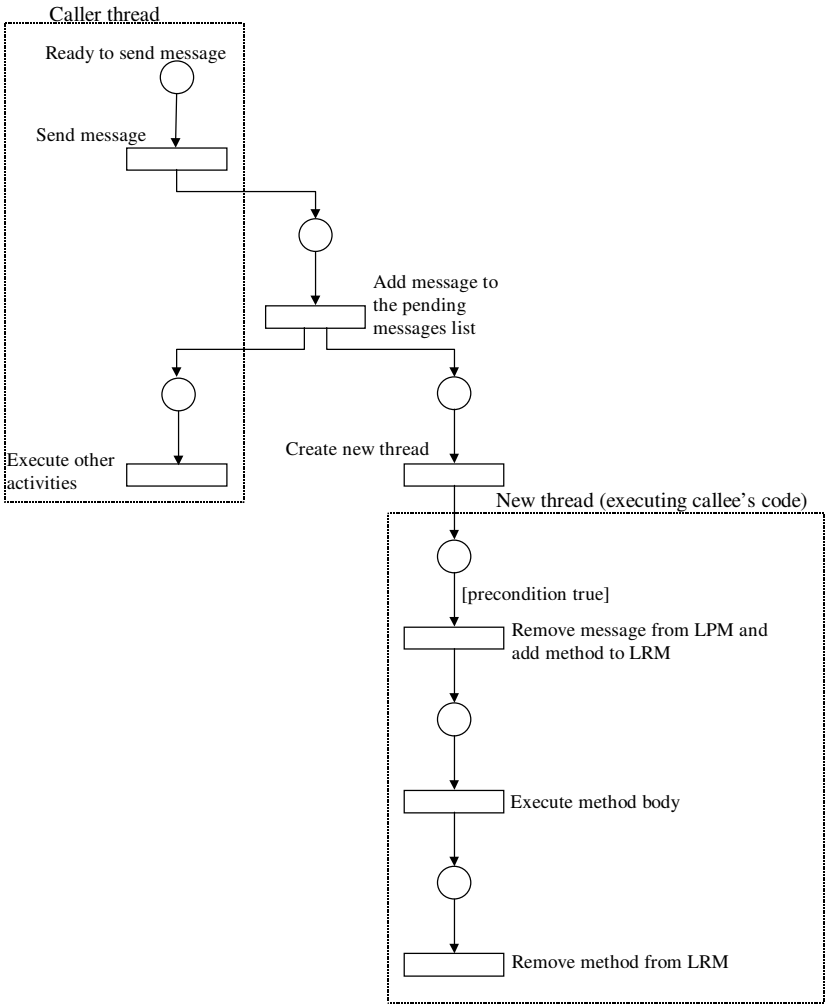

Figure 8: The CJava method call scheme

```
concurrent class ConcurrentPrinter {
  protected boolean onLine=false;
  public void online() {
    precondition !running("online") && !running("offline"){
    onLine=true;
  }
  public void offline()
    precondition !running("offline") && !running("online"){
    onLine=false;
  }
  public void print(Document d)
    precondition onLine && !running("print") && pendingBefore("print")==0 {
    ... // print the document
  }
}
```

Figure 9: A concurrent printer in CJava

In traditional OO languages, if an invocation `x.op1` is followed by an invocation `y.op2`, the execution of `op1` precedes the execution of `op2`. *This is not guaranteed in CJava when* `x` *and* `y` *are concurrent objects. It is not guaranteed also if the methods* `op1` *and* `op2` *are invoked on the same object* `x`. As we may easily deduce by looking at Figure 8, the CJava run-time system does not guarantee that the

execution order of called methods reflects the order of calls. The CJava programmer is free of choosing the ordering she/he prefers in answering messages. In particular, the programmer can implement synchronization constraints based on the order in which messages are received by using the CJava predicates `pending`, `pendingBefore` and `running`.

- Predicate `pending("method name")` is used to evaluate if a certain message is pending (i.e., it belongs to LPM). The predicate returns true if a call to the method whose name is "`method name`" has been issued but the corresponding method body did not start running yet.

- Predicate `pendingBefore("method name")` returns an integer representing the number of messages corresponding to the method whose name is "`method name`" which precede the message that is being processed in the list of pending messages. As an example, suppose to have the following fragment of CJava code:

```
p.print(doc1);
p.print(doc2);
p.offline();
p.print(doc3);
```

  where `p` is an object belonging to the class `ConcurrentPrinter` described in Figure 9. Suppose that all these messages are saved into `p`'s LPM as they are issued because the corresponding preconditions are found to be false. When the first call to method `print` is issued, let us assume that `pendingBefore("print")` evaluates to 0 (this means that no other pending `print` messages precede the `print` message resulting from the first call in LPM). When the second call is issued, `pendingBefore("print")` evaluates to 1. When the third call is issued, `pendingBefore("print")` evaluates to 2. Similarly, `pendingBefore("offline")` for the third call to `print` evaluates to 1. Suppose now that the second call starts running. The corresponding message is removed from the LPM and, as a consequence, `pendingBefore("print")` for the third call evaluates to 1.

- Predicate `running("method name")` returns true or false depending whether the mentioned method belongs to LRM or not.

Figure 9 gives an example of how to use method preconditions in CJava. It shows the CJava implementation of the concurrent class `ConcurrentPrinter` introduced in Section 4.1. Methods `online` and `offline` are specified to run in mutual exclusion. As in the previous case of Figure 6, the method `print` may run in parallel with both `online` and `offline`. Moreover, only a single call to method `print` may be accepted and running at any time. The class definition uses the predicate `pendingBefore` to guarantee that the `print` messages sent to a `ConcurrentPrinter` object are processed in the same order in which they are received. The predicate `running` is used to implement the consistency constraints for the class `Printer.` It guarantees that methods `online` and `offline` run in mutual exclusion and that only a single instance of method `print` can be running at any time.

Like C++, Java allows method overloading. The same class may feature two or more methods with the same name provided that they have different signatures (that is, they differ in the number or type of their parameters). On the other hand, using the `pending` or `running` statements, the CJava programmer cannot distinguish between two different methods having the same name. This is not a undesired side effect of some implementation decision, but rather it is a deliberate choice made to enforce a good programming style. Expert programmers know that two methods with the same name should have the same "meaning" (i.e., they should perform the same task). It should not be relevant which of the two methods is running with respect to synchronization constraints This is

why we decided not to allow CJava programmers to distinguish among methods having the same name but different signatures in method preconditions.

As mentioned before, CJava supports single inheritance. A concurrent classes `ConcSon` may inherit from a different concurrent class `ConcParent`. As a result, `ConcSon` inherits all the methods (both preconditions and bodies) and attributes of `ConcParent` obeying to the standard visibility rules defined in Java.

The methods inherited from `ConcParent` may be redefined in `ConcSon`. One of the main features of CJava is that method preconditions and method bodies may be redefined separately within subclasses. The precondition of a method `M` of `ConcSon` may refer to the precondition of a method having the same name and defined in the superclass `ConcParent` of `ConcSon` by using the statement `super(arg1,...,argn)` where `arg1,...,argn` are values whose type matches the signature of `M`. . The precondition of a method `M` of `ConcSon` may also refer to the precondition of a method defined in the same class and having the same name but different signature by using the statement `this(arg1,...,argn)`. In this way, CJava programmers may reuse the same precondition for a set of methods that differ only in their signature but perform the same task, obeying to the same synchronization constraints. Finally, it is also possible to reuse a parent's method body in a subclass by invoking the method as follow:

`super.method_name(arg1,…,argn)`.

The example given in Figure 10 shows how the `super` and `this` statements together with the pseudo-variable `super` may be used to allow a separate redefinition of method bodies and preconditions. Figure 10 shows the CJava implementation of a class `EnhancedPrinter` that extends the class `Printer` by adding a method `print` with two parameters to print a document specifying if letter quality is needed. An enhanced printer is also able to know whether the toner is low or not. The two print methods (the new one and the one inherited from the superclass `Printer`) share the same precondition. In particular they may be called only if the constraints of the superclass `print` method are satisfied and if the toner is not low. In class `EnhancedPrinter`, the `print(Document)` method was redefined to add the `!tonerLow` expression to the precondition defined in the superclass and the method `print(Document,boolean)` having the same precondition of the redefined method `print(Document)` was added. It is important to observe that the body of the `print(Document)` method coincides with the one defined in the superclass (that is, the new body simply calls the superclass' method body).

As we will better explain in Section 4.3, the CJava ability of separately inheriting method preconditions and method bodies greatly reduces the occurrence of the inheritance anomaly.

Futures

Concurrent execution of methods is not confined to procedures (i.e., methods that do not return a value back to the caller). The invocation of a function (i.e., a method returning a non void value) is also asynchronous. It delivers a *future* for the result. A future is analogous to a bill of exchange. It is not the result the caller needs but it may be used as a substitute for the result. Future management and synchronization between the caller and the callee in order to exchange the result of the call when needed is managed by the CJava run-time system and is completely transparent to the programmer.

```
concurrent class EnhancedPrinter extends Printer {
  protected boolean tonerLow=false;
  public void print(Document d, boolean letterQuality)
    precondition this(d) {
    ...
  }
  public changeToner {
    tonerLow=false;
  }
  public void print(Document d)
    precondition super(d) && !tonerLow {
    super.print(d);
  }
}
```

Figure 10: Class `EnhancedPrinter` in CJava

Through futures, the caller and the called objects are synchronized using an approach similar to the wait-by-necessity approach [14] described in Section 3.1. In particular, the caller thread is synchronized with the thread of the callee (i.e., it has to wait) only when it attempts to use the result of a method that did not yet complete its execution[9]. As an example, suppose to have the following CJava program fragment:

```
i=cq.top();
System.out.println("I got a value from cq");
System.out.println("The integer I got is: "+i);
```

where `i` is an integer variable and `cq` is a reference to an object belonging to the class `ConcurrentQueue` defined in Figure 7. The first call to `println` may be executed without the need of waiting for `top` to complete. Conversely, before executing the second call to `println`, the caller needs to wait the execution of `top` to complete (the `println` statement, in fact, needs to use the result of the call to `top`). Variable `i` is what we called a "future".

4.3 CJava and the inheritance anomaly

This section describes how CJava overcomes the inheritance anomaly. In [10] several examples may be found that shows the different kinds of inheritance anomalies that affect the languages based on method preconditions. We will use these examples to test CJava ability in overcoming the inheritance anomaly. We will show that CJava allows an easy implementation of such examples solving the problems resulting from the inheritance anomaly.

In [10] the class `BBuf` is first described. Class `BBuf` implements a first-in first-out bounded buffer. It has two public methods, `put` and `get`. `put` stores an integer in the buffer and `get` removes the oldest integer. Two instance variables `in` and `out` count the total numbers of integers inserted and removed, respectively, and are used as indices into the buffer. The location where the next integer has to be inserted is indexed by (`in mod size`); the one from which the next integer has to be removed is indexed by (`out mod size`). Upon creation, the buffer is empty and the only acceptable message is `put`; `get` messages are not accepted i.e., they are kept unprocessed into the list of pending messages until the first integer is inserted. When a `put` message is processed, the buffer is no longer empty and can accept both `put` and `get` messages. When the buffer is full, it can only accept `get` messages. Moreover, only a single instance of method `get` may be running at any time, and the same holds for `put`. Figure 11a shows the CJava implementation of such bounded buffer.

[9] With "use a result" we mean trying to access the r-value of the variable that contains the result.

```
concurrent class BBuf {
  protected int size, in, out, buf[];
  public BBuf(int s) {
    size=s; in=0; out=0; buf=new int[size];
  }
  public void put(int elem)
    precondition !running("put") && in<out+size {
      buf[in % size]=elem;
      in++;
  }
  public int get()
    precondition !running("get") && in>out {
      int val=buf[out % size];
      out++;
      return val;
  }
}
```

a) Concurrent class `BBuffer`

```
concurrent class GBBuf extends BBuf {
  protected boolean afterPut;
  public GBBuf(int s) {
    super(s); afterPut=false;
  }
  public void put(int elem)
    precondition super(elem) {
      super.put(elem); // calls superclass method
      afterPut=true;
  }
  public int get()
    precondition super() && !running("gget") {
      afterPut=false;
      return super.get(); // calls superclass method
  }
  public int gget()
    precondition !running("gget") && !running("get")
                 && in>out && !afterPut {
      return get();
  }
}
```

b) Concurrent class `GBBuffer`

Figure 11: Concurrent classes `BBuffer` and `GBBuffer`

Suppose we need to define a subclass `GBBuf` of `BBuf` 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`. To implement class `GBBuf` using a language whose synchronization scheme is based on method preconditions, we need to define an extra instance variable `afterPut` and we need to redefine both `get` and `put` to update such variable. This is true for language CJava also, but the ability to separately inherit method preconditions and method bodies minimizes the redefinition effort. The reader may convince himself of this claim by looking at the CJava implementation of class `GBBuf` given in Figure 11b. Methods `get` and `put` had to be redefined but only the statements needed to update the new instance variable `afterPut` had to be added.

Most important, the CJava solution does not break object encapsulation. Making use of the Java pseudo variable `super`, in fact, the subclass method bodies may refer to the superclass ones without the need to access the instance variables defined in the superclass. The same holds for the method preconditions which may refer to the superclass ones by using the CJava statement `super(...)`.

The second example presented in [10] to show the inheritance anomaly in languages featuring method preconditions is the following: suppose we need to create a class `LBBuf` inheriting from `BBuf` and from `Lock`. `Lock` is a *mix-in* class [32] that exports the methods `lock` and `unlock`. They have the following semantics: an object, upon accepting the `lock` message will suspend the reception of further messages until it receives and accepts the `unlock` message. Direct instances of `Lock` are not created; rather, the purpose of class `Lock` is to be "mixed-into" other classes in order to add them the locking capability. To implement the `LBBuf` class using languages whose synchronization scheme is based on method preconditions, the preconditions of the methods inherited from class `BBuf` have to be redefined in order to account for the instance variable `locked`. This instance variable is inherited from class `Lock` and it indicates whether the object is currently locked or not.

Because CJava allows single inheritance only, it is not possible to implement the `Lock` example in CJava. On the other hand, if this was not the case (i.e., if CJava would allow multiple inheritance)

the CJava implementation of class `LBBuf` would require redefinition of all the `BBuf` methods as any other language whose synchronization scheme is based on method preconditions. On the other hand, as in the `GBBuf` example, the redefinition effort would be very limited thanks to the CJava ability to refer to superclass method preconditions and bodies in subclasses.

In conclusion it is possible to assert that CJava, by featuring the ability of referring to superclass method preconditions and bodies in subclasses, greatly reduces the occurrence and the seriousness of the inheritance anomaly. In fact this approach allows separate inheritance of method bodies and method preconditions and allows incremental implementation of concurrent objects.

4.4 Implementation

We took advantage of Java's support to multithreading to implement a prototype of a translator from CJava to Java. The actual version of such prototype implements all the CJava features described in this paper excluding futures, that actually are not supported.

To translate a CJava concurrent class into a Java class it is necessary to translate each concurrent method into a set of methods that have to:

- add the message corresponding to the called method to LPM,
- evaluate the method precondition,
- remove the message from LPM and add the corresponding method to LRM
- instantiate a new thread to execute the method body and,
- perform the operations described into the method body,

(not necessarily in this order).

Different strategies can be followed to perform the translation. In particular, one needs to decide who is responsible for evaluating methods' preconditions and who is responsible for instantiating the threads that execute the methods' bodies. The main alternatives are:

1. The caller evaluates the precondition of the called method, waits for the precondition to become true and then instantiates a new thread to execute the called method's body. This means that the caller and the callee synchronize at call time and remain synchronized until the precondition does not evaluate true. This approach is similar to the one adopted by Ada's rendezvous.
2. The caller instantiates a new thread that is responsible for waiting for the precondition to become true and for executing the called method's body when its precondition becomes true.
3. The caller simply appends the message corresponding to the called method to the object's LPM. A separate thread (which is part of the run-time system) evaluates method preconditions and instantiates new threads to execute the method bodies of the called methods whose preconditions are true.
4. As before, the caller simply appends the message corresponding to the called method to the LPM. A separate thread (which is part of the run-time system) instantiates a new thread for each received message. These new threads have the responsibility of waiting for methods' preconditions to become true and executing methods' bodies when their precondition becomes true.

If one of the last two strategies is chosen, one must decide if the separate thread that is described as part of the run-time system exists in a single instance system-wide, or there exist one instance for each concurrent object.

Alternative 1, which was adopted by B. Meyer in his description of the concurrent extension to Eiffel [33], was rejected because it reduces the level of concurrency. Moreover, from a theoretical point of view, it should not be the caller's responsibility to wait for preconditions to become true. Conversely, the alternative 2 increases the overall system concurrency. This is why it was adopted in the first version of our preprocessor.

The preliminary experimental evaluation of our preprocessor has shown that thread creation in Java is a time consuming operation. The caller spends time to instantiate the new threads that evaluate preconditions and execute bodies. To minimize the "message passing" time (i.e., the time spent by the caller to simply execute the call, independently from the time needed to accomplish the call) we abandoned alternative 2 in favor of alternatives 3 and 4. Alternative 4 was finally selected in order to increase the level of concurrency, as we discussed above.

We decided also to have one thread for each concurrent object, which periodically examines the LPM and instantiates a new thread for each message. This new thread, as mentioned before, has the responsibility of waiting for the method precondition to become true and of executing the method body.

5 Comparison and Conclusions

In this paper, we described the problem of inheritance anomaly that affects concurrent object-oriented languages (COOLs). After briefly sketching the problem, we surveyed several COOLs by classifying them with respect to the synchronization scheme they adopt in order to examine if they solve the inheritance anomaly. Such survey shows that the problem of inheritance anomaly has not been yet solved in the context of a true concurrent object-oriented language. Therefore, the remainder of the paper described the design and implementation of the language CJava, a true concurrent extension to Java. CJava provides a declarative approach based on method preconditions to describe the synchronization constraints that concurrent objects have to fulfill in order to avoid the occurrence of race conditions and to keep their internal state consistent even in the presence of parallel executing methods.

Two main features distinguish CJava from other COOLs: its ability of concurrently answering messages having the same object as target (i.e., true concurrency) and its ability of separately reusing the methods' implementation code and the synchronization code that each class inherits from its superclass.

Most of existing COOLs are quasi concurrent, that is, they do not support concurrency within a single object. Conversely, CJava is a true concurrent object-oriented language. Each CJava concurrent object may answer to several messages in parallel. This greatly increases parallelism in CJava applications.

The ability of separately reusing the implementation and the synchronization code of a method was showed to be fundamental to allow extensive code reuse and to avoid the need of breaking object's encapsulation. Most of the existing COOLs force synchronization code to be inherited together with method implementations. This reduces the ability of reusing code because, in most cases, the synchronization code needs to be rewritten in subclasses even if the method bodies would not [10]. We show that CJava solves this problem and eliminates the occurrence of the inheritance anomaly.

The current implementation of CJava was also described. It is based on a preprocessor that translates CJava programs in Java, together with a small number of classes that implement the

CJava run-time system. The current implementation focused on the expressive power of the language and on portability and compatibility with the standard Java run-time system. As a consequence, performance related issues were not taken into great consideration. Future work will be devoted at examining the feasibility of a more efficient implementation of the language based on a modification of the Java Virtual Machine in order to directly support concurrent objects.

Acknowledgements

I would like to thank Prof. Stefano Crespi Reghizzi who was my advisor for the work presented in this paper. He was always able to find time to discuss with me even if his agenda was really full. I would also thank Prof. Carlo Ghezzi for his valuable advice and for finding time to review all the version of this paper from the very first one.

Bibliography

[1] Object-Oriented Concurrent Programming. A. Yonezawa and M. Tokoro editors. MIT Press, 1987.

[2] G. Agha, *Concurrent Object-Oriented Programming*. Communications of the ACM, Vol. 33, No. 9, September 1990.

[3] Research Directions in Concurrent Object-Oriented Programming. G. Agha, P. Wegner, and A. Yonezawa editors. MIT Press, 1993.

[4] ISO/IEC, *International Standard ISO/IEC 8652:1995(E)*. International Organization for Standardization - International Electrotechnical Commission, 1995.

[5] J. Gosling, B. Joy, and G. Steele, The Java Language Specification. Addison-Wesley Inc., 1996.

[6] P. Niemeyer and J. Peck, Exploring Java. M. Loukides and P. Ferguson editors, O'Reilly & Associates Inc., 1996.

[7] A. Goldberg and D. Robson, Smalltalk-80: the Language and Its Implementation. Addison-Wesley, 1983.

[8] P. Wegner, *Dimensions of Object-Based Language Design*. In Proceedings of OOPSLA '90, pp. 168-182, ACM, Orlando, Florida, December 1987.

[9] M. Papathomas, *Concurrency Issues in Object-Oriented Programming Languages*. Object-Oriented Development Tech. Rep., Centre Universitaire Informatique, U. Geneva, D. Tsichritzis Ed., 1989.

[10] S. Matsuoka and A. Yonezawa, *Analysis of inheritance anomaly in object-oriented concurrent programming languages*. Research directions in Concurrent Object-Oriented Programming, G. Agha, A. Yonezawa, and P. Wegner editors, the MIT Press, 1993.

[11] S. Matsuoka, K. Taura and A. Yonezawa, Highly Efficient and Encapsulated Re-use of Synchronization Code Concurrent Object Oriented Languages. Proceedings of the OOPSLA'93 conference, Washington DC, September 1993.

[12] K.P. Löhr, *Concurrency Annotations for Reusable Software*. Communications of the ACM, Vol. 36, No. 9, September 1993, pp. 81-89

[13] Bertrand Meyer, *Eiffel: The Language*. Prentice Hall, second printing, 1992.

[14] D. Caromel, *Service, asynchrony and wait-by-necessity*. Journal of OO programming, Vol. 2, No. 4, November 1989.

[15] T. Nakajima, Y. Yokote, M. Tokoro, S. Ochiai and T. Nagamatsu, *DistributedConcurrentSmalltalk: a Language and System for the Interpersonal Environment*. ACM SIGPLAN Notices, Vol. 24, No. 4, April 1989, pp. 43-45.

[16] Y. Yokote and M. Tokoro, *The Design and Implementation of ConcurrentSmalltalk*. Proceedings of OOPSLA '86, pp.331-340.

[17] P. Mehrotra and J. Van Rosendale, *Concurrent Object Access in BLAZE 2*. ACM SIGPLAN Notices, Vol. 24, No. 4, April 1989, pp. 40-42

[18] S. Crespi Reghizzi, G. Galli de Paratesi, and S. Genolini, Definition of Reusable Software Components. Proceedings of the ECOOP'91 conference, Springer LNCS 512 (1991), pp. 148-166

[19] S. Crespi Reghizzi, M. De Michele, and S. Perotta, Behavioural Inheritance: Concepts, Ada Implementation and Experience. In proceedings of the TRIAda conference '92, Orlando, 1992.

[20] P. America, *Inheritance and Subtyping in a Parallel Object-Oriented Language*. Proceedings of the ECOOP '87 conference, Paris, Springer LNCS 276 (1997), pp. 234-242

[21] P. America, *A Parallel Object-Oriented Language with Inheritance and Subtyping*. Proceedings of the ECOOP/OPSLA '90 conference. SIGPLAN Notices, Vol. 25, No. 10, October 1990, pp. 161-168

[22] D. Caromel, *Toward a Method of Object-Oriented Concurrent Programming*. Communications of the ACM, Vol. 36, No. 9, September 1993, pp. 90-102

[23] M. Karaorman and J. Bruno, *Introducing Concurrency to a Sequential Language*. Communications of the ACM, Vol. 36, No. 9, September 1993, pp. 103-116

[24] P. A. Buhr, G. Ditchfield, R. A. Stroobosscher, B. M. Younger and C. R. Zarnke, *μC++: Concurrency in the Object-Oriented Language C++*. Software Practice and Experience, Vol. 20, No. 2, February 1992.

[25] D. G. Kafura and K. H. Lee, *Inheritance in Actor Based Concurrent Object-Oriented Languages*. . Proceedings of the ECOOP '89 conference, Cambridge Univ. Press, 1989, pp. 131-147.

[26] G. Agha and C. Hewitt, *Concurrent Programming Using Actors*. In Object-Oriented Concurrent Programming, A. Yonezawa and M. Tokoro editors, The MIT Press., 1987, pp. 37-53.

[27] H. Okamura and M. Tokoro, *ConcurrentSmalltalk-90*. Proceedings of TOOLS Pacific'90, Dec. 1990.

[28] O. M. Nierstrasz, Active Objects in Hybrid. Proceedings of the OOPSLA'87 conference. SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 243-253

[29] R. Trehan, N. Sawashima, A. Morishita, I. Tomoda, T. Imai and K. Maeda, *Concurrent Object Oriented 'C' (cooC)*. . ACM SIGPLAN Notices, Vol. 28, No 2, February 1993, pp. 45-52

[30] C. A. R. Hoare's, *Communicating Sequential Processes*. Communications of the ACM, Vol. 21, No. 8, August 1978, pp. 666-677.

[31] C. Ghezzi and M. Jazayeri, Programming Language Concepts, 3rd Edition, J. Wiley & Sons, 1997 (in press).

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

[33] B. Meyer, *Systematic concurrent object-oriented programming*. Communications of the ACM, Vol. 36, No. 9, September 1993, pp. 56-80.