JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, No. 1, May-June 2002

Design principles for highly reusable concurrent object-oriented systems

Emilio García-Roselló, José Ayude, J. Baltasar García Pérez-Schofield, Manuel Pérez-Cota
Computer Sciences Department of University of Vigo, Spain

Abstract

Designing a concurrent object oriented language isn't an easy task. After many years of research, the merging of concurrency and object oriented paradigms hasn't been achieved in a totally satisfactory way. Although recent models have partly solved important problems such as the inheritance anomaly, they do still present limitations due to the lack of reusability and adaptability. Approaches based on separation of concerns appear to point to the development of models which achieve effectively those requirements. In this essay we argue that the establishment of orthogonal design principles should be useful in this respect, as it was in other fields of object systems research. We'll propose these principles basing on well-known principles of design of programming languages and orthogonal persistence. In order to show the adequacy of these principles, we present CoJava, a model based on the separation of concurrent and functional aspects by means of their implementation in different component classes and their composition at runtime. CoJava has been designed applying our principles to offer a high degree of orthogonality, which results in better reusability than previous models.

1 INTRODUCTION

Concurrent programming is a powerful paradigm to build software that makes a more efficient use of hardware, and that can execute many activities concurrently. But concurrent programming isn't easy (Hoare, 1985; Andrews, 1991). Concurrent threads executing on the same resources could lead to undesired situations as deadlocks, livelocks, or data inconsistencies. These situations are more probable as the number of concurrent processes grows, thus making developers often limit potential concurrency in software design to avoid excessive complexity. Furthermore, classical concurrent languages have clearly separated abstractions for concurrency and data, thus limiting very much reusability of developped software (Lopes, 1997; Bergmans, 1994).

How to cite this article: E. García-Roselló et al: *Design principles for highly reusable concurrent object-oriented systems*, in Journal of Object Technology, vol. 1, no. 1, May-June 2002, pages 107-123, http://www.jot.fm/issues/issue_2002_05/article3

Object-oriented paradigm seems a priori especially well suited to solve these problems. In this paradigm a program is a collection of objects that represent physical or conceptual entities of the real world. Each one could be seen as an autonomous agent being able to handle potentially concurrent requests. Since objects are concurrent by nature, they make easier it to identify tasks that can be executed in parallel, thus simplifying concurrent software design (Agha, 1990). What is more, object-orientation allows a potentially high degree of reusability (Meyer, 1996).

During more than a decade of active research in this field, many concurrent object-oriented languages (COOL's) have been proposed[1]. But despite these advantages which theoretically should have made the merging of concurrent and object-oriented paradigms easy, this integration hasn't been satisfactorily implemented until now. Although recent models managed to partly avoid the problem of the inheritance anomaly (Matsuoka & Yonezawa, 1993), they do still present an important lack of reusability (Papathomas et al, 1997; Sánchez et al. 1998).

In this work we describe the approximation we've followed in our attempt to set some guidelines to design a COOL that overcomes these problems. To undertake this work, we decided to establish some principles of orthogonal concurrency as design principles to build object-oriented systems that integrate concurrency in a way that offers high reusability and adaptability. We'll show that thanks to these principles, we have been able to design a COOL, named CoJava, which effectively achieves the separation of the concerns that describe object functionality and concurrency, thus offering higher reusability than previous proposed COOL's. We have also verified that considering concurrency as a monolithic concern hinders a good design. Therefore, we'll show that orthogonal concurrency principles should be complemented with an additional principle of separation of concurrency concerns.

The remainder of this article is structured as follows: first, the principles of orthogonality are presented and then discussed. Finally, CoJava is presented as an example of the application of these principles, and theoretical conclusions about experiences of this application are discussed.

[1] Surveys can be found in Philippsen (2000), Papathomas (1995), or Holmes (1999) among others.

2 A PROPOSAL OF ORTHOGONAL CONCURRENCY PRINCIPLES

In programming languages design the fact is well accepted that expressive power and reusability of a language is directly related with separation of the concepts it includes and with availability of powerful composition rules to combine them. Several authors have pointed out this general orthogonality principle in different aspects of language design, like Strachey (Strachey, 1967) and later Tennent (Tennent, 1977) who formulated the design general principles of correspondence, abstraction and data type completeness. This can also be seen in works on concurrent (Wegner, 1987; Nierstrasz 1993; Papathomas, 1995) or persistent (Atkinson & Morrison, 1995) object-oriented languages design.

The latter is an especially outstanding example. In order to integrate persistence in such a way that the resulting system would offer high software consistency, productivity, reusability and adaptability, orthogonal design of this aspect was investigated and some principles were defined, which specify the features a system should show to be considered orthogonally persistent (Atkinson & Morrison, 1995). The benefits of the definition of these principles in the persistence field are evident, since they have established a universally accepted framework to construct orthogonal and highly reusable persistent object-oriented systems.

But curiously, a clear attempt to do the same in object-oriented concurrency research has not been taken. This is even more surprising when we consider that orthogonality of concurrent and object-oriented mechanisms has been clearly identified as a requisite to achieve reusability (Wegner, 1987; Nierstrasz, 1993; Papathomas, 1995). Also, it is easy to establish a parallelism between persistence and concurrency, as both are widely considered as non-functional aspects, thus separable from functionality (Hürsch & Lopes, 1995; Kiczales et al, 1997). Its no-separation leads to tangling between code that implements functionality and that one which implements these other aspects into the class, thus seriously hindering its reusability. The problem has been coined as inheritance anomaly (Matsuoka & Yonezawa, 1993). In fact, several COOL's have been implemented on the basis of separating concerns, like Dragoon (Atkinson et al, 1992), Composition Filters (Bergmans, 1994) or D-language (Lopes, 1997) recognizing the convenience of considering concurrency widely orthogonal to functionality. But it isn't until very recently that this similarity between persistence and concurrency has been considered – aiming at studying if solutions from one field can be applied to the other (García Roselló et al, 2001; Ayude, 2001).

Basing on the arguments and works previously described, we propose the definition of orthogonal concurrency principles which serve as unified guidelines to design highly reusable concurrent object-oriented languages. These principles are as follows:

- **Principle of concurrency independence**: the form of a program is independent of the concurrent behaviour of objects it manipulates.

- **Principle of data type orthogonality**: all objects should be allowed to have any permitted concurrent behaviour, irrespective of their type.
- **Principle of concurrency identification**: the identification of the concurrent behaviour of an object is orthogonal to the universe of discourse of the system, and therefore it can't be related to the type system.

In short, to approach to orthogonal concurrency, a system should support the programmer to describe concurrency with minimal information, and this information has to be clearly separated from functional code, as happens in orthogonally-persistent systems (Atkinson & Morrison, 1995).

An object system that shows all these principles is considered orthogonally concurrent. It's clear that any system that adheres to these principles would present a high degree of reusability and expressive power. It's also clear that if a model fails in one of these principles, it will almost certainly fail in some way in all the others, since they are closely related. Next we are describing each one of these principles and their implications in COOL's design.

Principle of concurrency independence

This principle states that a program has to look the same independently of concurrency. That implies orthogonal management of all objects whether its concurrent behaviour. It's certainly the most important of the principles, since it can clearly improve reusability, while it's also the most difficult to achieve. In fact, this principle claims for total transparency of concurrency in a system. Several COOL's have partly approached to this aim. For example ABCL/1 (Yonezawa et al. 1987) or Sina (Bergmans, 1994) have an homogeneous model of active objects. Java// (Caromel et al, 1998) and Mentat (Grimshaw, 1993) implement a transparent asynchronous call mechanism. But it's clear that, unlike persistence or distribution, where quite total transparency could be achieved, concurrency require some explicit code, particularly in defining synchronization policies. These policies cannot always be infered from the need of data integrity (i.e. a policy that priorizes some method calls above others). That implies that no COOL can have a totally transparent syncronization mechanism.

Homogeneous models of active objects, even if clearly more orthogonal, have the problem of being very inefficient. That's why many COOL's have been designed basing of an heterogenous model, allowing the programmer to determine when an object has an active or passive behaviour. Efficiency is an argument frequently used to not implement full orthogonality, for example in persistent systems (i.e. Cooper & Wise, 1996), and it's clear that in this case it's even more justified.

As it could be seen, this principle has some clear limitations in its practical application. However we don't see that as a lack of validity, because these principles are abstract design guidelines that shouldn't be influenced by practical limitations. It will be the responsibility of system designers to achieve an agreement between them and other considerations as efficiency requirements, hardware availability, etc.

Principle of data type orthogonality

The principle of data type orthogonality should be understood as claiming for all objects as first-class objects with regard to concurrency.That is, all objects should be allowed to have all range of supported concurrent behaviours, irrespective of their type. The relevance of this principle in achieving more reusability is clear, as it allows a simple adaptation of any functionality to changing concurrent needs.

For example, the most of the COOL's allow to define an active class from any other class. That can be made through multiple inheritance as in Eiffel// (Caromel, 1990), or simply using a class modifier as in sC++ (Petitpierre, 1998). But many COOL's aren't so orthogonal with regard to synchronization. That's particularly true for that one based on an heterogeneous model. In those COOL's the most habitual approach is support only the combinations active+synchronized and passive+unsynchronized objects. Moreover, once a class has been characterized as active, it isn't possible to define a passive one basing on it. These are clear violations of data type orthogonality principle.

COOL's having an homogenous model don't have this problem, as all objects are active and synchronized.

Principle of concurrency identification

This principle states that identification of the concurrent behaviour of an object has to be orthogonal to the universe of discourse of the system. Therefore this implies that concurrent behaviour can't be related to the type system. The reason of this principle is to avoid interference between concurrent and functional features and thus improve reusability. If type system and concurrency aren't largely independent in a model, it's quite sure that its reusability would be weakened. For example, active or passive versions of the same functionality couldn't be of the same type, thus not totally substitutable in a method call, forcing to write redundant code.

In object-oriented models, that means for example that inheritance couldn't be used as mechanism to provide concurrency to an object. Several existing models violate this rule. For example, in some models an object must inherit from a special class to be active or synchronized as in Eiffel// (Caromel, 1990) or the proposal of Karaorman & Bruno (Karaorman & Bruno, 1993). This have a reuse problem: active classes must have to be defined as subtypes of passive ones in order to reuse defined classes, without any semantical reason which justifies it. Other models use a language keyword as a kind of class modifier to define it as active, as in Mentat (Grimshaw, 1993) or Charm++ (Kale & Krishnan, 1993). Although inheritance is not directly used, the same problem arises because this mechanism prevents from freely substituting active and passive versions of a class.

The same problem exists with synchronization. Any COOL we know offers totally orthogonal identification as synchronization is implemented into class, and thus considered part of the class type. Proposals as Composition Filters (Bergmans, 1994) or D-Language (Lopes, 1997) have the advantage of being based on mechanisms orthogonal

to object-oriented features, therefore they show very high orthogonal identification of synchronization. But it's not complete, since in both of them we have to define different subtypes in order to combine the same functionality with different synchronizations.

Orthogonality of concurrent concerns

Initially we considered that these principles established a framework useful and thorough enough to help in the design of highly reusable COOL's. But when used to design our concurrent language CoJava, soon it was clear that concurrency was reluctant to fit to these principles when applied to it as a monolithic concern. That can also be deducted from the previous exposition of each principle. For example, it's relatively easy to orthogonalize interaction between objects by means of mechanisms like wait-by-necessity that make asynchronous calls transparent to the language. The same is true for activity, as demonstrate homogeneous models where functionality is orthogonal to active behaviour of objects, but in this case practical issues make quite mandatory implementing languages with an heterogeneous model. Synchronization is, in turn, impossible to completely orthogonalize, as it can require explicit code, but it's at least possible to implement it in such a way that fulfils data type orthogonality and identification principles. Also, we have to pay attention to models that combine several concerns, as for example those that only allow active objects to be synchronized.

This distinction between several concurrent aspects, that is interaction, activity and synchronization, isn't new. Papathomas (1995) or Kafura&Lavender (1993) already make it in their taxonomy of COOL's, distinguishing between an animation model, an interactive model and a synchronization model. Even in classical non object-oriented concurrent systems we can distinguish between different mechanisms for these three concerns. Orthogonality of these concerns in the sense of Wegner (1987) can be argued, as there are examples of languages that implement some of them but not the other ones. But until now there isn't any work about the convenience of separating these concerns in COOL's design, since attention has been centered on separating synchronization from functionality.

Therefore, we argue that is important to establish a principle of separation of concurrent concerns as follows:

- **Principle of orthogonality of concurrent concerns**: concurrency can be considered as formed by the orthogonal concerns of activity, interaction and synchronization.

Therefore, in the design of a concurrent system, these concerns should be keep in mind as separated aspects, and the principles of orthogonal concurrency previously stated could (and should) be applied to each one. This should lead to define separate and orthogonal mechanisms implementing each concern, which clearly provides higher reusability.

It could be argued that even these concerns aren't totally monolithic. For example, synchronization has been divided in several subconcerns by some authors (i.e. Holmes, 1999; Bader & Elrad, 1998). Of course, that's absolutely legitimate and perfectly

combinable with our principles. Orthogonalization can be applied recursively, but, as argues Morrison (1979) some of the consequences of following this course can lead to complications. At each decomposition stage the designer has to make a decision. The orthogonalization can either be ignored on some other constraint, for example implementation efficiency, or else he has to live with the consequences. Anyway, the process must end at some point, commonly when a balance between advantages and limitations has been reached.

Basing on the existing work on COOL's, we argue that the orthogonalization level we propose here is the more widely agreed, thus offering a good mixing of reusability, expresivity and abstraction for most general-purpose languages. If the designer considers convenient to decompose some of the proposed concurrent concerns he can do it. The proposed principles of orthogonal concurrency remain valid and appliable to each considered concern.

For example, as it'll be explained in the next section, when designing our COOL CoJava, we decided to decompose synchronization concern in two orthogonal subconcerns in order to achieve reusability of synchronization policies.

3 DESIGNING AN ORTHOGONALLY CONCURRENT SYSTEM: COJAVA

To verify the usefulness of the proposed principles, we used them in the design of a COOL called CoJava. Therefore, CoJava has these features:

1. The following aspects are taken as orthogonal in the language: functionality, activity, interaction and synchronization. Furthermore, synchronization is considered as a composite concern of both an implementation-depent and an implementation-independent aspects (Bader & Eldar, 1998; holmes, 1999). This point will be further explained in more detail.
2. The language supports a separate description of the different aspects of an object at the conceptual level, and their composition at runtime.
3. Active and passive objects are allowed, as well as intraobject concurrency. Any type of object is allowed to be active or passive, in a per-instance basis.
4. Synchronization policies are generic, reusable, and can be combined with any type of object, without restriction.
5. Interaction between objects is transparently managed by means of a wait-by-necessity mechanism.
6. Only functionality of an object determines its type.

Feature (1) is based on the principle of orthogonality of concurrent concerns. Features (2), (3), (4) and (6) allows to fulfill data type orthogonality as well as concurrency identification principles, as it'll be detailed later. Feature (5) points to independency principle applied to the interaction concern. Obviously, feature (3) prevents from totally

fulfil independency principle for the activity aspect, as CoJava follows an heterogeneous model. As it has been previously explained, this can be justified by efficiency reasons.

Therefore, CoJava has a high compliance with the design principles previously explained. In fact, as far as we know, it fulfils these principles better than any other COOL with the exception of the activity aspect, more orthogonal in homogeneous models. In the next sections we'll show how this design has been implemented in a language, describing CoJava' syntax. We'll also show how it offers a high reusability thanks to its orthogonal design.

The CoJava language

CoJava has been implemented as an extension of Java (Gosling, 1996), extending it in two ways: (1) it provides a sublanguage that allows description of synchronization policies, based on synchronization and mapping classes, and (2) a sublanguage for creating objects by means of compositions of the behavioural aspects defined in the language.

Separation of concerns in CoJava is supported by the existence of three separate kinds of classes, each one with its corresponding base class, which allow to define the semantics of each one of the aspects considered as independent in the model:

- FunClass classes, derived from Object base class, are functional classes, which correspond to the classes in a classical class-based language such as Java or C++.
- SyncClass classes, derived from SyncClass base class, are classes where we define synchronization restrictions. As we will show later, another kind of class called MapClass exists to compose synchronization with functionality.
- WrapperClass classes, derived from WrapperClass base class, which define an active/passive behaviour.

Every object in CoJava is composed of a functional component, a wrapper component, and an ordered set of synchronization components. This shortlist is specified when the object is created. For example, the following code would create an object with a functional component instantiated from the Buffer class, the active behaviour corresponding to Active wrapper class, and the synchronization defined in BufferSync class:

```
Buffer A;
A= new Buffer() with wrapper Active synchronized by
{BufferSync};
```

It can be seen that it is the functional class that determines the type of the object. Therefore in CoJava the objects with the same functional type are manipulated in the same way, and totally interchangeable as parameters in a method call, independently of their active/passive behaviour or synchronization.

The activity component or wrapper can not accede directly the functionality of the object. Therefore, it's generic and composable with any functional and synchronization classes.

The synchronization in CoJava has been considered as two different concerns. One of them deals with defining necessary mutual exclusions between methods with regard to their concurrent execution. From now, we'll better call them restrictions, to avoid misunderstandings. These restrictions are implementation-dependent, as they derive from a concrete implementation of the class. A change in this implementation can lead to changes in these restrictions, even if functionality, that is, class interface, has remained unchanged. Therefore, we considered that these restrictions have to be defined as part of functional class.

The other aspect of synchronization is implementation-independent. For example, a priorities-based policy doesn't depend of any particular implementation of a functional class. In fact, such a synchronization policy could be applied to any functional class. Furthermore, if we consider the data type orthogonality principe, it should be potentially generic, to be applied to any data type, and thus reusable. But the synchronization component deals with the problem of having to refer to methods of the functional class. So, in order to support its genericity, we have chosen to create an intermediate mapping class which acts as a connector between synchronization an functional classes.

Thus, in CoJava, SyncClass are abstract classes defining synchronization. In a SyncClass we can define abstract states and abstract methods. An abstract method could have some preconditions to its execution, normally based on some of the defined abstract states, as well as preactions if necessary. Subclasses of a SyncClass may be defined, thereby creating synchronization SyncClass hierarchies. However, as they are abstract classes, SyncClass may not be directly instantiated neither included as object's component. For this purpose we have to use MapClass classes. MapClass are in charge of the establishment of a correspondence between the abstract states and methods of a synchronization class and concrete methods of the functional class. This separation of synchronization and functionality is somewhat similar to the proposed one in Dragoon (Atkinson et al, 1992), but with important differences. In CoJava it is permitted to construct synchronization class hierarchies, which in turn is totally forbidden in Dragoon. Also, in Dragoon mapping between abstract and concrete methods is carried out inside the functional class, but in CoJava we use a separated class, the MapClass, which can be composed at runtime with a functional class. This prevents programmers from having to define a subclass of a functional class in order to include the required synchronization for a specific situation. As we have been discussing, this solution based on subclassing derives from a lack of orthogonality and a loss of reusability, as appends in Dragoon, Composition Filters or D-Language, for example. CoJava avoids this problem thanks to separating mapping in different kind of classes. In this way, it provides total polymorphism to the synchronization class, as it may be composed with different functionalities by simply defining the adequate mapping class.

SyncClass base class has two data members, which are CurrentCall and Queue. This allows to accede to the data of the current call and the waiting calls queue for the object. This MetaObject Protocol allows us to define relatively complex synchronization policies (Caromel, 1990; Caromel et al, 1998).

A simple example in CoJava

In order to illustrate the main features of CoJava, we will now present a typical example, consisting on the implementation of a bounded buffer which is acceded concurrently by a producer and a consumer.

We define the functional class Buffer which implements a buffer based on an array of limited size, and the generic synchronization class BoundedContainer for a bounded container. The BufferSync class maps the generic synchronization class over the functional class:

```
FuncClass Buffer{
  Incompatible methods{
    put={put, get};
  }
  private Integer[] buffer=new Object [4];
  private Integer P=new Integer(0);

  public void put(Object X){
    buffer[P]=X;
    P++;
  }

  public Object get(){
    P--;
    return buffer[P];
  }

  public Boolean empty(){
    return (P==0);
  }

  public Boolean full(){
    return (P==buffer.length);
  }

  public Integer size(){
    return P;
  }
}
```

```
SyncClass BoundedContainer{
  abstract states={empty, full};
  abstract methods={push, pop};

  preconditions{
    (!full) for push;
    (!empty) for pop;
  }
}


MapClass BufferSync maps BoundedContainer over Buffer{
 states map{
   full=full();
   empty=empty();
 }

 methods map{
   push=put();
   pop=get();
 }
}
```

A very simple program which implements the use of this buffer as a passive object with the above mentioned synchronization by a producer and a consumer could look like this:

```
Buffer B= new Buffer() with wrapper Passive synchronized by
{BufferSync};
Consumer C= new Consumer () with wrapper Active synchronized
by {}; //void synchronization
Producer P= new Producer () with wrapper Active synchronized
by {};/ /void synchronization

while (true){
  B.put(P.getObject());
  C.putObject(B.get());
}
```

Note that despite the appearance of "sequentiality" of the code, concurrency is achieved because the calls to the active objects are asynchronous, and so both the producer and the consumer will only stop if the buffer gets filled up or empty respectively, or due to the wait-by-necessity mechanism provided that they obtain a reference to an object which may be not yet available. A sequential version of the program might be obtained by simply turning the active wrapper of either the producer or the consumer into a passive one, albeit in this case it would not be logical to use a buffer to communicate them, as it would never contain more than one element.

The classical example of the addition of a get2() method to the Buffer class, which in our case simply removes two values from the buffer and returns the second one, could be implemented in CoJava by extending the previously defined classes:

```
FuncClass Buffer2 extends Buffer{
  Incompatible methods{
    get2={put,get,get2};
  }

  public Object get2(){
    P=P-2;
    return buffer[P];
  }

}

SyncClass BoundedContainer2 extends BoundedContainer{
  abstract states={half};
  abstract methods={pop2};
  preconditions{
    (half) for pop2;
  }
}

MapClass BufferSync2 extends BufferSync{
 states map{
    half=(size()>=2);
 }
 methods map{
    pop2=get2();
 }
}
```

Note in these examples the clear-cut separation of concerns and the high degree of reusability of synchronization code obtained in CoJava, due to its genericity and extensibility. Mapping classes prevent the inclusion of synchronization code of any kind in the functional class, or code that depends on functionality in the synchronization class, promoting the independence of both components.

The use of the MOP allows us to implement policies which are more difficult to convey by just bearing in mind the object's states. For example, let us suppose a container object with read and write methods. Due to its implementation, readings can be performed concurrently, but writings require an exclusive access to the object. A naïve synchronization could lead to an inanition of the calls to write method if there is a continuous flow of read calls. The implementation of a FIFO policy would avoid this

problem, but it cannot be defined with abstract states. On the other hand, it's easy to implement it using both the reified call and the waiting queue of the object as variables:

```
SyncClass FIFO {
  abstract methods={anymethod};

  boolean accept(){
    return (CurrentCall=Queue.oldestCall());
  }

  preconditions{
    (accept) for anymethod;
  }
}
```

As we can see, the synchronization policy defined here is quite simple but expressive, and in this case does not require any abstract states. We must also point out that the fact of some methods requiring exclusive access is not reflected in the synchronization class, as it depends exclusively on the implementation of functionality, which takes charge of determining this concern. Thus, this same synchronization class would be valid in order to impose a FIFO policy on any functional class regardless of its restrictions derived from a particular implementation.

Implementation of CoJava runtime

For this first implementation of CoJava runtime, we extensively based on reflective facilities of Java language. The use of metalevel programming and reflection for implementing the separation and integration of concerns has been widespread (Aksit et al, 1996; Hürsch & Lopes, 1995; Kiczales et al, 1997). In CoJava, the creation of an object at the base level implies the creation of three objects at the metalevel, each one corresponding to an object component, which are communicated by means of a MetaObject Protocol (MOP). Access to each object comes through a proxy defined on the metalevel, which reifies the calls to object's methods. Each reified call is transferred to the wrapper object of the metalevel. The wrapper uses the information contained in the metaobject corresponding to the synchronization component in order to decide when a call can be served. As previously noted, the MOP is accessible in a limited way to the programmer by means of the instance variables CurrentCall and Queue of the synchronization classes, which allow access to the reified call being evaluated as well as to the queue of reified waiting calls.

4 CONCLUSIONS

In this paper, we have proposed the definition of some orthogonal concurrency principles as a way to establish a useful framework for the design of object-oriented concurrency

models which offer a high degree of reusability and avoid problems like inheritance anomaly. With this aim, we have proposed four principles, based on well-known principles and works of programming languages design, orthogonal persistence and separation of concurrent concerns.

We have presented the CoJava model, whose design is based on the proposed principles. Therefore CoJava shows high concurrency independency, total data type orthogonality and a mechanism of concurrency identification orthogonal to the type system. Thanks to this, CoJava achieves higher reusability than previous COOL's proposals, avoiding completely inheritance anomaly. Therefore, we think that usefulness of our principles as helpful design guidelines for COOL's has been shown. We must also point out that CoJava model fulfils the adaptability requirements for COOL's defined by Sánchez et al. (1998).

5 RELATED WORK

Object-oriented concurrency has been a fertile field of research since the late 80's. A good survey can be found in Philippsen (2000). Some authors have suggested principles or guidelines for COOL's design. For example, Caromel (1990), Papathomas (1992), Bergmans (1994), Meyer (1996) or Lopes (1997). But those works focused very much on pragmatic decisions, more than on well-established design principles. For example, among other considerations, Papathomas argues that a COOL must be able to implement the administrator pattern (Gentleman, 1981). Caromel and Meyer defend a sequential object model in order to simplify design and better reuse code from not-concurrent applications. Both Bergmans and Lopes base their proposals on the separation of concerns principle, but they don't clearly specify how to apply it in the language design, and anyway they also include as design principles an efficient implementation and the convenience of extending an existing language.

However, it seems clear that even if the principles we have proposed in this work aren't explicitly defined until now, the majority of the proposed models that attempt to join concurrent and object-oriented paradigms have commonly used mechanisms that in fact improve concurrency orthogonality in some way, in order to reach a higher degree of reusability, simplicity and transparency to the user. In fact, inheritance anomaly can be easily seen as a lack of orthogonality between concurrency and inheritance mechanisms (Nierstrasz, 1993), and in the same way proposed solutions would try to reach higher orthogonality. For example mechanisms like *wait-by-necessity* (Caromel, 1990) or homogeneous models (i.e. Yonezawa et al, 1987) contribute to higher orthogonality making synchronization of asynchronous calls transparent to programmer. Furthermore, the separation of concerns paradigm of software engineering, which was successfully used in some of the most recently proposed COOL's (Bergmans, 1994; Lopes, 1997; Bader & Elrad, 1998; Holmes, 1999) is a concept strongly related to orthogonality: it involves the identification of orthogonal components, and the definition of mechanisms to support its description separately and its later composition (Hürsh & Lopes, 1995).

REFERENCES

[Agh90] Agha G.: *Concurrent Object-oriented programming*, Communications of the ACM, 33:125-141.

[Aksi96] Aksit M., Tekinerdogan B., Bergmans L.: *Achieving adaptability through separation and composition of concerns.* In *Special Issues in Object oriented programming*, Mühlhaüser M. (ed.), Workshop reader of the ECOOP'96, Linz, Austria.

[Andr91] Andrews G. R.: *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing.

[Atki92] Atkinson C., Crespi-Reghizzi S., Di Maio A., Goldsack S.: *Behavioural Inheritance: themes and variations*. In *Workshop on object-based concurrency and reuse*, ECOOP'92, Uthrecht, 1992.

[Atki95] Atkinson M., Morrison R.: *Orthogonally Persistent Object Systems. VLDB Journal*, 4:319-401.

[Ayud01] Ayude J.: *Ortogonalidad de la Concurrencia en Modelos de Concurrencia Orientados a Objetos*. Ph.D. Thesis, Department of Computer Sciences, University of Vigo.

[Bade98] Bader A., Elrad T.: Adaptative Arena. Language constructs and architectural abstractions for concurrent object-oriented systems. In Proceedings of the Internatoinal Conference on Parallel and Distributed Systems - ICPADS 1998. IEEE Comp. Soc., Los Alamitos, CA, USA,. p 599-606.

[Berg94] Bergmans L. M. J.: Composing Concurrent Objects: Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs. Ph.D. dissertation, University of Twente, Netherlands.

[Caro90] Caromel D.: Programming Abstractions for Concurrent Programming. In Proceedings of the 2nd Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'90), pp245-253.

[Caro98] Caromel D., Klauser W.,Vayssière J.: Towards Seamless Computing and Metacomputing in Java. Concurrency Practice and Experience, 10:1043-1061.

[Coop96] Cooper T.B., Wise M.: Critique of Orthogonal Persistence. In Proceedings of International Workshop on Object Oriented Operating Systems, IWOOOS'96.

[Garc01] García Roselló E., Ayude Vázquez J., García Perez-Schofield B., Pérez Cota M.: Using orthogonal concurrency principles to effectively separate concerns in COOL's design. In Proceedings of the 8th IEEE Congreso Internacional de Investigación en Ciencias Computacionales. November 28-30, 2001 — Colima, Mexico

[Gent81] Gentleman W.M.: Message passing between sequential processes: the reply primitive and the administrator concept. Software-practice and experience 11:435-466.

[Gosl96] Gosling J.: *The Java language specifications*. Addison-Wesley, Massachussets.

[Grim93] Grimshaw A.S.: Easy to use object-oriented parallel programming. IEEE Computer, 26:39-51.

[Hoar85] Hoare C. A. R.: *Communicating Sequential Processes*. Prentice Hall.

[Holm99] Holmes D.: *Synchronisation Rings: ComposableSynchronisation for Object-Oriented Systems*. Ph.D. dissertation Macquarie University, Sydney, Australia.

[Hürs95] Hürsch W.L., Lopes C.V.: *Separation of concerns*. *Technical report NU-CCS-95-03*, Northeastern University, Boston, USA.

[Kafu93] Kafura D. G., Lavender R. G.: *Concurrent Object-Oriented Languages and the Inheritance Anomaly*. In *Parallel Computers: Theory and Practice*, T. L. Casvant, P. Tvrdik, and F. Plasil (eds), IEEE Press.

[Kale93] Kale L.V., Krishnan S.: *Charm++: a portable concurrent object oriented system based on C++*. In *Proceedings of OOPSLA'93*, 8th Annual Conference on Object Oriented Programming Systems, Languages and Applications, pp:91-109, Washington, Sep 28- Oct 1. ACM SIGPLAN Notices 28(10).

[Kara93] Karaorman M.: Introducing concurrency to a sequential language. Communications of the ACM. 37: 103-116.

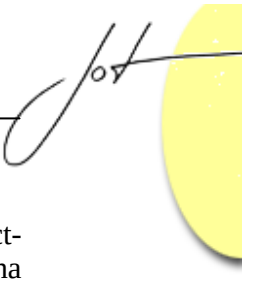
[Kicz97] Kiczales G., Lamping J., Mendhekar A., Maeda C., Videira Lopes C.V., Loingtier J.M., Irwin J.: *Aspect-Oriented Programming*. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, Finland.

[Lope97] Lopes C.V.: *D: A language framework for distributed programming*. Ph.D. dissertation, College of Computer Science of Northeastern University, USA.

[Mats93] Matsuoka S., Yonezawa A.: Inheritance anomaly in object-oriented concurrent programming languages. In Research directions in concurrent object-oriented programming languages. Agha G., Wegner P., Yonezawa A. (eds.) MIT press.

[Meye96] Meyer B.: Object oriented software construction, 2º ed. Prentice-Hall.

[Morr79] Morrison, R.: *On the development of algol*. Ph.D. Dissertation. Department of Computational Science, University of St Andrews, UK.



[Nier93] Nierstrasz O.: Composing active objects. The next 700 concurrent object-oriented languages. In Research directions in Object-based concurrency,Agha G., Wegner P., Yonezawa A. (eds), MIT Press.

[Papa95] Papathomas M.: *Concurrency in object-oriented programming languages.* In *Object oriented software composition*, Nierstrasz O. & Tsichritzis D. (eds), Prentice-Hall.

[Papa97] Papathomas M., Hernández J., Murillo J.M., Sánchez F.: *Inheritance and Expressive Power in Concurrent Object-Oriented Languages.* In *Proceedings of Langages et Modèles à Objets '97*, Ducournau R. & and Garlatti S. (Eds.), Hermes. Roscoff (France).

[Peti98] Petitpierre C.: *Synchronous C++: A language for interactive applications*. IEEE computer, Sep. 1998 pp:65-72.

[Phil00] Philippsen, M.: A survey of concurrent object-oriented languages. Concurrency: practice and experience 12:917-980.

[Sánc98] Sánchez F., Hernández J., Murillo J.M., Pedraza. E.: *Run-time adaptability of synchronization constraints in COOLs*. Paper presented in *II Workshop on Aspect Oriented Programming*, ECOOP'98, Bruselles, Belgium.

[Stra67] Strachey, C.: *Fundamental Concepts in Programming Languages.* Oxford University Press, Oxford.

[Tenn77] Tennent, R.D.: *Language Design Methods Based on Semantic Principles*. Acta Informatica 8:97-112.

[Wegn87] Wegner P.: *Dimensions of object-based language design*. In *Proceedings of OOPSLA'87*, Orlando (USA), ACM SIGPLAN Notices, 22:168-182.

[Yone87] Yonezawa A., Shibayama E., Takada T. and Honda Y.: *Modelling and Programming in an Object-Oriented Concurrent Language – ABCL/1*. In *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro (eds.), MIT Press, Cambridge.

About the authors

Emilio García-Roselló is associate lecturer at the Department of Informatics of the University of Vigo. His research work, together with the other authors, is mainly centered on reusability and component-oriented software engineering. He can be reached at erosello@uvigo.es .

José Ayude is associate lecturer at the Department of Informatics of the University of Vigo. He recently got his Ph.D. with a thesis on COOL's reusability concerns. He can be reached at jayude@uvigo.es .

J. Baltasar García Pérez-Schofield is associate lecturer at the Department of Informatics of the University of Vigo. His research work is mainly centered on object persistence. He's currently working on the schema evolution support for the persistent environment Barbados. He can be reached at jbgarcia@uvigo.es .

Manuel Pérez-Cota is chair professor at the Department of Informatics of the University of Vigo. He heads a research group on object-oriented software engineering and he has many publications about this issue. He can be reached at mpcota@uvigo.es .