Integration of Concurrency Control in a Language with Subtyping and Subclassing

Carlos Baquero, Rui Oliveira, Francisco Moura
Departamento de Informatica
Universidade do Minho, Braga, Portugal

Integration of Concurrency Control in a Language with Subtyping and Subclassing

Carlos Baquero* Rui Oliveira* Francisco Moura*

Departamento de Informática / INESC
Universidade do Minho
Braga - Portugal

Abstract

This paper describes the integration of concurrency control in BALLOON, an object-oriented language that separates the concepts of type and class as well of subtyping and subclassing. Types are interface specifications enriched with concurrency control annotations. Classes are used to implement the operational functionality of types as well as concurrency control mechanisms. Types, classes and concurrency control annotations are independently reusable and derivable. The language takes advantage of this separation to solve the typical problems of the inheritance anomaly.

1 Introduction

BALLOON is a novel object-oriented language intended as a test-bed for experimenting with concurrency in a object-oriented, potentially distributed setting. It is a compiled, statically-typed language that allows explicit creation and management of concurrency while avoiding the traditional conflicts between inheritance and concurrency.

The management of large-scale software projects calls for separate compilation and joint development through code reuse and incremental development. These features are strongly related to the notions of encapsulation, inheritance and component interface description which are the foundations of object-oriented technology. Therefore, BALLOON not only advocates the separation of subtyping and subclassing but also distinguishes concurrency control classes from operational classes. As a result, types, classes, and concurrency control are independently reusable and derivable, thereby eliminating the inheritance anomaly.

BALLOON is expected to contribute to the design of extensible libraries of concurrent components. In fact, BALLOON allows the derivation of concurrent components without requiring the knowledge of any internal details about its ancestors, as these may well be hidden in a compiled library. It suffices to know the type interface and develop a properly annotated class.

In the next section we briefly overview existing approaches to concurrency in object orientation focusing on the inheritance anomaly. Subsequent sections describe the model of types and classes in BALLOON with emphasis in the language constructs relevant for the following sections. Next, the extensions for concurrency control are introduced showing how they interact with the type and class structures of the language and how they cope with the inheritance anomaly. This is then compared with related work and finally the paper's contributions are summarized.

2 Concurrency in Object Orientation

The introduction of concurrency in the object-oriented model has been intensively investigated in recent years, where numerous alternatives and pers-

*Email: {cbm,rco,fsm}@di.uminho.pt. WWW: http://www.di.uminho.pt/ ~{cbm,rco,fsm}. The first author was partially supported by JNICT/Praxis XXI BD/3123/94.

pectives where evaluated; these approaches were classified and organized in several taxonomies [27, 14, 21, 6, 23]. A common conclusion stemming from this research is that integration of concurrency and object orientation is not a trivial task.

In the presence of multiple threads, and thus multiple active methods, some mechanisms are needed for regulating concurrent accesses to object data. However, the introduction of concurrency control mechanisms in the object model, can interfere with the inheritance mechanism. The definition of derived components often requires modifications to existing concurrency control code in the supertype, specially when this code is embedded within the operational code. As a result, inheritance is severely impaired unless appropriate abstractions are used for the incremental definition of concurrency control code. This interference is known as the inheritance anomaly [21], and it has been the subject of intensive research on several frameworks [22, 19], particularly on Actor based languages as ABCLONAP1000 [21], ACT++ [15, 14], ROSETTE [31].

2.1 Brief Introduction to the Inheritance Anomaly

In the study of the inheritance anomaly, some examples are widely used to depict specific problems in inter-object [21, 14] and intra-object[1] [22, 30, 18, 6, 4, 28] concurrency control.

2.1.1 Inter-Object Concurrency Control

Explicit Reception (Acceptance) of Messages

Languages such as ADA [12] and CSP [13] adopt bodies that regulate the acceptance of messages on active server objects. On these bodies, possibly guarded accept statements are interleaved within the operational body code. The addition of new methods on derived components incurs on possibly very complex body redefinitions, thus preventing the inheritance of bodies. Some affected languages are POOL-I [2], μC++ [8], EIFFEL// [10].

Accepting Methods According to the Abstract State

Addition of new methods often changes existing relations between the abstract object state and the invocable methods. Languages based on *accept sets* [15, 31] define named entities that describe which methods are invocable at a given moment.

Consider a stack with `push` and `pop` operations. We can identify the following accept sets that describe different abstract states:

State	Methods
Empty	push
Partial	push, pop
Full	pop

However, introducing a method `pop2` that removes two elements from the stack creates a new relevant state, thus partitioning the previous representation as shown below.

State	Methods
Empty	push
Singular	push, pop
Partial	push, pop, **pop2**
Full	pop, **pop2**

Languages that rely on the explicit switching of the current abstract state such as ACT++[2] [15] and ROSETTE [31] , are unable to program this example without awkward redefinitions.

The introduction of more flexible mechanisms for the use and extension of *accept sets* as those introduced in the new ACT++ version [14] and in ABCLONAP1000 [21] solves this problem. However, there remains the subtle problem of exposing the implementation details of the concurrency control code. The programmer of a derived component must know the structure of the inherited sets, so as to be able to change them, and include the newly defined methods where appropriate.

Accepting Methods According to History Dependencies

A potential alternative to the *accept sets* is to rely on the definition of method predicates [3, 19, 28], and to control method invocation with boolean guards. This allows rather elegant solutions to the previous problems.

[1] These schemes also apply to inter-object coordination.

[2] The version presented at ECOOP'89.

For example, if `load` expresses the current number of elements on the stack, and `MAX` is its maximum size, we can associate the guards $(load < MAX)$, $(load \geq 1)$, $(load \geq 2)$ to `push`, `pop`, `pop2`, respectively. These guards are mutually independent and can be added along with new methods on derived components.

Unless guards are assisted by additional synchronization actions, the definition of methods that depend on the invocation history of the components is made impossible as the appropriate history information must be recorded. The definition of a method `stat` that should only be accepted after 100 invocations of the other stack methods would require the use and update of a counter. Guards should also be extensible on derived classes, although not necessarily as in [19] where they can only be made more restrictive.

2.1.2 Intra-Object Concurrency Control

Intra-object concurrency can have a significant impact on performance. For example, in a simple experiment carried out on a 2-cpu system, disabling the internal concurrency of a producer-consumer through a bounded buffer reduced efficiency to 57% [5].

Approaches to the inheritance of intra-object concurrency control were evaluated with *accept sets* in [22] and *synchronization counters*[3] in DRAGOON [28], SINA [6] and DOOJI [30]. The use on DEMETER [18] of *exclusive regions* to which methods are associated is, to some extent, similar to the definition of named *accept sets*.

2.2 Concurrency Annotations Control Model

The CA (Concurrency Annotations[4]) model [4] defines a way of introducing separate concurrency control mechanisms for the coordination of multiple threads of execution and protect concurrent accesses to the internal object's state.

[3] Such as the number of threads that are executing a given method.

[4] This term is also used in CEiffel[17] but with a different meaning.

With CA, method invocation is protected by a guard and a lock. Only when the guard evaluates to true and the lock is opened can the corresponding method be allowed to execute, otherwise the calling thread is blocked in a wait queue. Concurrency control actions can be executed before and after method execution. These actions are standard code of the host language and may define and change private state elements of the synchronization code. They also have the special capability of acting on the method locks, and are able to use the interface of the controlled object. Actions and method guards can be independently expanded or redefined on derived components.

Concurrency control in BALLOON uses a modified version of the CA model. These features will be presented after an overall description of BALLOON.

3 The BALLOON Programming Language

BALLOON is a concurrent object-oriented programming language aimed at providing expressive abstractions for the reuse and composition of software components. In some aspects BALLOON resembles POOL-I [2] and SATHER [29] providing the complete separation of types and classes. Types specify the interface of objects sufficient for their correct composition and interaction in a message-based system [23]. Types are implemented by classes, which are the main structuring block of the language. This binding is a many-to-many relationship in the sense that a type can have multiple implementations and a class can also implement several types. The pair of a type and its implementations is usually referred to as a component.

Due to this explicit separation of types and classes, subtyping and subclassing relations are also distinguished concepts as advocated in [11, 24]. Subtyping serves data abstraction and conceptual structuring [16] while subclassing is left unconstrained allowing a flexible code reuse mechanism through inheritance [32, 25].

Genericity in BALLOON is achieved through generic components. Generic components differ from ordinary components by defining particular *nuclear types* on which the component's specification and implementation depend. The genericity mechanism provides the ability to derive new components by the explicit substitution of the nuclear types of a generic. Using explicit substitutions instead of parameterization, generic types and classes can be classified at the very same level as their ordinary counterparts. This is an important result for the orthogonality of the two code reuse mechanisms [26].

3.1 Type Hierarchy

Types form a well-defined hierarchy ordered by the subtype relation. As usual, because types are interface specifications, subtyping is defined in terms of interface compatibility [32]. A type is said to be a subtype of another if for every method of the later the former provides a method with the same name preserving covariance in the result and contravariance in its arguments [9]. Subtype polymorphism is supported by allowing an object to be substituted, in any context, by objects whose type is a subtype of the former's type.

The type hierarchy of BALLOON is therefore constituted by ordinary and generic types. The main order[5] in the hierarchy is given by subtypes. At the top of the hierarchy there is the type *Any* whose every type is subtype of, and there is also a virtual type, named *Nil*, which is a subtype of any other type in the hierarchy.

3.2 Class Hierarchy

Classes related by inheritance form an hierarchy. There is not any conformance relation between classes related by inheritance. The relation stands solely as a class constructor mechanism. Inheritance is the pure inclusion of code with subsequent re-binding of self references.

Because the inheritance mechanism is unconstrained, allowing multiple inheritance, renaming of inherited methods, and absence of any conceptual ordering, it is likely to yield a multi-topped hierarchy.

[5] Aside from subtyping, other type relations are defined in BALLOON but they are not relevant to this paper.

3.3 Relevant Constructions

Examples of the basic constructions of BALLOON relevant to this paper are now presented[6]. The examples remain simple serving as a base for the extensions that will be presented in the next section.

3.3.1 Types

A type in BALLOON is the declaration of the interface the objects of that type must support. An example is depicted next

```
Type IntQueue
  {
    Integer PutLeft(Integer);
    Integer GetRight();
    Integer Size();
  }
```

A possible subtype of IntQueue is an IntDEQueue (a double ended queue of integers). Its definition may be

```
Type IntDEQueue SubtypeOf IntQueue
  {
    Integer PutRight(Integer);
    Integer GetLeft();
  }
```

which means that IntDEQueue has all the methods of IntQueue unchanged plus the PutRight and GetLeft methods. As the received methods of IntQueue remain unchanged they are, by default, contravariant in their arguments and covariant in their results.

3.3.2 Generic Types

Container types (for example sets, bags, queues, etc) are more likely to be defined as generic types from where specific ones might be derived. The next example defines a generic DEQueue.

```
Type DEQueue
  {
    BasedOn T <: Any;

    T PutLeft(T);
    T PutRight(T);
    T GetLeft();
    T GetRight();
    Integer Size();
  }
```

DEQueue can hold any kind of objects because it is based on Any. Being based on Any it can be used

[6] A report on the definition of the language is currently being written [1].

to derive any kind of DEQueue. The above IntDEQueue could be obtained by

```
Type IntDEQueue
  DEQueue[Integer/T]
```

Deriving a new type from a generic one can be done by the substitution of a nuclear type by any subtype of the nuclear type.

It should be noted that a generic is itself a complete type. It can be used in any context an ordinary type is because their nuclear types are always bound to existing types. In the DEQueue example, T is bound in the type declaration to Any by the T <: Any expression. Defining a type as ordinary or as generic only depends on the conceptual entity it models.

The derivation of a new type from a generic does not force any type relationship between these types. As with any other types they may be related by subtyping or simply unrelated.

3.3.3 Classes

Classes provide the implementations of the interfaces declared in the types. The binding is made in the class definition. The class declares the type or types it implements. The following is the implementation of the IntDEQueue type. Notice that, in the example, we do not bother to validate any methods in exception conditions such as the full and empty queues. This will be done in the next section using concurrency control extensions.

```
Class CIntDEQueue Implements IntDEQueue
  {
     Data IntArray contents;
     Data Integer(CInteger) left, right, size;

     CIntDEQueue(Integer qsize)
       {
          contents = CIntArray(qsize);
          size := qsize;
          left := 0;
          right := size - 1;
       }
     Integer PutLeft(Integer elem)
       {
          contents.Put(left,elem);
          left := (left + 1) % size;
          return elem;
       }
     Integer PutRight(Integer elem)
       {
          contents.Put(right,elem);
          right := (right - 1 + size) % size;
          return elem;
       }
     Integer GetLeft()
       {
          left := (left - 1 + size) % size;
          return contents.Get(left);
       }
     Integer GetRight()
       {
          right := (right + 1) % size;
          return contents.Get(right);
       }
     Integer Size()
       {
          return size;
       }
  }
```

As IntDEQueue is a subtype of IntQueue its implementation CIntDEQueue is also an implementation of IntQueue. This holds for any type, that is, any implementation of a type is also an implementation of that type supertypes. Any implementation of a type is also an implementation of any of its supertypes.

To show the independence of the inheritance mechanism from type relationships consider the next example. Suppose a type IntStack (a stack of integers) defined as

```
Type IntStack
  {
     Integer Push(Integer);
     Integer Pop();
     Integer Top();
     Integer Size();
  }
```

which has no relation with IntDEQueue (nor with IntQueue). However, a possible implementation of IntStack can be achieved by inheritance of the CIntDEQueue class as depicted next

```
Class CIntStack Implements IntStack
  {
     Inherits CIntDEQueue Renames
          PutLeft as Push, GetLeft as Pop;

     CIntStack(Integer qsize)
       {
          CIntDEQueue(qsize);
       }
     Integer Top()
       {
          var Integer(CInteger) i;
          i := (left - 1 + size) % size;
          return contents.Get(i);
       }
  }
```

CIntStack provides an implementation to IntStack. It inherits from CIntDEQueue, renames PutLeft and GetLeft to match the IntStack type and implements (apart from the constructor) the Top method.

Finally, an implementation of a generic type may be sketched. Implementing the DEQueue type is like writing the CIntDEQueue class but now based on the type Any such as:

```
Class CDEQueue Implements DEQueue
  {
     BasedOn T <: Any;
     Data Array contents;
     Data Integer(CInteger) left, right, size;

     ...
```

that is replacing the relevant Integer type occurrences by T which is the nuclear type of the class.

For each type derived from the generic DEQueue an implementation can be derived from this generic class. As an example consider the following implementation for the IntDEQueue.

```
Class CIntDEQueue
  CDEQueue[Integer/T]
```

An interesting point about substitution instead of parameterization is that it is orthogonal to inheritance. This allows the two mechanisms to be composed, that is, both CDEQueue and CIntDEQueue could be now inherited and extended.

4 Concurrency Control In BALLOON

This section shows how the BALLOON type system is used to document and apply concurrency control policies. The IntQueue type will be enriched with a new section, **CC**, where some activation clauses and additional methods are defined. These methods are only accessible in activation clauses that are defined in the **CC** section of the type or in its subtypes. These activation clauses follow the format: **Delay** *method-name* **Until** *predicate*. The activation of the method is delayed until the predicate evaluates to true.

```
Type ConcIntQueue SubtypeOf IntQueue
  {
     CC:
     Integer MayPut();
     Integer MayGet();
     Bool AllowLeft();
     Bool AllowRight();
     Delay PutLeft Until ( AllowLeft() && MayPut() > 0 );
     Delay GetRight Until ( AllowRight() && MayGet() > 0);
  }
```

The two clauses specify in which circumstances should the operational methods PutLeft and GetRight be allowed to execute. As the Size method returns a constant value that is fixed after instance creation, there is no need for an activation clause. These clauses introduce a predicate that uses the operational and concurrency control methods defined in this type or in its supertypes.

Appearing in the type's definition, the **CC** constraints are meant to characterize the type's allowed concurrency. These constraints act, to some extent, as a specification for all implementations of the type. Although the concrete behaviour of the introduced methods is unspecified, the **Delay Until** clauses restrict the set of implementations of the type. An example of this will be given in section 4.1 after the introduction of the ConcIntDEQueue type.

The newly introduced concurrency control methods have their implementation in appropriate classes. These classes do not directly implement a type as before, but are said to control operational ones. Mixed through inheritance, an operational class and its controller class yield a class that fully implements the concurrent type. The encoding of the activation clauses takes place in any class that implements a concurrent type. This separation of operational and concurrency control classes provides the ability of their independent manipulation and reuse.

The example that follows presents an implementation of the **CC** methods in type IntQueue. In addition to the class syntax used above, the class CCIntQueue is able to define a wrapper that adds pre- and post-actions to the operational methods which are represented by an inner statement. The syntax is: **Wrapper** *method-name* {*pre-actions*} **Inner** {*pos-actions*}. It is worth to note that *inner* represents only the operational methods for objects of this type. In a subclass the wrapper is inherited and the *inner* represents the heir's own wrapper. This notion of the *inner* construct is similar to that found in BETA [20] inheritance mechanism.

These actions are in the scope of the class CCIntQueue definition, and so may only refer to instance variables defined in that class (or in inherited) and to the operational interface of the controlled class.

All concurrency control actions are executed with mutual exclusion in each object. So, only the operational methods are allowed to run in parallel. This also ensures that, if a method invocation successfully passes the activation clause, then the corresponding pre-actions are finished before allowing the evaluation of another invocation.

```
Class CCIntQueue Controls CIntQueue
  {
     Data Integer(CInteger) mayput, mayget;
     Data Bool(CBool) allowleft, allowright;

     CCIntQueue()
       {
          allowleft := true;
          allowright := true;
          mayput := Size();
          mayget := 0;
       }
     Integer MayPut() { return mayput; }
     Integer MayGet() { return mayget; }
     Bool AllowLeft() { return allowleft; }
     Bool AllowRight() { return allowright; }
     Wrapper PutLeft
       { allowleft := false; mayput := mayput - 1;}
       Inner
       { allowleft := true; mayget := mayget + 1;}
     Wrapper GetRight
       { allowright := false; mayget := mayget - 1;}
       Inner
       { allowright := true; mayput := mayput + 1;}
  }
```

With this scheme CIntQueue[7] is regulated for intra and inter-object concurrency. Operations that would mutually interfere are made exclusive, whilst those that are independent are allowed to run in parallel. This behaviour was expressed by the use of the flags *allowleft* and *allowright*. A new class that implements the concurrent type ConcIntQueue is built by mixing the operational class CIntQueue and its controller CCIntQueue as shown below

```
Class CConcIntQueue Implements ConcIntQueue
  {
     Mix CIntQueue, CCIntQueue;
  }
```

The semantics of **Mix** is that of inheritance plus the wrapping of operational methods.

[7] In fact the CIntQueue definition has been omited. It is subsumed by CIntDEQueue.

The coordination of inter-object concurrency was expressed in the type's activation clauses and relied on the current "putable" and "getable" number of elements of the queue.

The next section shows how these mechanisms are integrated with subtyping and subclassing.

4.1 Subtyping and Subclassing with Concurrency Control

The concurrency control information for the type IntDEQueue, subtype of IntQueue, may be defined as follows

```
Type ConcIntDEQueue SubtypeOf IntDEQueue, ConcIntQueue
  {
     CC:
     Delay PutRight Until ( AllowRight() && MayPut() > 0);
     Delay GetLeft Until ( AllowLeft() && MayGet() > 0 );
  }
```

The class CCIntDEQueue is a possible implementation for the concurrency control of CIntDEQueue.

```
Class CCIntDEQueue Controls CIntDEQueue
  {
     Inherits CCIntQueue;

     Wrapper GetLeft
       { allowleft := false; mayget := mayget - 1;}
       Inner
       { allowleft := true; mayput := mayput + 1;}
     Wrapper PutRight
       { allowright := false; mayput := mayput - 1;}
       Inner
       { allowright := true; mayget := mayget + 1;}
  }
```

The use of class inheritance, in this case reusing CCIntQueue, allowed the definition of concurrency control for just two methods, the ones introduced on the subtype IntDEQueue. Possible intra-object concurrency interferences are dealt by excluding operations on the same side of the queue. This is aided by the inter-object coordination that avoids the join of the sides, this would happen if, for example, two opposite get operations where allowed when there is only one element in the queue. The activation clauses and the modification of the mayput and mayget variables on the wrapper actions ensures that this situation does not occur.

The way the activation clauses were written in the ConcIntDEQueue type allows the concurrent execution of left and right methods. Recalling what have been said earlier about the specification role of the activation clauses, the reader should notice that implementations of ConcIntDEQueue are constrained by them. Only operational classes that do permit the concurrent execution of left and right methods may be used to implement this type. As a counter example, an algorithm based on a global counter of elements that would be incremented by both put methods and decremented by both get methods would not correctly implement this ConcIntDEQueue type.

It became apparent, and results quite naturally, that in most cases the pattern of class inheritance for concurrency control mimics the subtype relationships. However there are some exceptions, as in the type ConcIntStack.

```
Type ConcIntStack SubtypeOf IntStack
  {
    CC:
    Bool Allow();
    Delay Push Until ( Allow() && MayPush() > 0 );
    Delay Pop Until ( Allow() && MayPop() > 0 );
    Delay Top Until ( Allow() && MayPop() > 0 );
  }
```

Concurrency control is trivially implemented by reusing CCIntDEQueue code.

```
Class CCStack Controls CIntStack
  {
    Inherits CCIntDEQueue Renames
      AllowLeft as Allow, MayPut as MayPush,
      MayGet as MayPop;
  }
```

Finally, to what concerns generic types concurrency control, it could be easily shown that since the concurrency control does not interfere with the operational interface of types, the control of derived types can be exactly that of the generic type it was derived of. The same happens for concurrency control of generic implementations. As an example, the reader may notice that the concurrency control of the above IntDEQueue example would be the same of the DEQueue generic type.

4.2 Inheritance Anomaly Examples

Although the implementation of ConcIntDEQueue and ConcIntStack already shows the avoidance of the inheritance anomaly, some additional examples are required to show how the BALLOON approach deals with the anomaly. The previous examples also show that the addition of methods (such as Size) that do not interfere with existent ones is trivial since the default activation clause is: **Delay** *method-name* **Until** (true).

The introduction of a method GetLeft2 (equivalent to Pop2[8]) will show how to cope with changes of the abstract state of the object. Consider the following subtype of ConcIntDEQueue

```
Type ConcIntDEQueue2 SubtypeOf ConcIntDEQueue
  {
    IntegerPair GetLeft2();

    CC:
    Delay GetLeft2 Until ( AllowLeft() && MayGet() > 1 );
  }
```

and the controller class of the CIntDEQueue2 (which for space saving reasons is omitted) is

```
Class CCIntDEQueue2 Controls CIntDEQueue2
  {
    Inherits CCIntDEQueue;

    Wrapper GetLeft2
      { allowleft := false; mayget := mayget - 2;}
      Inner
      { allowleft := true; mayput := mayput + 2;}
  }
```

which delivers a concise description of the synchronization scheme for the new type and reuse all the previous code.

Until now, there was no need to redefine activation clauses or extend actions. The definition of a method GGetRight or GGetLeft that only performs a *get* operation if used immediately after a *put* operation would require the recording of additional information to express history dependencies. The same happens with a Stat method that may only work after 100 invocations.

The following implementation of Stat in a subtype of ConcIntDEQueue shows how this is achieved in BALLOON. From this example its straightforward to infer how to implement GGet operations.

```
Type ConcIntDEQueueS SubtypeOf ConcIntDEQueue
  {
    Integer Stat();

    CC:
    Integer Calls();
    Delay Stat Until ( Calls() > 100 );
  }
```

[8] If we had opted to extend the type Stack the description would be cluttered by irrelevant renaming issues.

```
Class CCIntDEQueueS Controls CIntDEQueueS
  {
     Inherits CCIntDEQueue;
     Data Integer(CInteger) calls;

     CCIntDEQueueS()
       {
          calls := 0;
       }

     Integer Calls() { return calls; }
     Wrapper PutLeft {} Inner { calls := calls + 1; }
     Wrapper GetRight {} Inner { calls := calls + 1; }
     Wrapper PutRight {} Inner { calls := calls + 1; }
     Wrapper GetLeft {} Inner { calls := calls + 1; }
  }
```

Consider, in this example, the wrapper construct of method PutLeft. Here **Inner** refers to the PutLeft method only. This whole wrapper constitutes now the **Inner** of the corresponding wrapper in the inherited controller class. That is, it corresponds to the **Inner** of the PutLeft wrapper defined in CCIntDEQueue.

The expansion in this case is as follows:

```
From CCIntQueue :
[ { allowleft := false; mayput := mayput + 1;}
  From CCIntDEQueue :
  [ {}
    From CCIntDEQueueS :
    [ {}
        Code of PutLeft from CIntDEQueue
      { calls := calls + 1; }
    ]
    {}
  ]
  { allowleft := true; mayget := mayget +1;}
]
```

These rules provide the ability for concatenation of pre- and pos-actions through the hierarchy of control classes.

A difficult case of inheritance anomaly is brought by the introduction of reusable generic locking mechanisms, generally provided in mixin classes [7]. Mixins are specially designed classes so that when mixed with other classes (usually through multiple inheritance) provide some added functionality.

The type ConcIntDEQueueL introduces three additional methods named AllLock, WriteLock and UnLock. These methods are intended to, respectively, lock all the methods, lock the methods that change the object state and unlock all of the previously locked methods. This type also shows how to redefine (in this case extend) previously defined activation clauses.

```
Type ConcIntDEQueueL SubtypeOf ConcIntDEQueue
  {
     AllLock();
     WriteLock();
     UnLock();

     CC:
     Bool AllowAll();
     Bool AllowRead();
     Delay PutLeft Until ( SuperClause && AllowAll() );
     Delay GetRight Until ( SuperClause && AllowAll() );
     Delay PutRight Until ( SuperClause && AllowAll() );
     Delay GetLeft Until ( SuperClause && AllowAll() );
     Delay Size Until ( SuperClause && AllowRead() );
  }
```

Consider the next two classes. They are a operational class and a control class that act as simple containers of generic code.

```
Class CLock
  {
     AllLock() {}
     WriteLock() {}
     UnLock() {}
  }

Class CCLock Controls CLock
  {
     Data Bool(CBool) allowall, allowread;

     CCLock()
       {
          allowall := true;
          allowread := true;
       }

     Bool AllowAll() { return allowall; }
     Bool AllowRead() { return allowread; }
     Wrapper AllLock { allowread := allowall := false; }
          Inner {}
     Wrapper WriteLock { allowall := false; }
          Inner {}
     Wrapper UnLock { allowread := allowall := true; }
          Inner {}
  }
```

These classes can now be inherited in order to implement the ConcIntDEQueueL type.

```
Class CConcIntDEQueueL Implements ConcIntDEQueueL
  {
     Mix CCLock, CLock, CIntDEQueue;
  }
```

With this scheme the introduction of locks in a given type only requires the extension of the existent activation clauses. The newly created type ConcIntDEQueueL clearly expresses in its signature the new invocation constraints, without being cluttered with implementation details. If necessary, some activation clauses can be defined for methods AllLock, WriteLock and UnLock so that, for example, UnLock is only allowed after a AllLock or WriteLock invocation.

5 Related Work

The separation of types from classes has already been proposed in object-oriented languages [2, 29]. This is mainly due to the observation that the subtype relation does not always follow inheritance [11]. America introduced these notions in a language with concurrency concerns [2] from where this paper in many aspects has its beginnings. Lately, several researchers [23, 21] point this separation as a promising issue in treating the inheritance anomaly in concurrent object-oriented languages.

In what concerns concurrency control the use of *accept sets* has been a common factor in most of the recent proposals for solving the inheritance anomaly. The amount of auxiliary mechanisms for the definition, extension and sometimes dynamic computation of *accept sets* varied considerably along these proposals. In particular, ABCLONAP1000 [21] introduced multiple mechanisms for the computation of *accept sets* showing how they were applicable to the inheritance anomaly. Recently the DOOJI model [30] studied the extension of some ABCL mechanisms for the support of intra-object concurrency. Coordination of intra-object concurrency relied on *synchronization counters* also used on DRAGOON [28] and SINA [6].

We believe that the default computation of *synchronization counters* can be avoided since, when needed, they are easily programmable in pre- and pos-actions and then used on BALLOON activation clauses. This reduces to "only when necessary" an otherwise fixed overhead.

BALLOON also avoids the use of *accept sets* since they require in some cases, such as when adding a Get2 operation, complex manipulations of the inherited sets (see [21, 30]). These manipulations and renamings are often difficult to track along the inheritance chain. The use of extensible activation clauses together with extensible wrapper actions provides, in most cases, a simpler and more self contained solution.

The separation of concurrency and operational code, mandatory in BALLOON, has been (in different degrees) previously defended on several frameworks [18, 28, 22, 4].

6 Conclusions

The main contribution of this paper is the introduction of a concurrency control model that takes advantage of the language distinction between types and classes to solve elegantly the typical problems of the inheritance anomaly. In particular, components that were designed without any concurrency concerns can be easily extended to fit in concurrent environments. This is achieved by annotating the component's type and designing appropriate controller classes for its coordination. This coordination respects intra- and inter-concurrency control.

The ability to define concurrency control for generic types introduces a new, orthogonal, reuse mechanism. Due to the modularity inherent to the model, concurrency control does not interfere with encapsulation and inheritance.

A similar concurrency control model (section 2.2) has already been tested as an extension of the C++ language, producing code for a shared memory multi-processor architecture with kernel-supported threads [5, 4]. This experience, although in a language with no separation of hierarquies, raised important implementation issues that have influenced the design of the BALLOON model.

A BALLOON compiler comprising the full set of language features described in this paper is under development.

Some issues are still open and under current research. One is the use of the inner construct for the composition of pre- and post-actions. Since the inner actions are nested, there is no easy way to cancel the inherited post-actions. However, such an example is rather unlikely in that it would also probably entail the redefinition of activations clauses. In this case, one would advocate a new control class.

7 Acknowledgments

We thank Paulo Sérgio Almeida for continous feedback on early drafts of this paper. We also thank Laurent Thomas, Cristina Lopes, Birger Andersen, Rachid Guerraoui, Lodewijk Bergmans, Eric Jul and the anonymous referees for their useful comments.

And of course, to the other members of the BALLOON development team, António Coutinho and Victor Fonte for their efforts in bringing BALLOON to life.

References

[1] P. Almeida, C. Baquero, A.Coutinho, V. Fonte, F. Moura, and Rui Oliveira. The Balloon programming language: Specification and Rationale. Technical report, DI - Universidade Minho, 1995. In Preparation.

[2] Pierre America. A parallel object-oriented language with inheritance and subtyping. In *ECOOP/OOPSLA'90*. Springer-Verlag, 1990.

[3] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, and Vandôme. Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1):31–67, 1991.

[4] C. Baquero and F. Moura. Concurrency annotations in C++. *ACM SIGPLAN Notices*, 29(7):61–67, July 1994.

[5] Carlos Baquero. Inheritance of synchronization code on object-oriented concurrent programming. Master's thesis, DI - Universidade do Minho, 1994. In Portuguese.

[6] Lodewijk Bergmans. *Composing concurrent objects*. PhD thesis, University of Twente, June 1994. ISBN 90-9007359-0.

[7] Gilad Bracha and William Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90*. Sringer-Verlag, 1990.

[8] P. Buhr, G. Ditchfield, R. Stroobosscher, and B. Younger. μ C++: Concurrency in the object-oriented language C++. *Software-Practice and Experience*, 22(2):137–172, February 1992.

[9] Luca Cardelli. Semantics of multiple inheritance. In *Semantics of Data Types*, LNCS 173. Springer-Verlag, 1984.

[10] Denis Caromel. Toward a method of object-oriented concurrent programming. *Comunications of the ACM*, 36(9):90–102, September 1993.

[11] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *POPL '90*, January 1990.

[12] G. Goos and J. Hartmanis. *The Programming Language Ada Reference Manual*. LNCS 155. Springer-Verlag, 1983.

[13] C.A.R. Hoare. Communicating sequencial processes. *Communications of the ACM*, 21(8):666–477, 1978.

[14] Dennis G. Kafura and R. Greg Lavender. Concurrent object-oriented languages and the inheritance anomaly. In *ISIPCALA'93*, 1993.

[15] Dennis G. Kafura and Keung Hae Lee. Inheritance in actor based concurrent object-oriented languages. In *ECOOP'89*. Springer-Verlag, 1989.

[16] Barbara Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5), May 1988.

[17] Klaus-Peter Lohr. Concurrency annotations for reusable software. *Comunications of the ACM*, 36(9):81–89, September 1993.

[18] Cristina Lopes and Karl Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In *ECOOP'94*. Springer-Verlag, 1994.

[19] Svend Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *ECOOP '92*, pages 185–196. Springer-Verlag, 1992.

[20] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison Wesley, 1993.

[21] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. Research Directions in Concurrent Object Oriented Programming, MIT Press, 1993.

[22] Christian Neusius. Synchronizing actions. In *ECOOP '91*. Springer Verlag, 1991.

[23] Oscar Nierstrasz. Composing active objects – the next 700 concurrent object-oriented languages. In *Research Directions in Concurrent Object Oriented Programming*. MIT Press, 1993.

[24] Rui Carlos Oliveira. Subtypes and subclasses in the object-oriented paradigm. Master's thesis, DI - Universidade do Minho, 1994. In Portuguese.

[25] Jens Palsberg and Michael Schwartzbach. Three discussions on object-oriented typing. *ACM SIGPLAN OOPS Messenger*, 3(2):31–38, 1992. Summary of ECOOP'91 Workshop on "Types, Inheritance and Assignments.

[26] Jens Palsberg and Michael I. Schwartzbach. Type substitution for object-oriented programming. In *OOPSLA/ECOOP'90*. Springer-Verlag, 1990.

[27] Michael Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming*. PhD thesis, Dept. of Computer Science, University of Geneva, 1992.

[28] S. Crespi Reghizzi, G. Galli de Paratesi, and S. Genolini. Definition of reusable concurrent software components. In *ECOOP'91*. Springer Verlag, 1991.

[29] C. Szyperski, S.Omohundro, and S. Murer. Engineering a programming language: The type and class system of Sather. LNCS 782. Springer Verlag, November 1993.

[30] Laurent Thomas. Inheritance anomaly in true concurrent object oriented languages: A proposal. In *IEEE TENCON'94*, pages 541–545, August 1994.

[31] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled-sets. In *OOPSLA '89*. ACM, 1989.

[32] Peter Wegner and Stanley Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP'88*. Springer-Verlag, 1988.