Concurrency Annotations in C++

Carlos Baquero* Francisco Moura†
DI / INESC
Universidade do Minho
4700 Braga, Portugal
{mescbm,fsm}@di.uminho.pt

February 4, 1994

Abstract

This paper describes **CA/C++**, **C**oncurrency **A**nnotations in **C++**, a language extension that regulates method invocations from multiple *threads* of execution in a shared-memory multiprocessor system. This system provides *threads* as an orthogonal element to the language, allowing them to travel through more than one object. Statically type-ckecked synchronous and asynchronous method invocations are supported, with return values from asynchronous invocations accessed through first class *future*-like objects. Method invocations are regulated with synchronization code defined in a separate class hierarchy, allowing separate definition and inheritance of synchronization mechanisms. Each method is protected by an access flag that can be switched in pre and post-actions, and by a predicate. Both must evaluate to true in order to enable a *thread* to animate the method code. Flags and method predicates are independently redefinable along the inheritance chain, thus avoiding the inheritance anomaly.

*Financed by JNICT grant BM92 / 3556 / IA.
†Part of this work was supported by JNICT PMCT 163/90.

1 Introduction

In the past, several approaches to concurrent object-oriented programming (**COOP**) raised the inheritance anomaly [10, 8], which restricts reuse by inheritance in the presence of synchronization code. Initial proposals to **COOP** were in fact largely exposed to this anomaly [6, 16, 1, 3]. Partial solutions to this problem were sought through (a) the separation of synchronization code and its reuse by inheritance [12, 15, 10] and (b) the first classing of synchronization code elements [5, 11]. Recently the use of multiple synchronization schemes in the same language was advocated, allowing the programmer to choose the most appropriate scheme for each case [9, 8]. **CA/C++** follows these ideas and provides two distinct synchronization mechanisms, thereby regulating both internal and external concurrency.

In **CA/C++** method invocations on any object can be specified on the client code as synchronous or asynchronous. In synchronous calls the client *thread* animates the object method, as in traditional **C++** code, returning control to the client after method completion. Asynchronous calls fork a new thread responsible for animating the method code. The client *thread* proceeds in parallel, though it can synchronize with the spanned thread and receive the results of the invocation by means of a *future*-like object. Note that all asynchronous interaction can be statically type-checked at compile time.

According to Kafura and Lavender's taxonomy of **COOP** [5] (which extends previous surveys [14, 13]), **CA/C++** is classified as an unrelated language in the animation model. This means that *threads* are not confined to the object boundaries but are able to animate several objects during their lifetime. As a result, *threads* represent an orthogonal element to the language. Since concurrency is external to the objects, simultaneous access to an object's state by multiple threads must now be synchronized. By contrast, recent proposals that cope with the inheritance anomaly, specially those derived from the Actor model [5], are classified as related approaches because threads exist only within an object.

Concurrency annotations are specified by a small amount of code and require less knowledge about the inherited synchronization code (annotations) than the approaches based on named sets of methods such as those present in **ABCL/R2** [9, 8] and some Actor languages [5, 16]. It contemplates synchronization schemes for both internal and external concurrency, raising a new

```
class Stack {
   protected:
   // private stuff
   public:
   void push();
   void pop();
};
// client code
...
Stack a, *pa=&a;
a.push();
pa->pop();
```

Figure 1: Stack class

perspective of the problem. In the next sections we describe **CA/C++**'s approach to the inheritance of synchronization code, and show its behavior in the presence of the anomaly cases.

2 Architecture of Concurrency Annotations

One of the main concerns in this model was to totally separate the classes that implement the object functionality from the classes that describe its behavior in the presence of concurrent method invocations. Our model creates separate class hierarchies in **C++** so that classes from the synchronization hierarchy can only access the public interface of the unregulated operational classes and the later don't even know the existence of the former.

Consider for example the typical sequential implementation of a Stack depicted in figure 1. With the information provided by the annotations, the **CA/C++** translator generates two new classes:

- A class that represents in **C++** the concurrency annotations of the Stack class.

- A class that defines a regulated Stack, which is used instead of the original unregulated Stack in the client code. This new class contains an instance of the unregulated Stack that provides the operational functionality, and inherits the concurrency class defined for Stack.

The hierarchy[1] defined by the regulated classes declares the same public interface of the unregulated classes and establishes the same inheritance relations. Therefore, a syntactic replacement in the client code suffices[2]. A translator for **CA/C++** was simply developed using James Roskind's **C++** grammar, integrating the concurrency annotation rules within the syntax.

The semantic of the **CA/C++** can be defined by means of the following informal rules

- In one object each method can be *on* or *off*. Its initial states are described in a constructor that can be extended in derived classes.
- Each method can be associated with a predicate, represented in **C++** by a Boolean 0-ary method that can be extended or redefined in derived classes.
- When a method is called, if its state in *on*, and there is no defined predicate or if there is a predicate and the function returns *true* then the caller *thread* is allowed to animate the object, otherwise the *thread* blocks in a wait queue.
- When a *thread* animates a method it executes firt the pre-actions defined in the **CA**, then the operational code defined in the contained unregulated object and finally the post-actions.
- The code in pre and post-actions can switch *on* or *off* any method defined in its class or in inherited classes. This code can also use private elements of the **CA** classes and the public interface of the unregulated object.
- For each object, mutual exclusion is enforced during the execution of pre or post-actions.

[1] In this case, a single class.

[2] Naturally some name colision checking must be performed in the translation proccess.

- After the execution of the pre or post-actions, some blocked *threads* may be allowed to run.
- Pre and post-actions can be extended by inheritance.
- Each **CA** class can define a constructor, responsible for initializing private elements and setting the initial state of each method invocability. These constructors are extensible by inheritance.

We can visualize the code responsible for the regulation of object invocations as placed in a meta-level to the unregulated object. While the contained object ignores the presence of synchronization code, this code only uses the public interface of the unregulated object, which leads to a strong encapsulation, and a clear frontier between these elements.

3 Concurrency Annotations for a simple Stack

Since the behavior of our stack with respect to internal concurrency must serialize operations that act on the same state elements, it follows that *pop* and *push* must exclude the set $\{pop, push\}$. This is denoted by the pre-action `@pop-; @push-;` and its corresponding post-action (figure 2). This is the traditional mutual exclusion synchronization that is necessary regardless of the stack internal state.

Condition synchronization [2] is necessary for expressing state-dependent semantics, for example to delay a *pop* invocation on an empty stack. In **CA/C++** this is achieved with the predicate. Note that unlike traditional guards and guarded commands, these predicates can be refined in derived classes, and are clearly separated from operational code.

The annotation in figure 2 keeps track of the stack state with the variable `fullness`. Although it is probably redundant because a similar variable is likely to exist in the unregulated stack, this redundancy is crucial to enable the annotation of pre-compiled libraries. Encapsulation would be compromised if the implementation was inspected.

The use of predicates to enforce condition synchronization does not prevent the occurrence of *deadlocks*. At present, no analysis of the client code is made for its detection. Some client code, with asynchronous method calls, that avoids a *deadlock* is shown in figure 3.

```
annotation Stack {
   state: { int fullness; }
   start: { @push+; @pop+; fullness=0; }
   push() {
      cond: { return (fullness!=MAX); }
      pre: { @push- ; @pop- ; }
      post: { fullness++ ; @push+ ; @pop+ ; }
   }
   pop() {
      cond: { return (fullness!=0); }
      pre: { @push- ; @pop- ; }
      post: { fullness-- ; @push+ ; @pop+ ; }
   }
}
```

Figure 2: Concurrency Annotation for the Stack class

```
Stack s;
assync Stack * @s_ptr;
@s_ptr(&s)->pop();
s.push(element);
result=@ptr<-pop();
```

Figure 3: Client code with asynchronous calls

4 Inheritance Anomaly and Concurrency Annotations

4.1 Partitioning of acceptable states

We will extend the unregulated stack code and provide the concurrency annotations for a new derived class. In this step, we introduce the method *pop2*[3] that will remove the two topmost elements of the stack, this operation must exclude $\{pop, push, pop2\}$ and the inherited ones must be extended to exclude the new method, since all act on the same state elements.

In the annotation described by figure 4, we relied in the previous use of `fullness` which lead to a small elegant solution, we may conceive an extreme situation were some pre-compiled library with compiled concurrency annotations is subject to extension, in such a case we could annotate the new method by defining a new variable in a constructor extension and properly extending the actions of the previous methods. Although such an extreme situation may seem very improbable it documents the flexibility and strong encapsulation provided by the **CA/C++** approach.

4.2 Addition of a history dependent method

The addition of a method *stat* usable only after 100 operations on the stack will show how operations that depend on the history of invocation can be annotated. The unregulated stack does not keep a counter for the number of invocations, neither does the previous two annotations, so there is no information on either state capable of expressing this behavior. The new annotation will extend the state (in the annotation hierarchy) to reflect the number of invocations and describe the predicate associated to *stat*. This extension will also be used to define a method *empty* that returns *true* from an empty stack.

The annotation, figure 5, of *empty* shows the locality and small interference of some extensions, specially those that do not change the state of the contained object. The method *empty*, as expected, may be invoked with no restrictions respecting internal concurrency. Approaches based on method

[3]Note that in the presence of concurrency, two pop invocations can be interleaved by other requests.

```
class RStack : public Stack
{
   public:
   void pop2();
};

annotation RStack : public Stack {
   start: {    // Extension to the constructor
      @pop2+;
   }
   pop2() {
      cond: { return (fullness>=2); }
      pre: { @push- ; @pop- ; @pop2- ; }
      post: { fullness-=2 ; @push+ ; @pop+ ; @pop2+ ; }
   }
   push() {   // Extension to the push actions
      pre: { @pop2- ; }
      post: { @pop2+ ; }
   }
   pop() {   // Extension to pop actions
      pre: { @pop2- ; }
      post: { @pop2+ ; }
   }
}
```

Figure 4: Stack with method pop2

```
class RRStack : public RStack {
   public:
   void stat();
   void empty();
};

annotation RRStack : public RStack {
   state: { int calls; }
   start: { calls=0; @stat+; @empty+; }
   empty() {}
   stat() {
   cond: { return( calls>=100 ); }
   }
   pop2() { post: { calls++ ; } }
   push() { post: { calls++ ; } }
   pop() {  post: { calls++ ; } }
}
```

Figure 5: Stack refined with methods stat, empty

sets [5, 9] require much more information about the previous synchronization code, since new methods must be added to existing sets or force some set's redefinition. With **CA/C++** the minimum information required to annotate an extension to a previously annotated hierarchy is the public interface, specifically the method names of the existing classes. Naturally, a good extension that minimizes redundancy and execution overhead can only be achieved by a correct integration with the inherited annotations.

If encapsulation is the prime concern, and object granularity is high enough to minimize the impact of **CA/C++** execution, the expressiveness provided could be used to create annotations that do not rely on the ones they inherit. Such approach, at the expense of redundancy, would enable localized changes to an arbitrary annotation, without redefinition of more derived ones[4].

4.3 Approach to the definition of *lock* extensions

Although multiple inheritance is not supported in this version of **CA/C++**, we will see how we could make a final extension to the current refinement of Stack that provides methods *lock* and *unlock* responsible for disabling and reenabling all the others methods. This extension, presented in figure 6, is a typical use of generic mixin classes [4] in multiple inheritance schemes, although not as generic as mixin classes, it shows how this behavior can be provided whenever needed.

In this annotation we extended any previous predicate in the inherited method annotations by a conjunction with the `locked` flag, the explicit reference to the previous predicate by the keyword `@cond`. Its omission leads to a redefinition of the method predicate; this conjunction creates a more restrictive predicate, similar to the ones proposed by Frolund [7].

5 Conclusions

The current version of the **CA/C++** translator uses the threads package of Solaris 2.3 on a SparcCenter 2000. Some preliminary results indicate that

[4]If a new variable `fullness`, with some other name, were defined in RStack annotation and method post-actions extended to actualize it, then we could remove `fullness` from the Stack annotation without afecting subsequent ones.

```
annotation LStack : public RRStack {
   state: { int locked; }
   start: { locked=1; @lock+; @unlock-; }
   lock() {
      pre: { @lock- ; }
      post: { locked=0; @unlock+ ; }
   }
   unlock() {
      pre: { @unlock- ; }
      post: { locked=1; @lock+ ; }
   }
   push()  { cond: { return ( @cond && locked ); } }
   pop()    { cond: { return ( @cond && locked ); } }
   pop2()  { cond: { return ( @cond && locked ); } }
   stat()  { cond: { return ( @cond && locked ); } }
   empty() { cond: { return ( @cond && locked ); } }
}
```

Figure 6: Stack with locks

the overheads introduced by the predicates, pre and post-actions and serialization code is comparable to those already introduced by the use of virtual methods. This suggests that relatively small-grain objects are feasible. The synchronization scheme is very simple and relies in a small number of features.

CA/C++ deals with the inheritance anomaly in the presence of both internal and external concurrency, while previous approaches were concentrated mainly with external (inter-object) concurrency. This pointed to the use of two different synchronization schemes, method access flags and predicates, respectively.

CA/C++ provides the means to regulate previously written, possibly compiled, class hierarchies. This capability improves the separation between operational and synchronization code, minimizing the influence of future changes to each hierarchy. It encourages the creation of independent annotations through the inheritance hierarchy.

References

[1] Pierre America. A parallel object-oriented language with inheritance and subtyping. In *ECOOP/OOPSLA'90*, pages 161–168. Philips Research Laboratories, ACM, October 1990.

[2] G. R. Andrews and F. B. Schnider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[3] Jan Van Den Bos and Chris Laffra. Procol, a parallel object language with protocols. In *OOPSLA '89 Proceedings*, pages 95–102, P.O. Box 9512, 2300 RA Leiden, The Netherlands, October 1989. University of Leiden, Department of Computer Science, ACM.

[4] Gilad Bracha and William Cook. Mixin-based inheritance. In *ECOOP/OOPSLA '90 Proceedings*, pages 303–311. ACM, October 1990.

[5] Dennis G. Kafura and R. Greg Lavender. Concurrent object-oriented languages and the inheritance anomaly. In *ISIPCALA'93*, pages 183,213, 1993.

[6] Dennis G. Kafura and Keung Hae Lee. Inheritance in actor based concurrent object-oriented languages. In *ECOOP'89 Proceedings*, pages 131–145. Cambridge University Press, 1989.

[7] Svend Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *ECOOP '92 Proceedings*, pages 185–196, 1304 W. Springfield Avenue, Urbana, IL 61801, USA, 1992. Department of Computer Science, University of Illinois at Urbana-Champaign, Springer-Verlag.

[8] Satoshi Matsuoka. *Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming.* PhD thesis, Department of Information Science, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan, April 1993. Thesis draft.

[9] Satoshi Matsuoka, Kenjiro Taura, and Akinori Yonezawa. Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages. In *OOSPLA '93 Proceedings, ACM SIGPLAN Notices*, volume 28, pages 109–126. n, October 1993.

[10] 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.

[11] Jose Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. Technical report, Computer Science Laboratory, SRI International, December 1992.

[12] Christian Neusius. Synchronizing actions. In *ECOOP '91 Proceedings*, pages 118–132. Springer Verlag, 1991.

[13] M. Papathomas. Concurrency issues in object-oriented programming languages. Technical report, s, 1989.

[14] Michael Papathomas and Oscar Nierstrasz. Supporting software reuse in concurrent object-oriented languages: Exploring the language design space. Technical report, 1990.

[15] S. Crespi Reghizzi, G. Galli de Paratesi, and S. Genolini. Definition of reusable concurrent software components. In *ECOOP '91 Proceedings*, pages 148–166. Springer Verlag, 1991.

[16] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled-sets. In *OOPSLA '89 Proceedings*, pages 103–112, 3500 West Balcones Center Drive, Austin, Texa 78759, October 1989. MCC, ACM.