CYES-C++: A Concurrent Extension of C++ through Compositional Mechanisms

Raju Pandey J. C. Browne
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{raju, browne}@cs.utexas.edu

December 10, 1994

1 Introduction

This paper describes a concurrent extension of the C++ programming language[1]. The extended C++ language, which we call CYES-C++, supports highly concurrent objects, implements a general concurrent method invocation mechanism, fully integrates the notion of inheritance with concurrency, and supports reusability of both method and synchronization specifications.

The computational model [4] for CYES-C++ is derived by integrating the ideas of a general model of computation, called the C-YES model[3], within the framework of the object-oriented paradigm. The major components of C-YES model of parallel computation are: i) separation of specification of computational behavior of components and specification of interactions among the components, and (ii) declarative specification of interaction as algebraic expressions over events. These ideas are incorporated in CYES-C++ by representing a concurrent object as a composition of two separate entities, methods and interaction specifications. Interactions among methods of objects are represented by algebraic expressions that capture semantic relationships among specific invocations of the methods. Also, concurrent method invocations in CYES-C++ are represented as a concurrent composition of invoking and invoked methods and expressions that define interaction between the methods. Separation of computation and interaction specification supports extensibility of concurrent classes through inheritance.

This paper is organized as follows: in section 2 we briefly describe the CYES-C++ computational model. We then present the various aspects of the language with suitable examples in section 3. In section 4, we give direction for future work along with some conclusions.

2 Computational Model

The computational model [4] for CYES-C++ is derived by integrating a general compositional model of concurrent programming, called the C-YES model [3], within the framework of the object-oriented paradigm. In the C-YES model, a concurrent program is composed from two separately specified entities: i) computational behavior specifications of components of the concurrent program, and ii) specification of interactions among the components. The computational behavior of a component merely specifies the operations that the component performs during its execution. Its interaction behavior, on the other hand, specifies how it relates with other components of the concurrent program. Separation of the specification for two aspects of execution behavior supports modular development of concurrent programs. Also, it supports extensibility of concurrent programs. The compositional approach in the C-YES model is unlike most other concurrent computational models in which specifications of components include both computational and interaction behavior specifications.

In the C-YES model, interaction is specified *declaratively* by algebraic expressions, called *event ordering constraint expressions*. Event ordering constraint expressions, called interaction expressions henceforth in the paper, represent interaction among component programs by specifying a set of relationships among specific occurrences of operations of the components. We call a specific occurrence of an operation an *event*.

In an object-oriented model, concurrency and interaction occur i) when methods execute in parallel within an object and interact while accessing shared resources (intra-object concurrency and interaction), and ii) when an invoking method and an invoked method execute in parallel and interact (inter-object concurrency and interaction). We use the concurrent program composition mechanism of the C-YES model to model both intra-object and inter-object concurrency and interaction.

2.1 Intra-object concurrency and interaction

Objects in the model are represented by the composition of two separate entities, method specifications and interaction behavior specifications. Formally, let the tuple $\langle M, \Phi \rangle$ define the composition of objects of a concurrent class C. Here, M is a set of methods and Φ is a set of event ordering constraint expressions. Every invocation of a method denotes a unique event. The semantics of the composition specifies that all invocations of different methods (method events) of an object O of class C execute in *parallel*, except for those whose execution orderings must satisfy *all* ordering constraints imposed by the expressions in Φ. Object O can thus be viewed as a concurrent program that is composed from various occurrences of methods in M and interaction expressions in Φ.

Unlike most approaches, where concurrency is added to a sequential object, our approach is to start with a model where concurrency is fundamental. All interaction constraints are specified explicitly. Composition of an object can be viewed in terms of defining semantic relationships among method events. In the absence of any knowledge about the application domain, concurrency is the fundamental relationship since it captures semantic independence among method events. Interaction, on the other hand, represents semantic dependencies (such as data dependency, data consistency, and priority) among the method events, and hence must be specified explicitly.

2.2 Inter-object concurrency and interaction

Inter-object concurrency and interaction occur when a method event invokes another method event in parallel. The two events interact while accessing common global and parameter objects. Our approach is to represent it as a composition of the calling and the called events. For instance, assume that a method event of m_1 invokes method m_2 on object o_2. The concurrency and interaction between the two is represented by embedding the following expression

$$\| \; O_2.m_2 \texttt{ where } \phi$$

in m_1's specification. Upon execution of the above expression, an $O_2.m_2$ event is constructed. The two events execute in parallel. However, their execution must satisfy the ordering constraints specified in the interaction expression ϕ. Interaction expression ϕ may specify ordering constraints over objects that the two events share. The above composition mechanism is general in that it subsumes traditional synchronous and asynchronous method invocation mechanisms. For instance, future-based asynchronous method invocation can be represented by specifying an event ordering constraint expression that orders the read and write events on a future variable.

2.3 Extensibility of Classes

Inheritance provides a framework for extending the definition of classes. There is a problem, termed the *inheritance anomaly* [2], with the inheritance of method implementations in many concurrent object-oriented programming languages. It arises because implementations of methods in most programming languages combine specifications of both computational and interaction behaviors. Any changes in the interaction behavior of a method in a subclass may require changes in its computational behavior as well, thereby breaking the inheritance mechanism. Our programming model fully integrates the notion of inheritance with concurrency by i) separating specifications of computational and

interaction behaviors, and ii) by allowing one to add/modify either or both components of a concurrent class. The model supports reusability of both method implementation and interaction specifications. Support for inheritance of interaction specifications is achieved by the modularity property of event ordering constraint expressions. Additional semantic dependencies can be incrementally added, and by localizing modifications in the semantic dependencies due to the changes in methods can be localized.

3 CYES-C++

We now give a brief overview of extensions that were added to C++.

3.1 Interaction specification

Interaction in CYES-C++ is represented by event ordering constraint expressions. Event ordering constraint expressions are constructed from a set of primitive ordering constraint expressions and interaction composition operators. A primitive ordering constraint expression represents interaction between two events, whereas the interaction composition operators are used to represent nondeterministic interactions as well as interactions among sets of events.

3.1.1 Event sets

Events sets form the abstraction for identifying and representing invocations of methods that interact with other invocations. They are fundamental to the interaction specification mechanism in that interaction expressions are defined over sets of events.

CYES-C++ supports a number of standard event sets. For instance, with each method `method` in a class, the following event sets can be used in interaction behavior specifications:

- `method`: set containing all invocations of `method`.
- `method:waiting`: set of all invocations of `method` that are waiting to be executed.
- `method:running`: set of all invocations of `method` that are currently executing.
- `method:terminated`: set of all invocations of `method` that have terminated.
- `method:future`: set of all invocations of `method` that will be invoked in future,

Note that the above sets allow one to capture the runtime state of different method invocations. In addition, the language also identifies a number of operators that allow a programmer to construct new event sets from existing events sets.

3.1.2 Primitive expression and interaction composition operators

Interaction expressions are defined by specifying relationships among events and by combining the relationships through the interaction composition operator:

- The primitive event ordering constraint expressions

  ```
  Cond => (Ev1 < Ev2)
  ```

  specifies that if condition `Cond` is true, event `Ev1` occurs before event `Ev2`. This expression is implemented by checking `Cond` before the execution of event `Ev2`. If `Cond` is true, `Ev2` is delayed until `Ev1` has terminated.

- And constraint operator (`&&`): The `and` constraint operator is used to impose additional ordering constraints by combining interaction expressions. For instance, expression `(a < b) && (c < d)` specifies that event `a` must occur before event `b` *and* event `c` must occur before event `d`.

- Or constraint operator(`||`) : The `or` constraint operator is used to model nondeterministic relationships among events. For instance, expression `(a < b) || (b < a)` specifies that either event `a` should occur before event `b` or `b` should occur before `a`.
- Forall: `forall` operator is an extension of the `and` constraint operator over a set of events. There are two forms of forall. In the first form,

```
forall variable  v in  EventSet :
    InteractionExpression(v)
```

  a variable *v* is used to ranges over events of set `EventSet` such that `InteractionExpression` holds for all events of `EventSet`. In the second form

```
forall occurrence  i in  EventSet :
    InteractionExpression(EventSet[i])
```

  occurrence number of event sets is used to identify events.
- Exists: `exists` extends the `or` constraint operator similarly.

3.2 Concurrent class specification

Concurrent objects in CYES-C++ are supported by explicitly specifying a concurrent class type. The interface of a concurrent class type contains, in addition to sequential C++ class interface definitions, interaction specifications among invocations of the methods. For instance, concurrent class `Queue` in figure 1 defines concurrent queue objects. The interface contains, in addition to `public`, `private`, and `protected` entities, `interaction` entities. The interaction entities are used to define the manner in which public methods interact with each other. The specification in figure 1 specifies that objects of class `Queue` are composed from set {`put, get, Full, Empty`} of methods and set {`SequentialPut, SequentialGet, DelayPut, DelayGet`} of interaction expressions. The semantics of the composition is that all invocations of the methods on an object of `buffer` execute in *parallel* except for those whose executions must satisfy *all* ordering constraints imposed by the event ordering constraint expression

```
SequentialGet && SequentialPut && DelayPut && DelayGet
```

Hence conceptually every method invocation creates a separate thread of control.

We now specify `SequentialPut`, `SequentialGet`, `DelayPut`, and `DelayGet`. Let interaction expression `WaitUntil` define interaction between events of sets `EventSet1` and `EventSet2` such that an event in `EventSet1` is delayed with respect to events of `EventSet2` until the boolean condition `cond` becomes false. The following expressions defines `WaitUntil`:

```
WaitUntil(EventSet1, EventSet2, cond) =
       forall variable  e1 in  EventSet1 :
           forall variable  e2 in  EventSet2 :
            cond => (e2 < e1)
```

Interaction expressions `DelayPut` and `DelayGet` can now be derived in terms of `WaitUntil`. Let `met:next` denote a set of events such that `met:next` contains all invocations of method `met` that i) are currently (at the time `met.next` is evaluated) executing or waiting, and ii) will be invoked in future. Hence,

```
DelayPut = WaitUntil(put, get:next, Full())
DelayGet = WaitUntil(get, put:next, Empty())
```

Similarly `SequentialPut` and `SequentialGet` are defined by specifying a an interaction expression `Serialize`:

```
concurrent class Queue {
    public:
       Queue() { first = 0, last = 0; };
       ~Queue() { };
       void put(char);
       char get();
       Boolean Full();
       Boolean Empty();
    interaction:
       SequentialPut;
       SequentialGet;
       DelayPut;
       DelayGet;
    private:
       char buffer[SIZE];
       int first, last;
}
```

Figure 1: Interface of concurrent class `Queue`

```
Serialize(EventSet) =
        forall occurrence  i in  EventSet :
            forall occurrence  j in  EventSet :
                (i < j)  =>  (EventSet[i] < EventSet[j])
```

Expression `Serialize` serializes events of `EventSet` in the order they arrive. Hence,

```
SequentialPut = Serialize(put)
SequentialGet = Serialize(get)
```

3.3 Method invocation

A method `func` can be invoked on an object `obj` by the following expression:

```
par obj.func(param1, param2, ..., paramN)
    where   Evoce
```

The interaction expression `Evoce` represents interaction between the calling method and an occurrence of `func`. CYES-C++ also supports the ability to invoke many methods in parallel through the `parfor` composition operator:

```
parfor (int i=0; i < n; i++)
    obj.func(param1, param2, ..., paramN)
        where   Evoce
```

3.4 Extensibility of concurrent classes

In our concurrent object-oriented programming model, interaction specifications of a concurrent class represent semantic relationships among methods of the class, and are assumed to be more specific than concurrency. As the concurrent class is extended by defining additional methods or by modifying existing methods, additional semantic dependencies among methods develop, and/or certain semantic dependencies need to be redefined. Support for the reusability of classes and interaction expressions in our model is provided by allowing one to extend interaction behavior of the methods incrementally and by localizing possible changes in the interaction behavior.

We show extensibility of classes through an example. Let class `ReadFirstQueue` extends class `Queue` by adding a method `getlast`. Method `getlast` retrieves the last element of the queue. Method `getlast` interacts with inherited method `put`. The interaction is specified by the following expressions:

```
DelayGetLast = WaitUntil(getlast, put:next, Empty())
DelayPutWithGetlast = WaitUntil(put, getlast:next, Full())
```

Note that computational behavior specifications of `put` and `get` are inherited in the subclass. Also, interaction expressions that define interaction among `put` and `get` events are inherited in the subclass. Methods of object of class `ReadFirstQueue` are constrainted by the expression

```
SequentialGet && SequentialPut && DelayPut && DelayGet && DelayGetlast && DelayPutWithGetlast &&
                                        SequentialGetlast
```

The reusability of interaction expressions is supported by allowing one to capture general interaction expressions such as `WaitUntil` and `Serialize`, and reinstantiate them in different situations.

4 Summary and future work

We presented a brief description of a concurrent programming language CYES-C++. CYES-C++ extends C++ programming language by incorporating concurrent composition mechanisms of the C-YES model. The two important components of the language are i) the separation of computation and interaction behavior specification, and ii) the interaction specification mechanism. The first allows us to integrate the notion of inheritance with concurrency. The second supports a general and modular mechanism for specifying interaction both within an object and among objects.

We are currently implementing CYES-C++ by writing a preprocessor that will translate a concurrent program represented in the CYES-C++ notation into C++ source program augmented with suitable calls for parallel thread creation and synchronization.

References

[1] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[2] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In *Research Directions in Object-Based Concurrency*. MIT Press, Cambridge, 1993.

[3] Raju Pandey and James C. Browne. Event-based Composition of Concurrent Programs. In *Workshop on Languages and Compilers for Parallel Computation, Lecture Notes in Computer Science 768*. Springer Verlag, 1993.

[4] Raju Pandey and James C. Browne. A Compositional Approach to Concurrent Object-Oriented Programming. In *IEEE Internation Conference on Computer Languages*. IEEE Press, May 1994.