Support for Extensibility and Reusability in a Concurrent Object-Oriented Programming Language

Raju Pandey
Computer Science Department
University of California
Davis, CA 95616

J.C. Browne
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

In many concurrent programming languages programs are difficult to extend and modify. This is because changes in a concurrent program (either through modification or extension) require re-implementation of some or all components. This paper presents the design of a concurrent object-oriented programming language based upon separate specifications of computations and interactions of programs. Separate specification of computations and interactions allows each to be separately modified and extended. We show that separation also facilitates extension of other language composition mechanisms such as class, inheritance, and template in order to define concurrent program abstractions. The resulting language supports extensibility and modifiability of concurrent programs as well as reusability of specifications of computations and interactions.

1 Introduction

There is significant interest in concurrent programming due to widespread availability of parallel and distributed systems. In recent years, many parallel systems have been introduced. These systems differ widely in their architecture, their scope, and the target problem domain. The design and implementation of concurrent programs for this wide range of machines has proven to be extremely difficult. Although there has been extensive work done in the area of concurrent programming, concurrent programs are still difficult to design, implement, and maintain. In many of these approaches, they are difficult to extend and modify because changes in a concurrent program (through addition of new components or modification of existing components) requires re-implementation in some or all components. Also, it is difficult to reuse specifications of components and interaction.

Concurrent object-oriented programming languages show promise in alleviating the modularity and extensibility problems. Concurrent objects form a natural basis for modeling entities of applications. Further, extensibility is naturally supported through the notion of inheritance. Many object-oriented programming languages have been proposed that extend a sequential object-oriented programming language by adding mechanisms for specifying concurrency and interaction. We note that the modifiability and extensibility problems are present in many concurrent object-oriented programming languages as well. For instance, there is a problem with the inheritance of method implementations in concurrent object-oriented programming languages. This problem, termed the *inheritance anomaly* [15], occurs when implementations of methods of a class cannot be inherited in a subclass due to the differences in synchronization constraints of the class and the subclass. Concurrent classes therefore cannot be extended easily. Similarly, interaction specifications cannot be reused easily.

The presence of the problems indicate that there are fundamental problems in the way a concurrent program is constructed from its components. We observe that concurrent programs are difficult to extend and modify because specifications of components include specifications of both its computations and interactions. Changes in either aspect (due to addition or modification of components) may require that the components be re-implemented. *Concurrent programs can be modified easily if specifications of computations and interaction are completely separated* . We call this approach "separation of concerns." [19]

In this paper, we present the design of an object-oriented concurrent programming, called CYES-C++. CYES-C++ is a concurrent extension of the C++ [21] programming language. The basis for the design of CYES-C++ is derived from the concept of separation of specifications of computation and interaction. The language supports mechanisms for specifying computations and interactions separately. In addition, it supports mechanisms that can be used for composing computational and interaction specifications in order to define concurrent program abstractions.

In proceedings of the International Parallel Processing Symposium 96

Many of the composition mechanisms such as class, inheritance, and genericity of C++ have been extended in CYES-C++ in order to define corresponding concurrent program abstractions. For instance, a concurrent class (a concurrent extension of class) defines a composition of method and interaction specifications, whereas inheritance forms a composition mechanism for extending a concurrent class by adding or modifying either component of a concurrent class. Separation of the two therefore allows one to inherit both specifications and extend them in suitable ways. The language therefore supports mechanisms that facilitate extensibility and modifiability of concurrent program abstractions as well as reusbility of both method and interaction specifications.

The ideas that we present in this paper are general in that they can be applied to extend any object-oriented concurrent programming language. We chose to apply them to C++ due to the wide acceptability of C++ and rich support in it for programming language abstractions. Further, we were motivated by the availability of many C++ tools. Indeed, we have been able to reuse many existing C++ and C libraries and tools for constructing a prototype implementation for CYES-C++.

This paper is organized as follows: In Section 2 we briefly describe the interaction specification mechanism used for specifying interaction among methods. Section 3 describes the syntactic and semantic details of the composition associated with a concurrent class. In Section 4, we describe the manner in which inheritance can be used to extend the composition of a concurrent class. Section 5 describes an extension of the template mechanism that allows one to capture common computational and interaction abstractions in generic concurrent classes. We give a brief overview of the related work in Section 6. Section 7 contains a brief summary and the status of our research.

2 Interaction specification

Interaction among programs is specified by an algebraic expression, called the *event ordering constraint expression*. It is used to represent semantic dependencies among events (specific invocations of operations or methods) of component programs by specifying execution orderings — deterministic or nondeterministic — among the events. An event ordering constraint expression is constructed from a set of *primitive ordering constraint expressions* and a set of *interaction composition operators*.

Primitive event ordering constraint expression: A primitive event ordering constraint expression is used to defined constraints on execution orderings of two events. It is defined:

```
E = (e1 < e2)
```

A computation satisfies expression `E` if event `e1` occurs before event `e2` in the computation.

Interaction composition operators: Interaction composition operators are used to combine primitive and non-primitive event ordering constraint expressions to construct more complex expressions.

i) And constraint operator (&&): The and constraint operator && is used to combine event ordering expressions such that additional constraints can be imposed on a program. An event ordering constraint expression containing && is defined:

```
E = (E1 && E2)
```

Intuitively, an execution of a program satisfies event ordering constraint expression `E` if it satisfies both `E1` and `E2`.

ii) Or constraint operator(||): The or constraint operator `||` is used to incorporate nondeterminism in the orderings of events. An event ordering constraint expression containing `||` is defined:

```
E = (E1 || E2)
```

Intuitively, an execution of a program satisfies event ordering constraint expression `E` if it satisfies *at least one* of the event ordering constraint expressions `E1` or `E2`.

iii) forall operator: The `forall` operator extends && in order to specify ordering constraints over sets of events. There are two ways in which the `forall` operator can be specified. The format for the first is:

forall var v in S { E(v) }

The above expression specifies that event ordering constraint expression `E(v)` holds true for all events `v` in event set `S`. The format for the other `forall` operator is:

```
forall occ i in S { E(S[exp(i)]) }
```

The above expression specifies that event ordering constraint expression `E(S[exp(i)])` holds true for all events `S[exp(i)]` of `S`. In this expression, variable `i` ranges over the occurrence number[1] of events of `S`. Here integer expression `exp(i)` determines the the occurrence number of the event for which `E` must hold. The differences in the two versions arise solely in the representation of events.

iv) Exists operator: The `exists` operator is similar to `forall` in that it extends the or constraint operator over a set of events.

The event ordering constraint expression is declarative in nature. Its power stems from the ability to decompose global interactions among programs into a set of local interactions. The local interactions can then be represented

[1]Every invocation of an operation or method is assigned a unique positive integer in an event set, termed its occurrence number.

by event ordering constraint expressions, and combined with suitable interaction composition operators to represent the global interaction. One of the implications of the modularity property of event ordering constraint expressions is that interaction behaviors of programs can be changed by modifying only the relevant and local interaction specifications. Also, it forms the basis for reusability of interaction behavior specifications in CYES-C++.

Further, the interaction specification mechanism is general in that it is not based on the semantic properties of any specific synchronization primitive. The generality of the approach also is useful in that it allows one to create separate abstractions of interaction which can be combined with other mechanisms such as inheritance and genericity to construct powerful concurrent program abstractions. We next describe how such compositions are supported in the language.

3 Concurrent class

Concurrent objects in CYES-C++ are represented by defining a concurrent class. An interface of a concurrent class contains an `interaction` section, in addition to `public`, `private`, and `protected` sections of C++ classes. The interaction section of a concurrent class contains definitions of event sets and event ordering constraint expressions used to represent interaction among the public methods of the concurrent class. Computational and interaction behaviors of methods of concurrent objects are therefore completely separated. The semantics associated with a concurrent class specifies that all invocations of public methods on a concurrent object execute in *parallel*, except for those whose executions must satisfy *all* ordering constraints specified in the interaction section.

In Figure 1, we show the concurrent class specification for concurrent `queue` objects. There are four constraints on the methods of `queue`: i) `put` invocations are sequential, ii) `get` invocations are sequential, iii) `put` events are delayed if the `queue` is full, and iv) `get` events are delayed if the `queue` is empty. In the figure, the constraints have been represented symbolically. In Section 3.1, we derive event ordering constraint expressions for the constraints. The semantics of the composition specifies that the methods execute in parallel by default. For instance, every invocation of method `put` on an instance of `queue` starts to execute in parallel. However, before it can be executed, all ordering constraints specified in the interaction section of `queue` must be satisfied. These constraints therefore determine if the invocation can proceed or should be delayed with respect to other invocations.

```
concurrent class queue {
public:
    queue();
    ~queue();
    void put(char);
    char get();
    Boolean Full();
    Boolean Empty();
interaction:
    SeqAdds;
    SeqRems;
    SyncQFull;
    SyncQEmpty;
private:
            ⋮
};
```

Figure 1: **Concurrent class specification of concurrent queue objects**

3.1 Event set

Event sets form the abstraction for identifying and representing invocations of methods that interact with other method invocations. They are fundamental to the interaction specification mechanism in that they allow us to represent both application-specific and application-independent states of a concurrent object. The application-specific state of an object is dependent on the semantics associated with an object. For instance, a `queue` object may have two states: full and empty. Both of these states are derived from the semantics of the object. An application-independent state, on the other hand, is defined for all objects. It is used to define the semantics of objects in general. An example of an application-independent state is the state of method invocations that are waiting to be executed. We call such states synchronization states.

For every method `M` of a concurrent class, the following default event sets are supported in CYES-C++: i) `M` denoting the set of all invocations of method `M`, ii) `M:waiting` denoting the set of all invocations of `M` that are waiting at an instance, iii) `M:running` denoting the set of invocations of `M` that are currently executing, and iv) `M:terminated` denoting all invocations of `M` that have terminated. In addition, CYES-C++ supports the following mechanisms for defining event sets in terms of other event sets:

Conditional event sets: Conditional event sets are used to capture states of concurrent objects and to associate these states with events. The term `M:B` denotes an event set. It contains all events of event set `M` for which the boolean condition `B` is true. An example of a conditional event set is the event set `get:empty()`. It captures all `get` invocations for which the condition `empty()` is true.

Named event sets: CYES-C++ supports the ability to name event set expressions. For instance, the expression

```
fullqueue = put:full()
```

defines an event set `fullqueue` that contains all events of set `put:full()`.

Event set expressions: Event sets can be combined with other event sets with the union (+) and difference (−) operators. Hence, an expression of the form

```
fullqueue = fullqueue + putlast:full()
```

extends `fullqueue` to include events of set `putlast:full()`.

We now illustrate the manner in which specifications of event sets, events, and event ordering constraint expressions can be used. The example shown here derives expressions for the interaction section of the concurrent `queue` class in Figure 1.

Example 3.1. *(Interaction specification).* We first define two named event ordering constraint expressions:

```
Serialize(S) {
  forall occ i in S {
    (S[i] < S[i+1])
}}
Priority(S1, S2) {
  forall var a in S1 {
    forall var b in S2 {
      (a < b)
}}}
```

Expression `Serialize` orders events of set `S` according to their occurrence number. (Term `S[i]` denotes the ith invocation of a method in set `S`). Expression `Priority` gives events of `S1` higher priority over events of `S2`. We now define four event sets, each capturing those method invocations that may interact with other methods:

```
AddQ = put
RemQ = get
QEmpty = get:empty()
QFull = put:full()
```

Set `AddQ` contains all events that add information to the `queue`. Set `RemQ` contains all events that remove information from the `queue`. Set `QEmpty` contains all events for which the queue is empty. Similarly set `QFull` contains all events for which the queue is full.

We now instantiate the named event ordering constraint expressions with suitable named event sets:

```
SeqAdds = Serialize(AddQ)
SeqRems = Serialize(RemQ)
SyncQEmpty = Priority(AddQ, QEmpty)
SyncQFull = Priority(RemQ, QFull)
```

Expression `SeqAdds` therefore serializes all events in set `AddQ` (invocations of `put`). Expression `SyncQEmpty`, on the other hand, delays all events in set `QEmpty` (invocations of method `get` for which the queue is empty) with respect to invocations of events of `AddQ`. ■

We emphasize the following: i) Interaction among the methods is specified by defining generic event ordering constraint expressions such as `Serialize` and `Priority`, and by instantiating them with specific event sets. The expressions can be reused in other interaction specifications as well. This shows one of the many ways in which abstractions for interaction can be created and reused. ii) Interactions are defined by identifying those method invocations that interact, and by representing them through the abstractions of named event sets. An example is the notion of the set `AddQ` which captures the abstraction of all events that add information to the queue. We will see later in the paper that such an abstract representation of interacting events make it easier to extend or modify interaction behaviors of methods.

4 Extensibility

In this section, we examine the notion of inheritance as a mechanism for extending program composition of a concurrent class by adding and/or modifying methods and their interaction behaviors.

4.1 Inheritance anomaly

In many concurrent object-oriented programming languages there is a problem with the inheritance of method implementations. This problem, termed the *inheritance anomaly* [15], arises due to the differences in synchronization requirements of a class and its subclasses. We illustrate the problem through the following example:

Example 4.1. *(Inheritance anomaly).* Let a concurrent class C define two methods m_1 and m_2. Implementations of m_1 and m_2 contain, in addition to specifications of computations, synchronization primitives used to define their interaction behavior. Let S be a subclass of C. It extends class C by defining a new method, say m_3. Method m_3 interacts with m_1 and m_2, thereby changing the interaction behaviors of m_1 and m_2, as defined in C. Methods m_1 and m_2 need to be re-implemented in S in order to represent the modified interaction behaviors. The implementations of m_1 and m_2, thus, are not inherited in S. ■

The inheritance anomaly arises because specifications of methods contain specifications of both computational and *interaction behaviors* [20]. Since specifications of methods include specifications of both computational and interaction behaviors, any changes in the interaction behavior may, therefore, require changes in the implementation as well. There are two components to the resolution of the inheritance anomaly: the first is separation of specifications of computational and interaction behaviors of methods. Separation makes it possible to inherit the two behaviors separately, and to modify either to reflect changes in the concurrent program composition of a concurrent class. The second is the ability to make changes in the interaction behaviors of methods. The inheritance anomaly has been studied in great detail and many solutions [12, 23, 17, 22, 16] have been proposed. Most of these solutions are based on the separation of synchronization constraints from method implementations.. Changes in interaction behavior of a method is achieved by changing the relevant synchronization constraints. Different instances of the inheritance anomaly do not occur in CYES-C++ because concurrent objects are specified as a composition of separate computational and interaction behavior specifications. In addition, CYES-C++ supports many mechanisms to allow changes in the interaction behavior of methods.

We give an example that illustrates the way in which the state partitioning anomaly can be resolved. The state partitioning anomaly occurs in the behavioral abstraction-based languages [12, 23, 14] when additions or modifications of methods in a subclass partition the states of objects of a superclass. Since the implementation of a method includes the state transitions that an object makes after the execution of the method, changes in the states (due to the state partitioning) therefore require that the method be re-implemented in the subclass in order to include transitions to the newly defined states. In CYES-C++, since states are captured through event sets, state partitioning is represented by additions or modifications of event sets.

Example 4.2. *(State partitioning).* Let `queueone` be a subclass of `queue`. It defines a method `gettwo`. Method `gettwo` accesses two elements of the `queue` object atomically. Invocations of `gettwo` are delayed with respect to `put` if the buffer is empty or has one element. Note that a `queue` object can be in one of the three states: full, empty, or partially filled. The addition of method `gettwo` thus partitions the partially filled state of `queue` into two: `queue` with one item, and `queue` with more than one item.

In CYES-C++, state partitions can be represented by defining new event sets in `queueone`. Let method `one()` return true if a `queueone` object contains one item. We first define the following event set:

```
GetOne = gettwo:one()
```

The event ordering constraint expression

```
SyncQOne = Priority(AddQ, GetOne)
```

represents the interaction between events of `GetOne` and events of `AddToQ`. We add events of `gettwo` to the following sets:

```
QEmpty = queue::QEmpty + gettwo:empty()
RemQ = queue::RemQ + gettwo
```

The event ordering constraint expressions of `queue` apply to invocations of `gettwo` as well. ■

In [18], we show many other instances of inheritance anomalies, and how they are resolved in CYES-C++.

5 Genericity

C++ provides the template mechanism for specifying generic classes which capture essential elements of objects or functions. In this section, we describe the manner in which the template mechanism is extended to define generic concurrent classes.

Generic concurrent classes capture common computational and interaction behavior specifications of methods of concurrent classes. They can be instantiated with user classes to associate the computational and interaction behaviors with user defined abstractions. Such classes support reusability of both computational and interaction behavior specifications. We present an example of a generic concurrent class below:

Example 5.1. *(Generic sync class).* CC++ [5] supports the notion of `sync` synchronization variables. A `sync` variable is a write-once-read-many variable. All reads to the variable are delayed until the first write has taken place. In CYES-C++, a generic class that captures the computational and interaction behavior of a `sync` variable is defined in the following manner:

```
template <class T> concurrent class sync {
public:
   virtual T & read();
   virtual void write(T &);
private:
   T data;
interaction:
   ReadSet = {read};
   WriteSet = {write};
   Interaction(WriteSet, ReadSet)
}
```

Expression `Interaction(WriteSet, ReadSet)` defines the interaction between `read` and `write` invocations as:

```
Interaction(WriteSet, ReadSet) {
    forall occ i in ReadSet {
        (WriteSet[0] < ReadSet[i])
}}
```

We omit implementations of `read` and `write` here. The generic `sync` class can now be instantiated to define different `sync` concurrent classes and objects. We show two instantiations of the `sync` generic concurrent class below:

```
sync<int> intSyncVar;
typedef sync<userClass> userClassSync;
```

Variable `intSyncVar` is an integer `sync` variable. Class `userClassSync` is a `sync` class whose contents are defined by the class `userClass`. Interaction behaviors of reads and writes to `intSyncVar` and objects of `userClassSync` are defined by the event ordering constraint expression `Interaction(ReadSet, WriteSet)`; reads are delayed until the first write has occurred. We would like to underline the fact that there are no restrictions on instantiations of the sync generic concurrent class; any user defined class can therefore behave like a `sync` primitive. ■

The template and concurrent class mechanism can therefore be used to define generic concurrent classes that capture essential concurrency, interaction, and computational attributes of concurrent classes. These generic classes can then be composed with other classes to construct concurrent classes. CYES-C++ therefore allows one to construct libraries of generic synchronization primitives that can be instantiated with user-defined classes.

6 Related work

Several concurrent programming languages have used the concept of encapsulated "object" as a basis for specifying concurrency. For instance, the concept is used in i) rendezvous-based languages such as ADA [7]; ii) approaches based on message passing such as CSP [10]; iii) approaches based on abstract data types (ADT) such as Monitors [9], ADT with path expressions [3]; and iv) actor-based approaches [1]. Further, many object-oriented concurrent programming languages have used C++ as the basis for including concurrency and synchronization. Examples of such languages are: CC++ [5], Mentat [8], Charm++ [13],COOL [4], μC++ [2], and ACT++ [11].

The different approaches to interaction specification in these languages can be categorized into three: i) languages that use traditional synchronization primitives such as locks and semaphores [4, 2], write-once-read-many variables [5], and data flow based data dependencies [8] for specifying interaction among methods. ii) Languages such as enable-based approaches [17, 22], disable based approaches [6], and behavior abstraction based approaches [12, 23, 14] that use boolean conditions to determine if a method should be executed or delayed. iii) Approaches that use regular expression [3] and temporal logic expressions for specifying interaction.

Many of the interaction specification mechanisms do not allow one to define abstractions of interaction behaviors that can be reused. Also, event ordering constraint expressions support composition operators for modular development of interactions. This forms the basis for extensibility and reusability of interaction specifications. Further, event ordering constraint expressions allow specifications of interaction among specific invocations of methods, whereas all interaction specification mechanisms specify interaction constraints for all invocations of methods. Event ordering constraint expressions therefore provide greater flexibility in terms of specifying interaction in that any interaction behavior for any invocation of a method can be specified.

There is some similarity between our notion of conditional event sets and accept states of the behavior-abstraction based languages in that both capture invocations of methods for which specific boolean conditions are true. In the case of the behavior-abstraction based languages, however, the interaction behavior of the events of accept sets is predefined. In the case of event ordering constraint expression, on the other hand, any interaction behavior can be specified for the events of the event set by defining suitable event ordering constraint expressions.

7 Summary and status

We have presented the design of a concurrent object-oriented programming language that supports extensibility and modifiability of programs as well as reusability of computational and interaction specifications. The basis for the design of the language is based on separation of specifications of computation and interaction. Separation of the two specifications allows one to extend or modify either of the components. Also, the abstractions for computation and interaction can be combined with other program composition mechanisms such as templates to construct concurrent program abstractions.

We have developed a prototype implementation for CYES-C++. We have done preliminary performance analysis of a number of simple applications (such as the N-Body problem and Gaussian Elimination algorithm). The results show that languages based on separation of concerns can be implemented efficiently. The details of the implementation and the performance analysis can be found in

[18]. Our current and future effort involves porting the current implementation to other platforms and extensive performance analysis of many large applications.

References

[1] Gul A Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, Massachusetts, 1986.

[2] Peter A. Buhr and Richard A. Strossbosscher. μC++ Annotated Reference Manual. Technical Report Version 3.7, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, June 1993.

[3] R. H. Campbell and A. N. Habermann. The Specification of Process Synchronization by Path Expressions. In *Lecture Notes on Computer Sciences*, volume 16, pages 89–102. Springer Verlag, 1974.

[4] R. Chandra, A. Gupta, and J. L. Hennessy. COOL: A Language for Parallel Programming. In *Languages and Compilers for Parallel Computing Conference*, pages 126–147. Springer Verlag, 1992.

[5] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional Parallel Programming. Technical Report Caltech-CS-TR-92-13, Cal Tech, 1992.

[6] Svend Frolund. Inheritance of Synchronization Constraints in Concurrent Object–Oriented Programming Languages. In *ECOOP '92, LNCS 615*, pages 185–196. Springer Verlag, 1992.

[7] Narain H. Gehani. *Ada: Concurrent Programming*. Prentice Hall, Englewood Cliffs, N.J., 1984.

[8] Andrew S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, 26(6):39–51, 1993.

[9] C. A. R. Hoare. Monitor: An Operating System Structuring Concept. *Communication of the ACM*, 17(10):549–557, 1974.

[10] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, 1978.

[11] D. G. Kafura and G. Lavender. Recent Progress in Combining Actor-Based Concurrency with Object-Oriented Programming. In *ACM OOPS Messenger, Proceedings OOPSLA/ECOOP 90 workshop on Object-Based Concurrent Systems*, volume 2, pages 55–58, April 1991.

[12] Dennis Kafura and Keung Lee. Inheritance in Actor based Concurrent Object-Oriented Languages. In *Proceedings ECOOP'89*, pages 131–145. Cambridge University Press, 1989.

[13] L.V. Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object-Oriented System Based on C++. In *OOPSLA '93*, pages 91–108. ACM Press, 1993.

[14] Satoshi Matsuoka. *Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming*. PhD thesis, The University of Tokyo, Japan, June 1993.

[15] 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.

[16] José Meseguer. Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming. In *Proc. 7th ECOOP'93*. Springer Verlag, 1993.

[17] Christian Neusius. Synchronizing Actions. In *ECOOP '91*, pages 118–132. Springer Verlag, 1991.

[18] Raju Pandey. *A Compositional Approach to Concurrent Programming*. PhD thesis, The University of Texas at Austin, August 1995.

[19] 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.

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

[21] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Second Edition edition, 1991.

[22] Laurent Thomas. Extensibility and Reuse of Object-Oriented Synchronization Components. In *Parallel Architecture and Languages Europe, LNCS 605*, pages 261–275. Springer Verlag, 1992.

[23] Chris Tomlinson and Vineet Singh. Inheritance and Synchronization with Enabled Sets. In *OOPSLA '89 Conference on Object-Oriented Programming*, pages 103–112. ACM Press, 1989.