International Journal of Parallel Programming, Vol. 29, No. 4, 2001

Support for Implementation of Evolutionary Concurrent Systems

Raju Pandey[1] and James C. Browne[2]

Received August 2000; revised November 2000

Concurrent programs that embed specifications of synchronizations in the body of their component are difficult to extend and modify. Small changes in a concurrent program, particularly changes in the interactions among components, may require re-implementation of a large number of components. Even specifications of components cannot be reused easily. This paper presents a concurrent program composition mechanism in which both specification and implementation of computations and interactions are completely separated. Separation of specifications and implementations facilitates extensions and modifications of programs by allowing one to separately change the implementations of computations and interactions. It also supports their reusability. The paper also describes the design and implementation of a concurrent object-oriented programming language based on this model, including a compiler for the language, and reports on the execution behavior of programs written in the language.

KEY WORDS: Software evolution; concurrent programming; object-oriented programming; inheritance; genericity.

1. INTRODUCTION

Software systems are evolutionary entities. They change during the initial development stage, and often after they have been deployed. These changes may occur due to changes in its requirements, in hardware configurations, and/or in the execution environment. Such changes often mean that new software components must be added and/or existing components must be

[1] Parallel and Dist. Computing Laboratory, Department of Computer Science, University of California, Davis, California 95616. E-mail: pandey@cs.ucdavis.edu

[2] Department of Computer Sciences, The University of Texas, Austin, TX 78702. E-mail: brown@cs.ucdavis.edu

re-defined. The fundamental principle for managing the evolution of complex software systems states that changes in a system should be proportional to the size of the changes, and *not* the size of the system. Specifically, small changes in a system should be entail modifications of only a small number of components.

In poorly designed systems, software components and relationships among them are not clearly defined. Making minor changes in such systems often requires major effort because components that are directly affected and those that are affected due to the relationships may be difficult to identify and modify. Changing the implementation may, thus, require re-implementing a large number of the components.

In this paper, we explore the ability to modify concurrent systems that are developed using the traditional concurrent programming methodologies. Concurrent systems are difficult to design and implement because several factors (such as nondeterminism, complex interactions among programs, program granularity, data partitioning, data distribution, load balancing, and target machine configurations) drive the design of a concurrent program. An efficient and correct implementation of a concurrent program involves careful analysis of these factors, interactions among them, and their representation. Ability to change implementations of concurrent systems add to the complexity of developing concurrent systems. Specifically we are interested in the following in this paper:

- *How suitable are many traditional programming methodologies for supporting evolutionary concurrent systems?* In this paper, we show that it is often difficult to change implementations of concurrent systems: changes in the implementations of a small number of components may require the implementations of a *disproportionately* large number of components. Also, concurrent program abstractions cannot be composed easily with existing program abstractions. This inhibits re-usability of the components of a program.
- *Why are such systems difficult to change? Also, what kinds of programming techniques are needed to support incremental changes in such systems?* We show that the ability to change systems depends on how computational and synchronization aspects of the systems are specified. We present a structuring scheme in which *implementations of computational and synchronization aspects are completely separated.* Separation has direct implications on the extensibility and modifiability of concurrent systems. Concurrent systems can be extended and modified by adding and modifying implementations of either computational, synchronization, or both aspects. Further, the scheme advocates a programming design methodology where

concurrent systems can be quickly constructed from existing computational and synchronization components.

- *How are computational and synchronization aspects of programs specified?* We describe a concurrent programming model that defines general mechanisms for representing computations and a declarative mechanism for specifying synchronization. We also describe the design of a concurrent object-oriented programming language, called CYES-C++, based on the above scheme. CYES-C++ supports extensibility and modifiability of concurrent programs as well as re-usability of computational and synchronization components.

This paper is organized as follows: In Section 2, we illustrate the difficulty of extending or modifying concurrent systems. We analyze the reasons for the problems, and show how some of these problems can be resolved in Section 3. We then present the details of a concurrent programming model, and how it supports implementation of evolutionary concurrent systems. Section 4 briefly describes a concurrent object-oriented programming language that is based on the idea of separation. We briefly describe the implementation details of the language in Section 5. We present the related work in Section 6. Section 7 contains concluding remarks, status of the research, and discusses our future work.

2. SUPPORT FOR EVOLUTIONARY CONCURRENT SYSTEMS

We begin by first examining the implications of changing concurrent programs.

In a majority of concurrent programming languages, the approach to implementing a concurrent program involves partitioning a problem into a set of components, each implemented as a process, task, or thread. An implementation of a component usually contains operations that implement its computations, synchronization with other components, data decomposition, and distributions and task scheduling algorithms. We now show that concurrent programs specified in this manner are difficult to change and modify: extensions and modifications in a concurrent program may require that a large number of its components be modified. We illustrate this through a simple example. We show that changes in the interaction among the components of a simple concurrent program require re-implementation of some or all of the components.

Example 1 *(Modifiability of concurrent programs).* Let examprog1 be a concurrent program containing two components: producer and consumer. producer repeatedly produces data, which are consumed by

```
examprog1(){
    channel buf;
    producer(buf) || consumer(buf);
}

producer(channel buf){
    while (TRUE) {
        info = produce();
        send(buf, info);
    }
}

consumer(channel buf){
    while (TRUE) {
        info = receive(buf);
        consume(info);
    }
}
```

Fig. 1. An example concurrent program.

consumer. We show implementations of examprog1, producer and consumer in Fig. 1. The components interact with each other through send and receive primitives over a mailbox[1] where programs can deposit and retrieve information in a FIFO manner. We assume that send is nonblocking whereas receive is blocking. Note that the synchronization operations (send and receive) are embedded within the implementations of the components.

We first consider a simple extension of examprog1 that involves adding another consumer component, for instance because consumer is slow relative to producer. Assume that data are now shared between the two consumer components *alternately*. The extended program requires modifying producer, consumer, or both components. We show one possible implementation of the extended program in Fig. 2. Here, consumer1 (Fig. 2) is a modified version of consumer. In this program, operations send and receive on sync represent the interaction between the two consumer components.

We now look at the modifiability of examprog1. We do that by defining additional constraints between producer and consumer: there are at most N unconsumed values. producer therefore must wait for consumer if there are N unconsumed values. One possible implementation is shown in Fig. 3. In the implementation, consumer sends an acknowledgment after every received message.

2.1. Conclusions

Even though examprog1 is a simple program with two components and simple computational and synchronization operations, the potential

```
consumer1(channel buf, sync) {
    myTurn = myId % 2;
    while (TRUE) {
        if (myTurn) {
            info = receive(buf);
            consume(info);
            send(sync, ack);
            myTurn = FALSE;
        } else {
            ack = receive(sync);
            myTurn = TRUE;
        }
    }
}

examprog2() {
    channel buf;
    channel sync;

    producer(buf) || consumer1(buf, sync) ||
    consumer1(buf, sync);
}
```

Fig. 2. A representation of an extended concurrent program.

```
producer1(channel buf) {
    while (TRUE) {
        info = produce();
        send(buf, info);
        count = count+1;
        if (count == N) {
            ack = receive(buf);
            count = count - 1;
        }
    }
}

consumer2(channel buf) {
    while (TRUE) {
        info = receive(buf);
        consume(info);
        send(buf, ack);
    }
}

examprog3() {
    channel buf;
    producer1(buf) || consumer2(buf);
}
```

Fig. 3. An implementation of a modified concurrent program.

implications of the changes and extensions are widespread. Specifically, they demonstrate the following weaknesses in the programming methodology:

- *Changes in the implementations of a small number of components may affect the implementations of large number of components.* For instance, a simple modification in the concurrent program results in possible re-implementation of producer and consumer. A concurrent programming language must provide mechanisms that encapsulate different components of a concurrent system so that changes in components or additions of components do not involve changing some or all existing components of the concurrent program. However, as we saw earlier, changes in concurrent programs are often visible in most components. Implementations of component programs are, therefore, not encapsulated.
- *Implementations of components cannot be reused easily.* For instance, in the three different versions of the example program, much of the behaviors of producer and consumer remains unchanged. However, we create the different versions of the components by manually copying code from one version to the other. In addition, we cannot use synchronization code easily because it is embedded *procedurally* inside component implementations.
- *Modifications in components often involve making modifications in existing source code.* Such modifications in source programs are error prone. Indeed, they are one of the major sources of bugs in concurrent programs.

There are alternative designs for the concurrent producer/consumer problem where the interactions of the producer and consumer are mediated by a shared buffer. In this design, the implementations of the producer and consumer are stable under modification of interactions through buffer. However, the implementation of the buffer must be modified to implement any new interactions. Implementing interaction control in the buffer violates fundamental tenants of distributed systems: it centralizes control of an inherently distributed set of processes. Further, it places a third part in control of independent distributed entities.

2.2. Program Composition Anomaly

More importantly, the example underlines the problem associated with constructing new concurrent program abstractions in terms of existing program abstractions.

Definition 1 *(Program composition anomaly).* The phenomenon in which the composition of program abstractions requires changes and modifications in the program abstractions themselves is called the program composition anomaly.

The program composition anomaly highlights the inability to compose concurrent program abstractions from existing program abstractions. For instance, program abstractions producer and consumer of Example 1. may not be simply composed with other program abstractions without requiring changes in them in many cases. Since programming languages use many composition mechanisms for defining abstractions in terms of other abstractions, the presence of the program composition anomaly causes breakdowns in many of these composition mechanisms. We enumerate two such cases.

2.2.1. Program Composition Anomaly in Concurrent Object-Oriented Programming Languages

There exists, in addition to the encapsulated data structure view, an active view[2, 3] of concurrent objects. In this view, a concurrent program can be associated (either explicitly[4] or implicitly[5]) with a concurrent object. The program determines the manner in which methods are executed and scheduled. The composition of the program is derived from the methods to which the object responds, the manner in which they are executed, and the synchronization mechanisms employed.

Object-oriented languages support two fundamental composition mechanisms: *aggregation* and *inheritance.* Aggregation is used to define the structure of an object in terms of its component objects. Inheritance, on the other hand, is used to extend the structure of an object. Both aggregation and inheritance can be viewed as implicit concurrent program composition mechanisms: aggregation as defining the concurrent program associated with an object as a composition of programs associated with its component objects, and inheritance as a means for extending the program composition of concurrent objects. The underlying characteristics behind the two is that of constructing new abstractions from existing abstractions. Here, it means combining, modifying, and extending concurrent programs. As we showed earlier, the program composition anomaly occurs precisely in these situations. We illustrate each through examples.

2.2.1.1. Aggregation Anomaly. The aggregation anomaly occurs when an object defines additional synchronization behavior for the methods of its component objects:

Example 2 *(Aggregation anomaly).* Assume that a concurrent class AtomicBuffer defines two methods: Read and Write. The two methods synchronize with each other while accessing common data structures of AtomicBuffer. Implementations of Read and Write contain codes for their computations and synchronization with each other.

Assume that a concurrent class TwoBuffers defines two components objects: LargeBuffer and SmallBuffer of class AtomicBuffer. In addition, it defines the following constraints on invocations of Read and Write over LargeBuffer and SmallBuffer objects: Write invocations on LargeBuffer have higher priority than Write invocations on SmallBuffer.

Since the synchronization operations of Write are embedded inside the implementation of Write in AtomicBuffer, the new synchronization behavior can be specified by re-implementing the method in AtomicBuffer, thereby requiring redefinition of AtomicBuffer.

In this example, class TwoBuffer is derived by composing two instances (LargeBuffer and SmallBuffer) of the abstraction AtomicBuffer along with additional synchronization constraint. However, such a composition requires changes in the abstractions (Read and Write).

2.2.1.2. Inheritance Anomaly. The second problem, termed the *inheritance anomaly*,[44] arises due to the diverse synchronization requirements of a class and its subclasses.

Example 3 *(Inheritance anomaly).* Let class NewBuffer extend AtomicBuffer by defining a new method PickLast (Fig. 4). Method PickLast interacts with Read and Write of AtomicBuffer. This implies that the synchronization properties of Read and Write change. Since implementations of Read and Write include synchronization code, synchronization properties of the methods can be changed only by re-implementing them. This can be achieved either by re-implementing AtomicBuffer or by re-implementing Read and Write in NewBuffer. In the latter case, implementations of Read and Write cannot be inherited in NewBuffer.

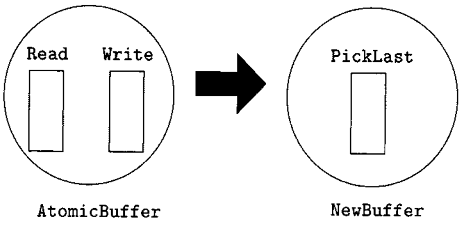

Fig. 4. The inheritance anomaly.

The inheritance anomaly is another instance of the program composition anomaly. Here, a subclass extends the program composition associated with a concurrent object either by adding new methods or by modifying inherited methods. Such extensions may require changes in the composition, which, in this case, means redefinition of the methods.

We therefore see that changes and modifications in a concurrent program often require re-implementation of some or all components. In addition, concurrent program abstractions cannot be composed easily with other program abstractions. This causes breakdown in many existing program composition mechanisms such as inheritance and aggregation.

3. THE C-YES MODEL OF CONCURRENT COMPUTATION

In this section, we describe a model of concurrent computation, called the C-YES model, that addresses the problem of concurrent program evolution. Before we describe the model of computation, we define several terms used in the rest of the paper.

Definition 1 *(Action).* An action is an identifiable operation. Actions represent the alphabet that a programmer uses for constructing a program. An action may denote a named function call or a labeled statement. For instance, operation produce denotes an action.

Definition 2 *(Program).* A program represents a set of primitive and nonprimitive actions that are combined using the constructs of a programming language.

Definition 3 *(Computation).* A computation denotes a specific execution of a program. Every program has a set of computations associated with it.

Definition 4 *(Event).* An event is an identifiable occurrence of an action in a computation. We illustrate the relationship between an action and an event through the following program:

```
for (i = 0; i < 5; i = i + 1)
        sum( );
```

In this program sum denotes an action. In a computation of the program, action sum is executed five times. Every execution of sum within the context of the computation denotes a unique event.

Notation. We represent an event in a computation by Action[Selector]. Here, the term Selector is used to uniquely identify an occurrence of Action. We use the notion of *event occurrence number* as a selector. An event occurrence number, *i*, of an event specifies that the event is the *i*th invocation of an action in a given computation. For instance, term produce[i] denotes the *i*th invocation of produce action in a computation of examprog1. Note that relative orderings in occurrence numbers merely specify the order in which invocations of an action occur; they do not suggest that executions of the invocations are serialized. For instance, it is possible for an event, say X[5], to terminate before an event, X[4], in a computation.

3.1. Resolution of Problems

We first examine the reasons for the occurrence of the program composition anomaly. There are two distinct behaviors of a component: *computational behavior* and *interaction behavior*. The computational behavior of a component specifies the operations performed during an execution of the component. For instance, the computational behavior of producer (Example 1) is to produce data. The interaction behavior of a component determines the manner in which the component affects or is affected by other components. It represents the component's semantic relationship with other components. For instance, the interaction behavior of consumer (Example 1) specifies that every invocation of consume depends on a preceding invocation of produce, representing a data dependency relationship between the actions.

The program composition anomaly arises because implementations of both—computational and interaction—behaviors of a component are embedded within an implementation of the component. This is because changes in components may induce changes in interaction behaviors of other components. Since implementations of the interaction behaviors are distributed in the implementation of the components, the changes can be effected only by re-implementing all components that contain the implementations of the interaction behavior. For instance, the computational behaviors of producer and consumer remain unchanged in the different extensions; only their interaction behaviors change. However, one must create the different versions because the interaction behaviors are embedded in the component implementations.

Our approach is based on a structuring technique for concurrent programs in which *implementations of computational and interaction behaviors are completely separated.* A concurrent program is, thus, composed from *separate* implementations of computational and interaction behaviors. The requirement for the separation highlights the orthogonality of the two behaviors. We think of the computational behavior of a component as its intrinsic property. It exists independently from the component's possible inclusions in different concurrent programs. For instance, the role of producer is to produce certain value. It is independent of the fact that it can be combined with a single consumer, multiple consumers, or even another producer. The intrinsic property—producing information—does not change. Its interaction behavior, on the other hand, is dependent on other components of a concurrent program. It should, therefore, be specified separately from the implementations of the computational behavior, and when the concurrent program is defined.

3.2. Elements of the Model

The C-YES model contains a concurrent program composition mechanism, an extended model of components, and a declarative interaction specification mechanism. We first describe the composition mechanism.

3.3. Concurrent Program Composition

Definition 5 *(Constrained concurrent program composition).* The expression

$$C = (C_1 \parallel C_2 \parallel \cdots \parallel C_n) \quad \text{where } \phi$$

specifies a concurrent program C that is composed from components C_1, C_2,..., and C_n and interaction behavior ϕ.

Components C_1, C_2,..., and C_n contain implementations of their computational behaviors only. ϕ specifies the interaction among the components. The semantics of the composition is that events of C_1, C_2,..., and C_n are concurrent by default. Hence, during an execution of C, they may occur in parallel. However, there are certain events that interact with each other. Occurrences of these events must satisfy all constraints specified by ϕ.

3.4. Representation of Component Programs

Given that implementations of component programs do not include implementations of interaction behavior, the question that needs to be answered is: how are component programs represented so that their interaction behaviors can be specified?

We construct a model of component programs by observing the execution behavior of a component: during an execution of a concurrent program, a component repeatedly performs certain actions. Occasionally, it interacts with its environment (other components) during the executions of certain actions. We call these actions *interaction points*. Interaction points of a component represent those invocations of actions where the component may need to be synchronized with other components. For instance, in CSP[6] a process represents a component. A process interacts with other processes by sending and receiving messages on communication channels. Thus, actions, send and receive, form the process' interaction points.

There are two parts of an interaction point: identity and role. The identity of an interaction point determines the action at which a component may interact. In CSP, for instance, the names of the communication channels along with the actions identify the interaction points of a process. The role of an interaction point determines the manner in which a program participates in an interaction at the interaction point. For instance, the role associated with a synchronous "receive" interaction point determines a process's behavior at the interaction point: the process is delayed until a message arrives.

In the C-YES model, the two elements of an interaction point—identity and role—are separated. Its identity is defined when the component is specified. Its role, on the other hand, is defined when the component is composed with other programs. A component program in the C-YES model is, thus, represented by its computations and interaction points. We call such components *interacting components*.

We represent the interaction points of a component implicitly: all actions on objects denoted by parameter variables are the interaction points of the component. (We assume that the parameters represent objects.) For instance, we show the interacting program representations of producer and consumer in Fig. 5. Interaction points of producer are represented by the term info.produce( ), which denotes the set of all produce events associated with producer in a computation.

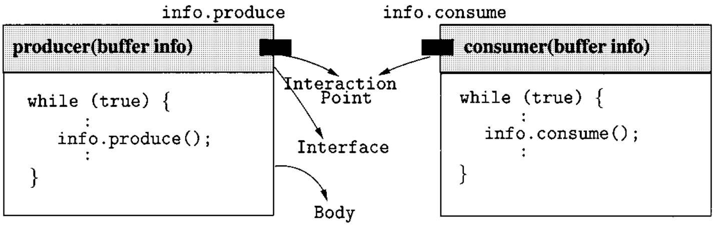

Fig. 5. Representations of interacting components.

3.5. Interaction Specification: Event Ordering Constraint Expressions

Interaction among programs in the C-YES model is specified by an algebraic expression, called the *event ordering constraint expression*. It is used to represent semantic dependencies among events 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*.

3.5.1. Primitive Event Ordering Constraint Expression

A primitive event ordering expression defines the interaction relationship between specific occurrences of two actions. It imposes constraints on execution orderings of two events. It is defined:

$$\phi = (e_1 < e_2)$$

A computation satisfies ϕ if e_1 occurs before e_2 in the computation.

3.5.2. Interaction Composition Operators

Interaction composition operators are used to combine primitive and nonprimitive event ordering constraint expressions. There are four interaction composition operators:

- And constraint operator (&&): The and constraint operator && is used for combining event ordering expressions in order to represent interactions among sets of events. It is defined:

  $$\phi = (\phi_1 \mathbin{\&\&} \phi_2)$$

  Intuitively, a computation satisfies event ordering constraint expression ϕ if it satisfies both ϕ_1 and ϕ_2.
- Or constraint operator (||): The or constraint operator || is used to incorporate nondeterminism in the orderings of events. It is defined:

  $$\phi = (\phi_1 \mathbin{||} \phi_2)$$

  Intuitively, a computation satisfies ϕ if it satisfies *at least one* of event ordering constraint expressions ϕ_1 or ϕ_2.
- Forall operator: The and constraint operator is used for combining two event ordering constraint expressions. Forall extends && in order to specify ordering constraints over sets of events. There are

two versions of forall: forall var and forall occ. We first describe forall var. Let S be a set of events and $\phi(e)$ be an event ordering constraint expression over event e. The relationship between forall and && is shown next:

$$\begin{array}{c}\text{forall var } e \text{ in } S\\ \phi(e)\end{array} = \mathop{\&\&}_{e \in S} \phi(e)$$

forall occ is similar to forall var in that it is used to enumerate over a set of events. While forall var iteratively binds a variable with the events of a set, forall occ binds a variable with the occurence number of the events.

- Exists operator: The exists operator is similar to forall in that it extends the or constraint operator over a set of events. There are two versions of exists: exists var and exists occ. We first describe exists var. Let S be a set of events and $\phi(e)$ be an event ordering constraint expression over event e. The relationship between exists and || is shown here:

$$\begin{array}{c}\text{exists var } e \text{ in } S\\ \phi(e)\end{array} = \mathop{||}_{e \in S} \phi(e)$$

exists occ is used to bind a variable with the occurrence number of events of a set.

3.6. Examples

We now present several examples. The goal here is to not only illustrate the manner in which the C-YES model can be used for specifying concurrent programs, but also to highlight the various characteristics of the model.

3.6.1. Mutual Exclusion

The simplest constraint is that of mutual exclusion between events of two sets S_1 and S_2. For events e_1 and e_2 such that $e_1 \in S_1$ and $e_2 \in S_2$, mutual exclusion between e_1 and e_2 can be represented as nondeterministic orderings of occurrences of e_1 and e_2:

$$\text{MutexEvents}(e_1, e_2) = (e_1 < e_2) \mid\mid (e_2 < e_1)$$

This relationship holds for all events of sets e_1 and e_2. Therefore,

```
MutuallyExclusive($S_1$, $S_2$) = forall var $e_1$ in $S_1$
                                      forall var $e_2$ in $S_2$
                                          MutexEvents($e_1$, $e_2$)
```

3.6.2. Producer and Consumer

Assume that we construct a concurrent program by composing components producer and consumer of Example 1. Let info.produce and info.consume respectively denote the interaction points of producer and consumer. We define an event ordering constraint expression, ConsExp, that represents the constraint that a consume event cannot occur until a corresponding produce event has terminated:

```
ConsExp = forall occ i in info.produce
              info.produce[i] < info.consume[i]
```

In this expression, term info.produce[i] denotes the ith possible invocation of produce in a computation.

3.7. Discussions

We now discuss the different aspects of the C-YES model. We show that the C-YES model supports extensibility and modifiability of concurrent programs and re-usability of both computational and interaction behavior implementations.

3.7.1. Extensibility of Concurrent Programs

In order to show that implementations of a concurrent program can be easily extended, we derive a concurrent program for examprog2 of Example 1. examprog2 is defined:

```
examprog2 = (producer || consumer || consumer) where ConsExp2
```

There are no changes in implementations of either producer or consumer. ConsExp2 implements the modified interaction behaviors of the three components. We derive it by observing that there are two sets of relationships among the events of producer and consumer (Fig. 6). The first is between *odd* events of produce and consume events of the one consumer component, and the second is between *even* events of produce and consume events of the other consumer component. Let consume1 and consume2

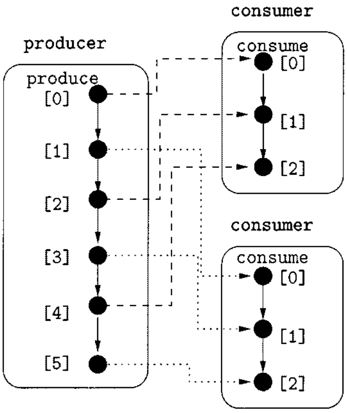

Fig. 6. Interaction relationship among events in a computation of an extended program.

denote the interaction points of the two consumer components. The following event ordering constraint expressions represents the two relationships:

$$\text{odd}(i) = (\text{produce}[2*i+1] < \text{consume1}[i])$$

$$\text{even}(i) = (\text{produce}[2*i] < \text{consume2}[i])$$

Since this relationship holds for all events of produce, ConsExp2 is defined as:

$$\text{ConsExp2} = \text{forall occ i in produce}$$

$$\text{odd}(i)\ \&\&\ \text{even}(i)$$

3.7.2. Modifiability of Concurrent Programs

We now look at the modifiability of concurrent programs. This next example highlights a program design methodology where concurrent programs can be constructed quickly from existing core components.

In this example, we derive an implementation for examprog3 of Example 1. In this program, there is additional constraint between the producer and consumer components: there can be at most N outstanding un-consumed values. An implementation of examprog3 is shown:

$$\text{examprog3} = (\text{producer}\ ||\ \text{consumer})\ \text{where ConsExp3}$$

In this program, there is a relationship—in addition to the one defined by event ordering constraint expression ConsExp1 of Example 1—between the info.produce and info.consume interaction points: a produce event cannot

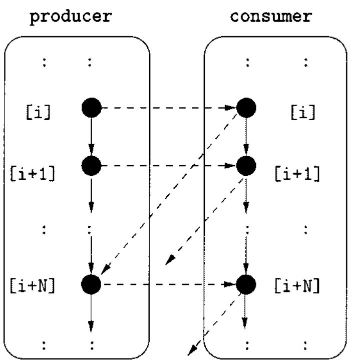

Fig. 7. Interaction relationship among events in a computation of a modified program.

occur until a consume event has occurred (see Fig. 7). ConsExp3 is defined by including the following additional constraint:

$$\text{ConsExp3} = \text{ConsExp1} \;\&\& \;\text{forall occ i in consume}\; (\text{info.consume}[i] < \text{info.produce}[i+N])$$

We observe from these examples that the C-YES model supports reusability of both computational and interaction behavior implementations.

3.7.3. Resolution of Aggregation and Inheritance Anomalies

Separation also forms the basis for the resolution of the aggregation and inheritance anomalies. In the case of the inheritance anomaly, interaction behavior of inherited methods can be extended and/or modified by defining interaction behaviors in a subclass.(2) The inheritance anomaly has been studied in great detail and many solutions(7–10) have been proposed. Most of these solutions are based on the separation of the implementations of synchronization behavior from the implementations of the methods.

4. CYES-C++

In this section, we describe the design of a concurrent object-oriented programming language, called CYES-C++. CYES-C++ is a concurrent extension of C++(11) and is based on the C-YES model.

4.1. Concurrent Class

CYES-C++ supports both intra- and inter-object concurrency. Concurrent objects in CYES-C++ are represented by defining a concurrent

```
concurrent class  Queue {
   public:
        Queue();
        ~Queue();
        void Put(char);
        char Get();
        Boolean Full();
        Boolean Empty();
   interaction:
        SeqPuts;             // serialize execution of puts
        SeqGets;             // serialize execution of gets
        DelayPutIfFull;      // delay put if queue is full
        DelayGetIfEmpty;     // delay get if queue is empty
   private:
          :
}
```

Fig. 8. Specification of a concurrent class, Queue.

class. In Fig. 8, we show an example of a concurrent class, Queue. A concurrent class is similar to C++ classes, except that it contains an additional section, called the interaction section. The interaction section of a concurrent class defines a set of event ordering constraint expressions that represent the interactions among the public methods. The semantics associated with a concurrent class specifies that all invocations of the public methods on one of its instances execute in *parallel*, except for those whose executions must satisfy *all* ordering constraints specified in the interaction section. A concurrent object, thus, is a constrained composition of method invocations and the event ordering constraint expressions.

In Fig. 8, we show four interaction constraints: (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, we represent the constraints symbolically. The constraints determine if Put and Get invocations can be executed or should be delayed. We derive event ordering constraint expressions for the constraints in Example 4.

4.1.1. Declaration of Concurrent Objects

Concurrent objects are declared by the declarator mechanism of C++. Next we show some examples of concurrent object declaration and usage:

```
Queue q;               // create a concurrent object
Queue *qptr;           // create a pointer to a concurrent object
Queue qarray[100];     // create 100 queue objects
    :
qptr = new Queue( );   // create a concurrent queue object
qptr -> Put(val);      // invoke a method on a queue object
```

4.1.2. Method Invocation

CYES-C++ supports both synchronous and asynchronous invocations of methods. In synchronous method invocation, the invoking method is blocked until the invoked method terminates. For instance, during an execution of the expression

```
q.Put(val);
```

the invoking method is blocked until Put terminates.

A method func can be invoked asynchronously in the following way:

```
par obj.func(p1, p2,..., pn)
    where evoce
```

The term evoce is an event ordering constraint expression. It represents the interaction among the interaction points of the calling method and an occurrence of func. CYES-C++ also supports a parallel iterator operator:

```
parfor (int i = 0; i < n; i++ )
    obj.func(p1, p2,..., pn) where evoce
```

This operator is similar to parfor of CC++.(5) However, invocations of methods are asynchronous in CYES-C++. The above statement terminates once all methods have been invoked. This is unlike the parfor operator of CC++ where a parfor expression terminates only after all invoked methods have terminated.

Note that unlike C++, CYES-C++ does not support the call-by-value semantics for concurrent object parameters. Instead, concurrent objects are always passed by reference.

4.2. Interaction Specification

Interaction in CYES-C++ is represented by event ordering constraint expressions. Now we describe how events, event sets and event ordering constraint expressions are expressed in CYES-C++.

4.2.1. Event Sets

Events sets form the abstraction for identifying and representing invocations of methods that interact with other method invocations. An event set is either a primitive event set or is constructed from other event sets.

CYES-C++ supports the following primitive event sets for every method M: (i) Set M denoting all invocations of M; (ii) Set M:waiting denoting all invocations of M currently waiting; (iii) Set M:running denoting all invocations of M currently executing, and (iv) set M:terminated denoting all terminated invocations of M.

Nonprimitive event sets are constructed from other event sets through the following mechanisms:

- Method parameters: Event sets can be constructed on the basis of the values of the parameters of method invocations. For instance, the expression add(2) denotes an event set containing all invocations of method add with the parameter value 2. Such event sets are useful in representing interactions among method invocations that can be distinguished by the values of their parameters. Currently we support only integer parameters.
- Conditional event sets: A conditional event set, written M:B, denotes all events of M for which the Boolean condition B is true.
- Named event sets: Event sets can be named. For instance, expression fullqueue = Put:Full( ) defines an event set fullqueue that contains all Put events for which Full( ) is true.
- Event set expressions: Event sets can be combined with other event sets through the union (+) and difference (−) operators. Hence, the expression

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

  extends the event set fullqueue to include the events of set putlast: Full( ).

Example 4 *(Interaction specification).* We now show how event sets and event ordering constraint expressions can be used for deriving expressions for the interaction section of Queue in Fig. 8.

We first define two named event ordering constraint expressions:

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

Expression Serialize orders events of set S according to their occurrence numbers. Expression Priority gives events of S1 higher priority over events of S2. We now define four event sets:

AddQ = Put	RemQ = Get
QEmpty = Get:Empty( )	QFull = Put:Full( )

Set AddQ contains all Put events. Set RemQ contains all Get events. Set QEmpty contains all Get events for which the queue is empty. Similarly, set QFull contains all Put events for which the queue is full.

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

```
SeqPuts = Serialize(AddQ)
SeqGets = Serialize(RemQ)
DelayPutIfFull = Priority(AddQ, QEmpty)
DelayGetIfEmpty = Priority(RemQ, QFull)
```

4.3. Inheritance

In this section, we examine what it means to extend a concurrent class. We also present a model of inheritance that specifies the manner in which the interaction behavior of a method can be extended in subclasses.

The model is derived from the idea of representing interactions as semantic dependencies among methods. A class may extend its superclasses by adding new methods, by modifying the existing methods, and by defining new interaction behaviors among the methods. These modifications engender additional semantic dependencies among methods. In the C-YES model, since semantic dependencies are represented by defining ordering relationships among events, changes in interaction behaviors of methods imply additional ordering relationships among the methods. The and constraint operator && precisely captures such additions of relationships. Formally, let class C and class S define event ordering constraint expressions ϕ_c and ϕ_s respectively for representing the interaction behavior of the methods. Interaction behavior of the methods of class S is defined by event ordering constraint expression ϕ:

$$\phi = \phi_s \,\&\&\, \phi_c \tag{1}$$

We now present several examples that illustrate the ways in which concurrent classes can be extended.

Example 5 *(Method addition).* In this example, we show how interaction behaviors of methods of a class can be extended. Let readlast-queue be a subclass of class Queue:

```
concurrent class ReadLastQueue: public Queue {
public:
     char GetLast( );
interaction:
          :
}
```

Method GetLast retrieves the last element of a Queue object. It interacts with both Put and Get of Queue: Invocations of GetLast must wait for invocations of Put if the Queue is empty. Similarly, invocations of Put must wait for invocations of GetLast if the Queue is full.

We can represent the interactions between the added method and the inherited methods by adding invocations of GetLast in the named event sets of Queue:

```
RemQ = RemQ + GetLast
QEmpty = QEmpty + GetLast:Empty( )
```

ReadLastQueue inherits both computational and interaction behavior of Queue. In addition, the event ordering constraint expression specified in Queue over the named event sets apply to the invocations of GetLast as well. DelayGetIfEmpty in ReadLastQueue ensures that both Get and GetLast events are delayed if the queue is empty. Further, SeqPuts ensures that Get and GetLast events are serialized.

Example 6 *(Concurrent object state partitioning).* In this example we show how we can use event sets to model states and partitioning of states in inherited classes. Let queueone be a subclass of Queue:

```
concurrent class QueueOne: public Queue {
public:
     void GetTwo(char val[]);
interaction:
     SyncQueueOne
          :
}
```

Method GetTwo accesses two elements of a 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 partitions the partially filled state into two: queue with one item, and queue with more than one item. We can model state partitions by defining new event sets. Let method One( ) return true if a QueueOne object contains one item. The event ordering constraint expression to represent the interaction between GetTwo and events of AddQ is defined as follows:

```
QueueOneObject = GetTwo:One( )

QEmpty = QEmpty + GetTwo:empty( )

RemQ = RemQ + GetTwo

SyncQueueOne = WaitWhile( AddQ, QueueOneObject )
```

The event ordering constraint expressions of Queue apply to invocations of GetTwo as well.

4.4. Generic Concurrent Classes

C++ provides the template mechanism for specifying generic classes. Templates allow one to capture essential elements of objects or functions. In this section, we describe the manner in which the template mechanism can be 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 specific computational and interaction behaviors with the classes. Such classes support re-usability of both computational and interaction behavior specifications. Next, we show an example of a generic concurrent class.

Example 7 *(Generic sync class).* CC++[5] supports the notion of sync synchronization variables. A sync variable is a write-once variable. All reads to the variable are delayed until the first write has taken place. We define a generic sync class in the following manner:

```
template <class T> concurrent class Sync {
public:
      virtual T & Read( );
      virtual Write(T &);
private:
      int wrcnt;
      T data;
interaction:
      ReadSet = Read
      WriteSet = Write
      Interaction(WriteSet, ReadSet)
}
```

Interaction(WriteSet, ReadSet) is defined:

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

Methods Read and Write are defined:

```
template<class T> T & sync<T>::Read( ) {
      return(data);
}
template<class T> T & sync<T>::Write(T &val) {
      if (wrcnt++ > 1) error( ); else data = val;
}
```

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 userClass. Interaction behaviors of reads and writes to intSyncVar and objects of userClassSync are defined by 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.

We therefore observe that separation of specifications computational and interaction behaviors have implications on language design as well. Abstractions of computational and interaction behaviors can be composed in many ways, some by extending existing program composition mechanisms, and some by defining novel program composition mechanisms.

5. IMPLEMENTATION

We have implemented a distributed version of CYES-C++. The current CYES-C++ implementation runs on a network of IBM RS/6000 workstations. It supports creation and distribution of concurrent objects both on local and distributed nodes. In addition, it supports both synchronous and asynchronous method invocations on local and remote objects, and event ordering constraint expressions containing forall, &&, and primitive event ordering constraint expressions. The current implementation uses the Nexus thread package(12) for implementing multi-threading, intra-machine synchronization, and communication.

The implementation identifies each method invocation as an event, and maintains data structures corresponding to the default and user-defined events sets. Inheritance of interaction specifications is implemented by propagating the contents of the event sets and interaction specifications in subclass implementations. We implement a primitive event ordering constraint ($a_1 < a_2$) by generating P and V operations on event-specific semaphores that ensure that the events occur in right order. More complex operators (such as forall) are implemented by associating an array of semaphores with each event pair and iterating over the semaphores. Since each event ordering constraint expression may iterate over infinitely many events, the implementation manages the semaphore arrays through a pooling mechanism.

We have conducted several experiments that evaluate both macro-level and micro-level performance of the current implementation. For details of the experiments and results see Pandey.(13) The experiments indicate that for several applications (such as Gaussian algorithm for matrix multiplication, Barnes and Hut algorithm(14) and parallel *pi* evaluation algorithm(15)) achieves speed ups comparable to the speed ups obtained by native implementation in Nexus. Further, the overhead of implementing CYES-C++ on top of Nexus is low.

6. RELATED WORK

We divide the literature survey into two parts: (i) concurrent program composition methodology, and (ii) synchronization mechanisms. While there have been several extensions of existing programming languages, such as C, C++, SmallTalk and Java, our focus in the section is more on the mechanisms used for defining concurrent programs, and less on specific concurrent programming languages.

6.1. Concurrent Program Composition Methodology

There has been extensive work done in the area of concurrent programming. Most of this work has focused on developing methodologies, languages, and tools for implementing concurrent programs. In many of these approaches, implementations of computations and synchronization are embedded within the implementation of components. Our approach differs from these approaches in how concurrent programs are composed. An extensive survey of the various concurrency mechanisms can be found [see Pandey;(13) Andrews and Schneider;(16) Bal *et al.*;(17) and Skillicorn and Talia.(18)]

The notion of separation has been proposed in various forms. Indeed, the motivation in all of these approaches has been to use separation of different elements for re-usability. The work closest to our approach is in the area of software architectures(19, 20) and interconnection languages,(21) which includes languages such as Wright,(22) UniCon,(23) Darwin(24, 25) and Polylith.(26) In these languages, a component is defined in terms of computations and a set of input and output ports. A component reads from and writes to its input and output ports respectively. A concurrent program is specified by *binding* input and output ports of components. Interconnection languages, therefore, separate the *structure* (port bindings) from components. These approaches differ in abstractions that they associate with ports and bindings and their focus. For instance, focus of Darwin and Polylith is on providing high-level concurrent programming models.

Wright, on the other hand, focuses on analysis of programs. Also, Darwin and Polylith support fixed sets of interaction; whereas Wright and UniCon provides mechanisms for defining many kinds of component interactions through connector mechanisms.

Our proposed approach is similar in that synchronization code is separated from components. However, it differs from the interconnection languages in two ways. First, there are no constraints on the nature of ports; ports can be any objects. Second, synchronization behaviors of operations on ports are specified explicitly, as opposed to interconnection languages where it is implicit in the semantics of the operations. Our approach therefore is more general than interconnection languages. An element missing from our approach is an explicit language mechanism for representing the interconnection structure of a concurrent program. We plan to add it to our proposed language.

Lopes and Kiczales(27) also use the notion of separation in developing an object-oriented programming language, called D. D includes aspects for controlling concurrency and dataflow between different interfaces. CYES-C++ differs from D in how synchronization is represented and integrated with other abstractions of the programming language. Separation of implementation of computational and interaction behaviors has been proposed for the resolution of the inheritance anomaly.(28) However, focus here has mostly been on resolving a specific instance of the program composition anomaly. It has not been studied within the general context of concurrent program composition. Svend and Agha(29) also use the notion of separation of implementations of object and coordination constraints in order to define a distributed coordination structure. However, the focus here is on re-usability of object and coordination constraints, and not on the modifiability and extensibility of concurrent programs in general.

Foster(30) also introduces the notion of separation of implementations of architectural elements from task implementations in order to support re-usability of implementations of the architectural specifications, and portability of concurrent programs. However, in the proposed approach, specifications of synchronization is not separated from computations.

6.2. Synchronization Mechanisms

We classify synchronization mechanisms into two types: *primitive-based* and *declarative* approaches. In primitive-based approaches, the semantics of a synchronization primitive associates predefined execution orderings among the invocations of the primitive. Some examples of synchronization primitives are semaphores,(31, 32) write-once-read-many variables,(5) data flow based data dependencies,(33) signal variables, enable-based approaches,(9, 34–38) disable based approaches,(39) and behavior abstraction based approaches.(7, 40) In our approach, there are no synchronization primitives. Also, there are no fixed ordering relationships. Synchronization is specified by defining ordering relationships among operations of components explicitly. Such a specification allows us to create, in many cases, precise ordering relationships among operations of components. Further, we support abstractions for defining interaction behaviors. The abstractions can be modified and extended in isolation from other abstractions. Also, they can composed with other computational abstractions in many different ways to construct powerful program abstractions.

An example of a declarative mechanism is Path Expression.(41) Event ordering constraint expressions differ from Path Expressions in that they are used to specify the ordering constraints that must be satisfied. Path expressions, on the other hand, are used to specify the valid sequences of operations through a regular expression. Further, Bloom(42) shows that path expressions do not adequately support modular development of interaction specifications because path expressions do not contain general mechanisms for directly representing states of objects, and for specifying interactions that depend on the states. States in event ordering constraints expressions can be easily captured through event sets.(43)

7. CONCLUSIONS

Concurrent programs are difficult to extend and modify in concurrent programming approaches that do not separate implementations of computations and interactions. More importantly, concurrent programs cannot be composed easily from existing program abstractions. Such compositions may often require changing the abstraction. Also, since programming languages use composition mechanisms for defining abstractions in terms of other abstractions, the inability to construct new program abstractions from existing program abstractions causes breakdowns in many of these composition mechanisms. Two examples of such breakdowns occur during definition of concurrent objects through the aggregation and inheritance mechanisms.

Concurrent programs can be easily modified and extended if implementations of both computational and interaction behaviors are separated. Separation supports encapsulation of implementations of both computational and interaction behaviors. It *localizes* the effects of changes in a concurrent program to specific implementations of computational and interaction behaviors. Further, implementations of both computational and interaction behaviors can be reused. In addition, implementations of computational and interaction behaviors can each be represented as separate abstractions.

These abstractions can be combined with other programming language composition mechanisms such as aggregation, inheritance, and genericity to construct new and powerful concurrent programming abstractions.

REFERENCES

1. Gregory R. Andrews, *Concurrent Programming*, The Benjamin/Cummings Publishing Company, Redwood City, California (1991).
2. Raju Pandey and James C. Browne, A Compositional Approach to Concurrent Object-Oriented Programming, *IEEE Int'l. Conf. Computer Lang.*, IEEE Press (May 1994).
3. Chris Tomlinson and Mark Scheevek, Concurrent Object Oriented Programming Languages, In Kim and F. H. Lochovsky (eds.), *Object Oriented Concepts, Databases, and Applications*, ACM Press, pp. 79–124 (1989).
4. Denis Caromel, Toward a Method of Object-Oriented Concurrent Programming, *Comm. ACM*, **36**(9):90–102 (September 1993).
5. K. Mani Chandy and Carl Kesselman, Compositional C++: Compositional Parallel Programming, Technical Report Caltech-CS-TR-92-13, Cal Tech (1992).
6. C. A. R. Hoare, Communicating Sequential Processes, *CACM*, **21**(8):666–677 (1978).
7. Dennis Kafura and Keung Lee, Inheritance in Actor Based Concurrent Object-Oriented Languages, *Proc. ECOOP'89*, Cambridge University Press, pp. 131–145 (1989).
8. S. Crespi Reghizzi and G. Galli de Paratesi, Definition of Reusable Concurrent Software Components, *ECOOP '91*, Springer-Verlag, pp. 148–165 (1991).
9. Laurent Thomas, Extensibility and Reuse of Object-Oriented Synchronization Components, *Parallel Architecture and Languages Europe*, *LNCS 605*, Springer Verlag, pp. 261–275 (1992).
10. Chris Tomlinson and Vineet Singh, Inheritance and Synchronization with Enabled Sets, *OOPSLA '89 Conf. Object-Oriented Programming*, ACM Press, pp. 103–112 (1989).
11. Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, 2nd ed. (1991).
12. I. Foster, C. Kesselman, and S. Tuecke, The Nexus Approach to Integrating Multithreading and Communication, *J. Parallel and Distributed Computation* (1996).
13. Raju Pandey, A Compositional Approach to Concurrent Programming, Ph.D. thesis, University of Texas at Austin, August (1995).
14. J. Barnes and P. Hut, A Hierarchical $O(N \log N)$ Force Calculation Algorithm, *Nature*, p. 324 (1986).
15. William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, Massachusetts (1994).
16. G. R. Andrews and F. B. Schneider, Concepts and Notations for Concurrent Programming, *ACM Computing Surveys*, **15**(1):3–43 (1983).
17. H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, Programming Languages for Distributed Computing Systems, *ACM Computing Surveys*, **21**(3):259–321 (1989).
18. D. B. Skillicorn and D. Talia, Models and Languages for Parallel Computation, *ACM Computing Survey*, **30**(2):123–169 (June 1998).
19. N. Medvidovic, A Classification and Comparison Framework for Software Architecture Description Languages, Technical Report UCI-ICS-97-02, University of California, Irvine, California (1997).
20. M. Shaw and D. Garlan, *Software Architecture: Perspectives on An Emerging Discipline*, Prentice Hall (1996).

21. J. Bishop and R. Faria, Languages for Configuration Programming: A Comparison, Technical Report UP CS 94/04, University of Pretoria, South Africa (1994).
22. R. Allen and D. Garlan, Beyond Definition/Use: Architectural Interconnection, *Int'l. Workshop on Interface Definition Languages*, *SIGPLAN*, **29**:35–45 (1994).
23. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, Abstractions for Software Architecture and Tools to Support Them, *IEEE Trans. Software Engng.*, **21**(4):314–335 (April 1995).
24. J. Magee, N. Dulay, and J. Kramer, Structuring Parallel and Distributed Programs, *Software Engng. J.*, **8**(2):73–82 (1993).
25. J. Magee, N. Dulay, and J. Kramer, Regis: A Constructive Development Environment for Distributed Programs, *Distributed Syst. Engng. J.*, **1**(5):304–312 (1994).
26. J. Purtilo, The Polylith Software Bus, *ACM Trans. Progr. Lang. Syst.*, **16**(1):151–174 (1994).
27. C. V. Lopes and G. Kiczales, D: A Language Framework for Distributed Programming, Technical Report SPL97-010, Xerox Palo Alto Research Center (February 1997).
28. Satoshi Matsuoka, Keniro Taura, and Akinori Yonezawa, Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages, *OOPSLA'93*, ACM SIGPLAN, ACM Press, pp. 109–126 (1993).
29. Svend Frolund and Gul Agha, A Language Framework for Multi-Object Coordination, *Proc. ECOOP'93*, pp. 346–360 (1993).
30. Ian T. Foster, Information Hiding in Parallel Programs, Technical Report MCS-P290-0292, Argonne National Laboratory (1992).
31. 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).
32. R. Chandra, A. Gupta, and J. L. Hennessy, COOL: A Language for Parallel Programming, *Languages and Compilers for Parallel Computing Conference*, Springer Verlag, pp. 126–147 (1992).
33. Andrew S. Grimshaw, Easy-to-Use Object-Oriented Parallel Processing with Mentat, *IEEE Computer*, **26**(6):39–51 (1993).
34. D. Dechouchant, S. Krakowiak, M. Meyesmbourg, M. Riveill, and X. Rousset de Pina, A Synchronization Mechanism for Typed Objects in a Distributed Systems, *Workshop on Object-Based Concurrent Progr.*, ACM SIGPLAN, ACM, pp. 105–107 (September 1989).
35. Narain H. Gehani, Capsules: A Shared Memory Access Mechanism for Concurrent C/C++, *IEEE Trans. Parallel and Distributed Systems*, **4**(7):795–810 (July 1993).
36. J. E. Grass and R. H. Campbell, Mediators: A Synchronization Mechanism, *Sixth Int'l. Conf. Distributed Computing Systems*, pp. 468–477 (1986).
37. Ciaran McHale, Bridget Walsh, Seán Baker, and Alexis Donnelly, Scheduling Predicates, *Object-Based Concurrent Computing Workshop*, *ECOOP'91*, *LNCS 612*, Springer Verlag, pp. 177–193 (1991).
38. Christian Neusius, Synchronizing Actions, *ECOOP '91*, Springer Verlag, pp. 118–132 (1991).
39. Svend Frolund, Inheritance of Synchronization Constraints in Concurrent Object–Oriented Programming Languages, *ECOOP '92*, *LNCS 615*, Springer Verlag, pp. 185–196 (1992).
40. Satoshi Matsuoka, *Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming*, Ph.D. thesis, The University of Tokyo, Japan (June 1993).
41. R. H. Campbell and A. N. Habermann, The Specification of Process Synchronization by Path Expressions, *Lecture Notes on Computer Sciences*, Springer Verlag, Vol. 16, pp: 89–102 (1974).

42. Toby Bloom, Evaluating Synchronization Schemes, *Proc. Seventh Symp. Oper. Syst. Principles*, ACM, pp. 24–32 (1979).
43. R. Pandey and James C. Browne, Support for Extensibility and Reusability in Concurrent Object-Oriented Programming Languages, *Proc. Int'l. Parallel Processing Symp.*, IEEE, pp. 241–248 (1996).
44. Satoshi Matsuoka and Akinori Yonezawa, Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, *Research Directions in Object-Based Concurrency*, MIT Press, Cambridge (1993).