A Compositional Approach to Concurrent Object-Oriented Programming

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

Abstract

This paper presents a model of concurrent object-oriented programming in which specification of computational behavior is separated from specification of interaction behavior of methods. It will be shown that this compositional approach to concurrent programming avoids some of the conceptual difficulties that have plagued the integration of concurrency and object-oriented models of programs. The compositional approach to concurrent object-oriented programming leads to declarative and incremental specification of interaction behavior and thus, to object/method definitions that can be readily adapted to different parallel execution environments. The approach supports inheritance of both method and synchronization specifications. It will be shown that compositional programming avoids the so-called "inheritance anomaly."

1 Introduction

Object-orientation [25] and concurrency are perhaps the two most significant current topics in programming language research. Many programming languages that integrate concurrency and object-orientation have been proposed. Some of these are reviewed in section 4. It does not appear, however, that there has been a fully consistent and coherent integration of the two. An example of the difficulty is the problem arising out of the interplay between inheritance and interaction among concurrent methods, the so-called *inheritance anomaly* [16]. An opinion [29] has been expressed that the inheritance anomaly is very difficult. Indeed, many concurrent object-oriented programming languages either limit the support for inheritance or do not support inheritance at all.

Also, many previous approaches integrating concurrency with object-orientation have been based on models that extend sequential semantics by the addition of procedural constructs for concurrent thread creation and synchronization. These models thus embed procedural representation of interactions among method invocations in specifications for computations. It appears to us that a successful integration of object-orientation and concurrency will have to be based on a model that separates specification of computation from specification of interaction.

This paper presents a compositional approach to concurrent object-oriented programming. The model given here is an extension and application of the C-YES model [20] to object-oriented programming. The C-YES model provides a general framework for defining the concurrent composition of programs. It has two major features: i) the computational and interaction behavior specifications of programs are completely separated, and ii) interactions among programs are specified by algebraic expressions over *events*, i.e., over specific occurrences of operations. An important property of the interaction specification mechanism is compositionality in that it allows one to divide the task of defining global interaction among a set of programs into many local interactions, develop specifications for the local interactions separately, and combine them.

A concurrent object in our model is a composition of two separate entities: methods, which contain specifications of their computational behaviors, and event ordering constraint expressions, which define interactions among the methods. The semantics of the composition associates a concurrent program with the object. The concurrent program determines the manner in which methods are accepted and scheduled for execution. Also, inter-object concurrency and interaction are represented by a method invocation mechanism which subsumes traditional synchronous and asynchronous method invocation mechanisms.

Interactions among methods add an extra dimension to the specifications of classes. In this paper, we examine the manner in which changes in the composition of objects of a concurrent class (due to additions and changes in its methods) affect the additional component. The focus here is to support inheritance of interaction specifications, and to minimize changes in interaction specifications if complete reusability cannot be achieved. We show that event ordering constraint expressions support inheritance of in-

In proceedings of the "International Conference on Computer Languages 1994"

teraction expressions by allowing one to i) incrementally add interactions that arise due to the additions of new methods, and ii) extend interactions among existing methods incrementally. Also, changes in interactions are localized to event ordering constraint expressions that involve changed methods and other interacting methods. Since specifications of computational and interaction behaviors are completely separated, the inheritance anomaly ceases to be a problem in our model. We analyze the problem in detail and discuss how they are resolved in our model.

We also examine the nature of interaction behavior specifications in class definitions and method invocations. We analyze the manner in which definitions of interaction specifications affect the reusability of class and interaction specifications. We observe that interaction behaviors of method invocations can be decomposed into two kinds of behaviors: i) *universal interaction behaviors* which represent those interaction behaviors that are common across all possible invocations of a method, and ii) *specialized interaction behaviors* which are specific to method invocations. We show that such a partitioning of interaction behaviors supports the design of concurrent classes that can be shared across many applications. We also show that event ordering constraint expressions can be used to specify such incremental specifications of interactions.

The balance of the paper is organized as follows: In section 2, we present a brief description of the composition model that forms the basis of our approach to concurrent object-oriented programming. Section 3 contains the details of our approach. It is divided into four parts: first, we present a compositional model of concurrent objects and define the semantics associated with the composition. We also present a method invocation mechanism for specifying interactions among independent objects. Next, we examine the manner in which changes in the composition of objects affect the interaction specification component. We then present the notion of universal and specialized interaction behaviors. Finally, we analyze the inheritance anomaly and discuss its resolution in our approach. In section 4, we present a survey of related research. We make concluding remarks in section 5.

2 The C-YES model

In most existing programming languages, a concurrent program is defined by specifying a set of component programs. Specification of each component program contains specification of operations that determine its computational behavior, as well as synchronization operations that determine its interaction behavior. The concurrent programming methodology is not modular since component programs must be constructed with other components in view. Also, it is difficult to modify and extend a concurrent program, since changes in the composition may require changes in its components.

The C-YES model [20] takes an alternate approach to composing a concurrent program from component programs. It is based on the observation that interactions among programs arise only when programs are composed in parallel, and hence should be defined when such a composition is defined. A concurrent program is, thus, composed from two separate entities: i) specifications of components defining computational behaviors, and ii) specifications of interaction behaviors. Formally, the constrained concurrent composition

$$C = (C_1 \parallel C_2 \parallel \ldots \parallel C_n) \texttt{ where } \phi_c \tag{1}$$

specifies a program C such that during C's execution, components $C_1, C_2, \ldots$, and C_n execute in parallel. Programs $C_1, C_2, \ldots$, and C_n contain only specifications of their computational behaviors. Interactions among the programs are defined *separately* by an expression ϕ_c.

2.1 Interaction behavior specification

Interaction among programs in the C-YES model is specified by an algebraic expression, called *event ordering constraint expression*. (We occasionally use the term *interaction expressions* for event ordering constraint expressions in this paper.) It is used to represent semantic dependencies among events of programs by specifying execution orderings — deterministic or nondeterministic — among the events. An event is a specific occurrence of an operation during a program execution. An event ordering constraint expression is constructed from a set of primitive ordering constraint expressions and a set of interaction composition operators. A primitive ordering constraint expression represents interaction between two events, whereas the interaction composition operators are used to represent nondeterministic interactions as well as interactions among sets of events.

The primitive event ordering constraint expression

$$p \Rightarrow (a \otimes b)$$

specifies a conditional ordering relationship between two events a and b. The binary operator $\owedge$ is used to combine relationships. For instance, expression $((a \otimes b) \owedge (c \otimes d))$ combines two ordering relationships, one between events a and b, and the other between events c and d. The binary operator $\ovee$ is used to model nondeterministic relationships among events. For instance, expression $((a \otimes b) \ovee (b \otimes a))$ specifies two ordering relationship, one between a and b and the other between b and a. Either of the two is a valid relationship between a and b. (Note that this expression models mutual exclusion between a and b.) Operators `forall` and `exists` respectively are extensions of $\textcircled{\wedge}$ and $\textcircled{\vee}$ in that they allow one to define $\textcircled{\wedge}$ and $\textcircled{\vee}$ relationships among sets of events.

2.2 Program representation in C-YES model

In a concurrent environment where many programs are executing in parallel, a program may perform certain operations and, occasionally, interact with its environment at certain points of executions, called *interaction points*. In the C-YES model, such programs are represented using a mechanism called *interacting blocks*. The notion of interacting blocks extends the notion of blocks or functions of sequential programming languages to admit the "interacting" nature of concurrent programs. Interacting blocks are used to represent both computational behavior as well as interaction points of a program.

The interface of an interacting block m is $m(p_1, \ldots, p_x;\ ip_1, \ldots, ip_y)$, where variables p_1, p_2, $\ldots$, and p_x parameterize m's computational behaviors, and interaction parameters ip_1, ip_2, $\ldots$, and ip_y capture those execution points where m may interact with other programs. In addition to the specification of the computational behavior, the implementation of m contains mappings between the interaction parameters and interacting events of m.

An interaction point is mapped to a unique event, since events form the basis for specifying interactions in the C-YES model. There are two ways in which the mapping between interaction parameters and events can be specified. In the first, the interaction points of m are derived *implicitly* from parameter variables p_1, p_2, $\ldots$, and p_x: all method invocations on objects denoted by these variables are the interaction points of m. Here, each parameter variable also denotes a set of method invocations. We use this approach for defining interaction points in this paper. In the other approach [20], the mappings between the interaction parameters and the events are specified by *explicitly* labeling invocations of methods and/or compositions of methods.

3 A compositional approach

We now present a compositional approach to concurrent object-oriented programming. The approach here is to model the behavior of an object as well as a collection of objects as a composition of two separate entities, namely specifications of computational and interaction behaviors. This allows us to examine the manner in which components of the composition are affected due to changes in the composition. We also analyze relationships among different composition mechanisms in order to determine reusability of both computational and interaction behavior specifications.

3.1 Concurrency and interaction among objects

Objects provide a natural basis for modeling entities of applications. Such entities exist independently, and many allow multiple activities to occur simultaneously. The notion of concurrency, both within entities and among different entities, exists naturally and can be modeled in object-oriented programming languages through inter- and intra-object concurrency. We use the concurrent composition mechanism of the C-YES model to represent concurrency and interaction both among and within objects.

3.1.1 Intra-object concurrency and interaction: In a concurrent environment, many methods can be invoked on an object in parallel. An object therefore must support an execution environment where many invocations of methods execute in parallel and interact while accessing common data structures and resources. The execution environment determines the manner in which method invocations are accepted and scheduled for execution. It can either be defined explicitly as in Mediators [10], or be determined implicitly from the nature of synchronization specification mechanisms as well as the mechanisms used for the implementation of objects.

In our object model, the execution environment associated with an object is derived *implicitly* from the composition of the object. Objects of a class in our model are represented by the composition of two separate entities, namely computation and interaction behavior specifications. The semantics of the composition is specified in terms of a concurrent program. The concurrent program and the manner in which interaction behavior specifications are evaluated determine the behavior of the execution environment. Formally, let the tuple $\langle M, \Phi \rangle$ define the composition of objects of a concurrent class C. Here, M is a set of methods m_1, m_2, $\ldots$, and m_n, and Φ is a set of event ordering constraint expressions ϕ_1, ϕ_2, $\ldots$, and ϕ_l. Methods in M are based on interacting blocks (see section 2.2) and contain specifications of only their computational behaviors and interaction points. We use the term *method event* to denote a specific invocation of a method.

Notation: Let the term (m_i, j) denote the method event associated with the j^{th} invocation of method m_i.

The semantics associated with the program composition is specified by defining a concurrent program which is derived from the specification of the composition. Let symbol $\mathcal{P}(O)$ denote the concurrent program associated with an object O of C. The program structure of $\mathcal{P}(O)$ is specified by the following expression:

$$\mathcal{P}(O) = \left(\mathop{\|}_{\substack{k \in \{1 \ldots \infty\} \\ m_j \in M}} (m_j, k) \right) \texttt{ where } \left(\bigotimes_{\phi_k \in \Phi} \phi_k \right) \quad (2)$$

The above expression specifies that all invocations of different methods of O execute in *parallel*, except for those whose execution orderings must satisfy *all* ordering constraints imposed by the expressions in Φ. Unlike most approaches, where concurrency is added to a sequential object, our approach is to start with a model where concurrency is a fundamental aspect of the model. The reason is that the composition of an object can be viewed in terms of defining concurrency and interaction relationships among invocations of methods of the object. In the absence of any knowledge about the application domain, concurrency is the fundamental relationship among method events since it captures semantic independence among them. Interactions, on the other hand, represent semantic dependencies (such as data dependency, data consistency, and priority) among method events, and hence are specified explicitly.

In addition to the implementation mechanisms that are used to create and schedule different execution threads for method events, the behavior of the execution environment of an object is also driven by the manner in which event ordering constraint expressions are evaluated. We evaluate different event ordering constraint expressions by constructing boolean *interaction constraints*, and by ensuring that the interaction constraints remain true during the life time of an object. Let $\mathcal{B}(\phi)$ denote the interaction constraint associated with event ordering constraint expression ϕ. The event ordering constraint expressions are transformed into corresponding interaction constraints in the following manner:

For a primitive event ordering constraint expression, the boolean constraint expression is:

$$\mathcal{B}(p \Rightarrow (a \otimes b)) \quad = \quad p \Rightarrow (a \mapsto b) \qquad (3)$$

In the above, $(a \mapsto b)$ is true if event a is executed before event b. This specifies that if boolean condition p is true, there is an execution ordering relationship between method events a and b. However, if p is false, there are no ordering relationships between a and b. Operationally, the execution environment keeps the interaction constraint invariant by ensuring that if p is true, event b is delayed until a has terminated.

For interaction expressions that include the interaction composition operators, the corresponding interaction constraints are:

$$\mathcal{B}(\phi_1 \owedge \phi_2) \quad = \quad (\mathcal{B}(\phi_1) \wedge \mathcal{B}(\phi_2)) \qquad (4)$$

$$\mathcal{B}(\phi_1 \ovee \phi_2) \quad = \quad (\mathcal{B}(\phi_1) \vee \mathcal{B}(\phi_2)) \qquad (5)$$

$$\mathcal{B}(\texttt{forall } a \texttt{ in } A\texttt{:}\phi) \quad = \quad \langle \forall a : a \in A : \mathcal{B}(\phi) \rangle \qquad (6)$$

$$\mathcal{B}(\texttt{exists } a \texttt{ in } A\texttt{:}\phi) \quad = \quad \langle \exists a : a \in A : \mathcal{B}(\phi) \rangle \qquad (7)$$

The interpretation of expressions that define relationships among infinite sets is done incrementally by evaluating the boolean interaction constraint incrementally, and by ensuring that the interaction constraint remains true over the lifetime of an object.

Our approach to defining interaction behavior of methods is also compositional in that the representation of interaction behaviors is composed from a set of event ordering constraint expressions, each representing certain semantic dependencies. The power of event ordering constraint expressions stems from the ability to decompose interaction behaviors of methods into a set of local interaction behaviors. The local interaction behaviors can each be represented by event ordering constraint expressions, and then combined with suitable interaction composition operators to represent the global interaction behavior. One of the implications of the modularity property of event ordering constraint expressions is that it allows one to change interaction behavior of methods by changing only the relevant and local interaction expression. Also, event ordering constraint expressions support reusability of interaction behavior specifications since interaction expressions can be derived from expressions that are predefined, or are defined in superclasses and are inherited (see section 3.2). Below we present a number of examples that underline our composition mechanism and the properties of the interaction specification mechanism:

Example 3.1 *(Mutual exclusion)* Assume that m_1 and m_2 are methods. The following interaction expression represents the synchronization constraint that invocations of m_1 and m_2 be mutually exclusive:

$$\begin{array}{l} \texttt{MutualExclusion(}m_1, m_2\texttt{) =} \\ \qquad \texttt{forall } e_1 \texttt{ in } m_1\texttt{:} \\ \qquad\quad \texttt{forall } e_2 \texttt{ in } m_2\texttt{:} \\ \qquad\qquad (e_1 \otimes e_2) \ovee (e_2 \otimes e_1) \end{array}$$

Here, terms m_1 and m_2 denote sets of events, each containing all possible invocations of the method. Variables e_1 and e_2 iterate over the two sets in order to specify interaction relationships among the events of the sets.

Example 3.2 *(Concurrent object composition)* Let the tuple $\langle M, \Phi \rangle$ define a class `queue`. Here, M = {`put`, `get` }. The interaction behaviors of the methods are specified by the following constraints: i) `put` and `get` invocations are sequential, ii) `put` events are delayed if the queue is full, iii) `get` is delayed if the queue is empty.
a) The computational behaviors of `put` and `get`, and the associated data structures are shown below:

```
char buffer[SIZE];
int first, last;
put(char data)
{
   buffer[last] = data;
   last = (last + 1) mod SIZE;
}
char get()
{
   char val;
   val = buffer[first];
   first = (first + 1) mod SIZE;
   return(val);
}
```

b) $\Phi = \{$`Serialize(put), Serialize(get), SyncPut(put), SyncGet(get)`$\}$. Let `met.next` define a set of events such that `met.next` contains all invocations of method `met` that i) are currently (at the time `met.next` is evaluated) executing or waiting, and ii) will be invoked in future. Note that `met.next` is time dependent, and may change over time. We define two generic interaction expressions `WaitUntil` and `Serialize` below:

```
WaitUntil(m1, m2, cond) =
      forall e1 in m1:
          forall e2 in m2.next:
             cond ⇒ (e2 ⧀ e1)
Serialize(met) =
      forall i in met:
          forall j in met:
             (i < j) ⇒ ((met,i) ⧀ (met,j))
```

The interaction expression `WaitUntil` defines interaction between events of m_1 and m_2 such that an invocation of method m_1 is delayed with respect to events of m_2 until the boolean condition `cond` becomes false. Expression `Serialize` serializes the events of `met` in the order they arrive. Expressions `SyncPut` and `SyncPut` are defined in terms of `WaitUntil`:

```
SyncPut(put) = WaitUntil(put, get, Full)
SyncGet(put) = WaitUntil(get, put, Empty)
```

In the above, `Full` and `Empty` are boolean conditions that respectively determine if the buffer is full or empty. The interaction behavior of `get` and `put` events is specified by the following interaction expression:

```
Serialize(put) ⓐ Serialize(get) ⓐ
SyncPut(put) ⓐ SyncGet(get)
```

Note that there are no serialization assumptions regarding the invocations of a method. All behaviors of programs that may affect behaviors of other programs are represented by a single mechanism, namely event ordering constraint expressions.

Example 3.3 *(Simple priority)* Let `read` and `write` be two methods. The interaction behavior of the methods is specified by the following constraints: i) `read` and `write` events are mutually exclusive, ii) only one `write` event can execute at a time, and iii) if `read` and `write` events are waiting, the `read` events have higher priority over the `write` events. The three synchronization constraints are represented by three separate interaction expressions:

```
1.  MutualExclusion(read, write)
2.  MutualExclusion(write, write)
3.  Priority(read.wait, write.wait)
```

Expression `MutualExclusion` is defined in example 3.1. Expression `Priority` is defined below:

```
Priority(m1,m2) =
      forall a in m1:
          forall b in m2:
             (a ⧀ b)
```

The event ordering constraint expression specifying the interaction behavior of the two methods is:

```
MutualExclusion(read, write) ⓐ
MutualExclusion(write, write) ⓐ
Priority(read.wait, write.wait)
```

In the above, `read.wait` and `write.wait` denote sets of events, containing waiting `read` and `write` events respectively. If the interaction behavior of `read` and `write` is changed such that the waiting `write` events have higher priority than the waiting `read` events, only the interaction expressions associated with the priority constraint must be changed in the following way:

```
Priority(write.wait, read.wait)
```

Example 3.4 *(Enable command representation)* Many programming languages use an enable command to specify interaction behaviors of methods. They associate a guard, called the *activation condition*, with every method of a class. Guards determine the interaction behavior of a method in the following way: an invocation of a method a can execute if its activation condition, say p, is true. However, if p is not true, the invocation is delayed until p becomes true.

The representation of the enable command in terms of an event ordering constraint expression is specified by examining the interaction relationship between an occurrence, e_1, of method a and an occurrence, say e_2, of set $\Sigma(a, e_1)$. Set $\Sigma(a, e_1)$ contains all events that i) are currently (at the time of occurrence of e_1) executing or waiting, and ii) will be invoked in future on methods of set $M - a$. Note that set $\Sigma(a, e_1)$ depends on both a and e_1. If p is false, e_1 is delayed with respect to e_2. However, if p is true, there are no relationships among the two events; they can execute in parallel. The following event ordering constraint expression represent an enable command:

$$
\begin{array}{l}
\texttt{enable}(p,\ a)\ \texttt{=} \\
\quad\quad \texttt{forall}\ e_1\ \texttt{in}\ a\texttt{:} \\
\quad\quad\quad \texttt{forall}\ e_2\ \texttt{in}\ \Sigma(a, e_1)\texttt{:} \\
\quad\quad\quad\quad (\neg p) \Rightarrow (e_2 \oslash e_1)
\end{array}
$$

As is evident from the above examples, interaction expressions are derived over sets of events such as m_1, `read.waiting`, and $\Sigma(a, e_1)$. An object-oriented programming language may define these sets as a part of the language in order to allow a programmer to specify interaction expressions that include these sets. In addition, it may allow one to construct sets of events from existing sets. We are currently looking at the nature of these sets and mechanisms for extending them. Note that certain sets of events may possibly contain infinite events. An example is the set associated with a method name. It contains all possible invocations of the method. Evaluation of interaction expressions that involve such sets can be done incrementally by ensuring that the interaction constraint associated with the interaction expression remains true during the occurrence of the events of the sets.

3.1.2 Inter-Object concurrency and interaction:

The notion of independence among objects captures concurrency among the program structures associated with objects. Interactions among these programs occur through method invocations. We define a method invocation mechanism for representing interactions among concurrent objects. The method invocation expression

$$\parallel (O_2.m_2(p_1, p_2, \ldots p_n))\ \texttt{where}\ \phi \tag{8}$$

specifies the mechanism for invoking method m_2 on an object O_2 in parallel. The event ordering constraint expression ϕ is used to represent interaction between calling and called methods. Expression 8 derives its form and semantics from the constrainted concurrent composition operator. Assume that method m_1 includes expression 8 in its specification. An execution of the invocation expression during an occurrence of m_1, say (m_1, j), maps m_2 to an occurrence, say (m_2, k), in object O_2. Events of (m_1, j) and (m_2, k) execute in parallel, except for those whose executions must satisfy the ordering constraints in ϕ. An extension of the above method invocation mechanism is specified by the following expression

$$\parallel O_1.m_1 \parallel O_2.m_2 \parallel \ldots \parallel O_n.m_n\ \texttt{where}\ \phi \tag{9}$$

Here, $O_1.m_1, O_2.m_2 \cdots$ and $O_n.m_n$ are invoked in parallel. Interaction among the calling and called methods is specified by ϕ.

The above invocation mechanism is general in that it subsumes traditional synchronous and future-based asynchronous method invocation mechanisms. For instance, in synchronous method invocation, the interaction between a calling method and a called method can be defined by specifying an interaction expression that orders the executions of the two methods. Similarly, in future-based method invocation, the interaction occurs between a read (in the calling method) event and a write (in the called method) event on a future variable. It is represented by the following event ordering constraint expression:

$$\phi = (m_2.\texttt{var.write} \oslash m_1.\texttt{var.read})$$

Here, `var` is a future variable, and `var.write` and `var.read` respectively denote write and read events over `var`.

Interaction expression ϕ in method invocation expressions 8 and 9 can be used to specify interactions among any kinds of operations on objects that methods share through the parameters. Such expressions may specify complex protocols among calling and called methods. Certain properties of the protocols can be verified by examining the nature of the interaction expressions.

3.2 Inheritance of interaction expressions

In sequential object-oriented programming languages, inheritance [26] provides a powerful mechanism for organizing classes in a generalization-specialization relationship. In this hierarchy, classes near the top of the hierarchy capture more general information than the ones below. Such an organization of classes in a hierarchy provides the ability to incrementally extend the behavior of superclasses by inheriting information such as conceptual behaviors, data structures, and/or implementation mechanisms, and by modifying or adding to the inherited behavior. Interaction behaviors of methods add an extra dimension to class specifications. In this section, we examine the manner in which this additional component can be inherited, modified, and extended.

Interactions among methods of a concurrent class represent semantic dependencies among them. As the concurrent class is extended in a subclass by defining more methods and/or by modifying existing methods, additional semantic dependencies among the methods develop. Also many semantic dependencies may need to be redefined. The focus here is to support inheritance of interaction specifications, and to minimize changes in interaction specifications if complete reusability cannot be achieved. It can be achieved if i) additional semantic dependencies due to new methods can be incrementally added, ii) modifications in the semantic dependencies due to the changes in methods can be localized, and iii) interaction behaviors of methods can be extended incrementally. We show that event ordering constraint expressions allow representations of incremental extensions and localized changes in interaction behaviors.

In a class $C = \langle M, \Phi \rangle$, each event ordering constraint expression in Φ characterizes interaction among a set of methods. The methods of the class can therefore be partitioned into interaction groups, each characterized by an interaction expression. Changes in interaction behavior due to the changes in the composition of a class can therefore be limited to changes and/or additions that are localized to the interaction groups. We analyze these changes below formally by examining possible interaction behavior changes that can occur. We also examine their representation in our model.

Let the tuple $\langle \Delta M_c, \Delta\Phi_c \rangle$ extend class $C = \langle M_c, \Phi_c \rangle$ in order to define a subclass S. Here, ΔM_c and $\Delta\Phi_c$ respectively denote sets of methods and interaction expressions. Let ΔM_n and ΔM_m be two components of ΔM. Set ΔM_n is a set of methods that are unique to S, and are not defined in C. Methods in ΔM_m are defined in C, but are modified in S. Set $M_c - \Delta M_m$, therefore, contains methods that are defined in C and are inherited in S without any changes. We now examine the components, $\Delta\Phi_n$ and $\Delta\Phi_m$, of $\Delta\Phi_c$, and the manner in which they arise.

Set $\Delta\Phi_n$ contains interaction expressions that represent new semantic dependencies among the methods of S. These expressions represent the following interactions:

1. Interactions among the methods of $M_c - \Delta M_m$: These interactions specify additional relationships among the methods of C that are inherited in S. They are represented by interaction expressions that impose additional ordering constraints on the methods of $M_c - \Delta M_m$. Interaction behaviors of methods can also be organized in a hierarchy, where interaction behaviors can be made more specific by defining additional ordering relationships. This leads to the organization of class hierarchies where common and more general interaction expressions are captured in more general classes. These interaction expressions can then be inherited and extended in subclasses. This gives rise to a class hierarchy where highly concurrent and nondeterministic classes occur at the top of the hierarchy, while serialized and deterministic classes occur at the bottom.

Consider the example of a class which defines two operations `put` and `get`. A general class may specify that `put` and `get` events are mutually exclusive (see example 3.1). Different subclasses can be defined that extend the highly nondeterministic interaction constraint to specify additional interaction constraints (such as priority and single buffer access constraints). The mutual exclusion constraint among the events of `get` and `put` is inherited in the subclasses.

2. Interaction among the methods of ΔM_n: These define interactions among the newly added methods of S. Interaction expressions here characterize interaction groups containing methods of set ΔM_n.

3. Interactions among the methods ofΔM_n and $M_c - \Delta M_m$: These arise among the newly added methods and inherited methods of S. We present an example that shows how such interactions arise:

Example 3.5 *(Inheritance of interaction expressions)* Let `ReadFirstQueue` be a subclass of class `queue` (see example 3.2). The subclass adds a method `getlast` to access the last element of the queue. This method interacts with method `put`, as it must wait for a `put` event to occur if the queue is empty. The interaction behavior of the methods of the subclass is extended by defining an interaction expression that defines ordering relationship among `getlast` and `put` events:

```
SyncGetLast(getlast) =
     WaitUntil(getlast, put, Empty)
```

Note that events of `put` also interact with those of `getlast`. Hence, the interaction behavior of `put` is also extended by the following interaction expression:

```
SyncPutEx(put) =
     WaitUntil(put, getlast, Full)
```

Other interaction expressions in `ReadFirstQueue` are inherited from the `queue` superclass.

4. Interactions among the methods of ΔM_n and ΔM_c: These represent interactions among the newly added methods and methods that exist in C but are modified in S.

Interaction expressions in $\Delta\Phi_m$ are defined in C but are modified in S in order to incorporate changes in the methods of C. They capture interactions among the methods of i) ΔM_c, representing modified interactions among the modified methods, and ii) ΔM_c and $M_c - \Delta M_c$, representing additional semantic dependencies among the modified and inherited methods of S.

The interactions expressions, $\mathcal{I}(S)$, of S therefore are:

$$\mathcal{I}(S) = \{\ \phi_i \mid (\phi_i \in \Delta\Phi_c) \vee ((\phi_i \notin \Delta\Phi_c) \wedge (\phi_i \in \mathcal{I}(Superclass(S))))\}$$

The interaction expressions for an object are not only defined in its class but are also inherited from the superclasses. In class S, the interaction expressions in set $\Phi_c - \Delta\Phi_m$ are inherited from the superclass C. The following event ordering constraint expression represents the interaction behavior of the methods of class S:

$$\phi_s = \left(\bigcirc\!\!\!\!\!\!\bigwedge_{\phi_i \in \Delta\Phi_n} \phi_i \right) \circledwedge \left(\bigcirc\!\!\!\!\!\!\bigwedge_{\phi_i \in \Delta\Phi_m} \phi_i \right) \circledwedge \left(\bigcirc\!\!\!\!\!\!\bigwedge_{\phi_i \in (\Phi_c - \Delta\Phi_m)} \phi_i \right) \tag{10}$$

3.3 Decomposition of interaction behavior

In a concurrent object-oriented programming language, interactions among concurrent methods can be specified either during class definitions or during method invocations (through expressions 8 and 9). Our goal in this section is to examine the nature of interaction specifications during the two definitions, and relationships between them. We discuss the implications of these specifications on the reusability of concurrent classes and interaction expressions, and examine their representations in our model.

We motivate our discussion through an example. Consider two concurrent programs that interact by sending and receiving messages over a message channel. There are two interaction constraints: i) every `receive` event must wait for the corresponding `send` event, and ii) there are no more than N messages that have not been received. There are three ways in which the constraints can be specified. The first approach is to define a concurrent class `AppChannel` that defines both constraints along with `send` and `receive` methods. In the second approach, one defines a concurrent class `ConcChannel` that does not specify any interaction constraints. The constraints are specified where the `send` and `receive` methods are invoked. In the third approach, the first interaction constraint is defined in a concurrent class `channel`, whereas the second constraint is specified during the invocations of the two methods.

We now examine the three approaches with respect to the reusability of concurrent class and interaction behavior specifications. The first approach advocates constructing concurrent classes that precisely implement requirements of applications. The problem with this approach is that the specifications of the classes are too specialized. They may not be reused in applications that do not impose similar interaction constraints. The applications must re-implement these classes in order to specify diverse interaction constraints. The second approach advocates defining classes that do not specify any interaction constraints. All interaction constraints are specified in applications that invoke the methods. Class `ConcChannel` is too general since applications that require asynchronous communication will specify the first constraint repeatedly. The third approach is based on the incremental specification of interaction behavior. Here, the definition of the `channel` class captures only those interaction constraints that are common across many applications. Additional constraints are specified incrementally in individual applications, thereby facilitating the reusability of both class and interaction behavior specifications.

The above analysis suggests that interaction behaviors of invocations of a method can be decomposed into two kinds of interaction behavior: the first, which we call *universal interaction behavior*, represents interaction behaviors that are common across all possible invocations of the method. All occurrences of the method inherit universal interaction behavior. The second, which we call *specialized interaction behavior*, is specific to a method invocation. For instance, in the above example, the universal interaction behavior (every `receive` event must wait for the corresponding `send` event) is valid for all invocations of `send` and `receive`. The interaction behavior that certain `send` events must be delayed if specific `receive` events have not occurred is valid for specific invocations of `send` and `receive` in the application. Different applications may specify diverse specialized interaction behaviors of invocations of `send` and `receive`.

Universal interaction behavior defines a single set of interaction constraints that guide execution behaviors of all occurrences of method. The appropriate place for defining universal interaction behaviors of methods is the class the methods are defined in, since the class captures general interaction constraints that govern access to objects of the class. Examples of such interaction constraints are data consistency, fairness, mutual exclusion, and priority. Note that universal interaction behavior of methods should only depend on the intrinsic properties of the objects and the methods, since it will allows one to maintain the notion of independence and encapsulation that classes exhibit — that is, the ability to define classes in isolation from other aspects of applications. Specialized interaction behavior of method events, on the other hand, depends on the environment in which the events occur. It is, therefore, possible that different occurrences of a method exhibit diverse specialized interaction behaviors, each extending the universal interaction behavior of the method.

Our programming model supports the above incremental approach to interaction specification by allowing one to define universal and specialized interaction specifications separately. The ability to separate the two is supported by the modularity property of event ordering constraint expressions. *The interaction behavior of a method invocation is a composition of the two — universal and specialized — interaction behaviors.* Hence, if event ordering constraint expression ϕ_u represents the universal interaction behavior of methods M_1 and M_2, and if interaction expression ϕ_s represents the specialized interaction behavior of events (M_1, p) and (M_2, q), the interaction behavior of (M_1, p) and (M_2, q) is represented by expression $\phi_u \circledwedge \phi_s$. Intuitively, we can think of universal interaction behavior as defining a set of possible ordering relationships among method activations, some of which must hold true. Specialized interaction behaviors of method activations impose additional constraints on the possible orderings among the events.

An example of a class that defines universal interaction behavior is `SafeBuf`, whose methods, `put` and `get`, are mutually exclusive. The interaction expression `MutualExclusion(put, get)` (see example 3.1) represents their interaction behavior. The specialized interaction behavior of method events of interacting programs is specified by identifying the method events of the programs, and by defining event ordering constraint expressions that include the method events. Interaction points of methods allow one to capture such invocations. Specialized interaction behaviors, therefore, are specified by defining event ordering constraint expressions that establish orderings among the interaction points of programs. We illustrate this by the following example:

Example 3.6 *(Specialized Interaction Behavior)* Let `m1(SafeBuf X)` and `m2(SafeBuf X)` be two methods that interact. Methods `m1` and `m2` invoke methods `put` and `get` over a shared buffer `X`. Assume that the application imposes a constraint on `put` and `get` events of the two methods. The constraint represents specialized interaction behaviors of these events. It states that every `put` event in `m1` occurs before the corresponding `get` event in `m2` (establishing some kind of data dependency among the `put` and `get` events).

Terms `m1.X.put` and `m2.X.get` denote sets of all `put` and `get` events invoked in `m1` and `m2` respectively. The expression

$$\phi_1 = \texttt{forall } k \texttt{ in m1.X.put:} \\ \quad (\texttt{m1.X.put, k}) \circledless (\texttt{m2.X.get, k})$$

specifies that all `put` events in set `m1.X.put` occur before the corresponding `get` events in set `m2.X.get` (the correspondence is made through the occurrence numbers of method events). Executions of `put` and `get` events in `m1` and `m2` must satisfy both i) the mutual exclusion constraint (universal interaction behavior), and ii) the above data dependency constraint (specialized interaction behavior). In the absence of any specialized interaction behavior specifications, the `put` and `get` events of `m1` and `m2` inherit the universal interaction behavior from class `SafeBuf`.

The interaction behavior of a method invocation is therefore a composition of its universal and specialized interaction behaviors. Such a separation of the interaction behaviors has implications on the design of concurrent classes with respect to the reusability of both concurrent class and interaction behavior specifications.

3.4 Inheritance of method implementation

In sequential object-oriented programming languages, inheritance [26] provides a basis for reusing method implementations. In concurrent object-oriented programming languages, there is a problem with the inheritance of method implementations. This problem, termed the *inheritance anomaly* [16], arises due to the diverse synchronization requirements of a class and its subclasses. We clarify the problem through the following example:

Assume that methods m_j and m_k are defined in a class C_m. Implementations of m_j and m_k may contain, in addition to specifications of computations, operations used to define interaction behaviors of m_j and m_k. Let S_m be a subclass of C_m. It inherits definitions of m_j and m_k. Due to the addition and/or modification of computational and interaction behaviors in subclass S_m, the interaction behaviors of m_j and m_k that are defined in C_m may not represent their interaction behaviors for objects of class S_m. Methods m_j and m_k must be re-implemented in S_m in order to represent the modified interaction behaviors. The implementations of m_j and m_k, thus, cannot be inherited.

We first explore the reason for the above anomaly. There are two distinct behaviors of a method: i) computational behavior, and ii) interaction behavior. Computational behavior of an inherited method remains unchanged in the subclass; only its interaction behavior changes due to changes in the composition of the superclass. For instance, computational behaviors of m_j and m_k do not change in S_m; only their interaction behaviors changes. The inheritance anomaly arises because implementations of methods in most concurrent programming languages combine the two — computational and interaction — behaviors. Any changes in interaction behavior, therefore, requires changes in the implementation as well.

There are two components in any solution to the inheritance anomaly. The first is the separation of computational and interaction behaviors of methods. The separation of the two behaviors makes it possible to inherit them separately and modify either to reflect changes in the composition of a class. The second is the ability to make changes in interaction behaviors of methods. Our programming model supports inheritance of method implementations by i) separating specifications of computational and interaction behaviors, and ii) providing the ability to change specifications of interaction behaviors of methods. The basis for the separation of the two is derived from the C-YES model (see section 2). We analyze the nature of changes in interaction behaviors in detail below.

Interaction expressions in a class represent universal interaction behaviors of methods of the class. These expressions may also include specifications of specialized interaction behavior of methods that are invoked inside these methods. Changes in interaction behavior of a method, therefore, may involve changing specialized interaction behaviors of methods (interaction points) it invokes. These changes imply that the events associated with the interaction points are used (with respect to their interaction characteristics) in a manner different from the way they are used in the superclass. Universal interaction behaviors of the interaction points remain unchanged, since they are determined in classes of methods associated with the interaction points. Our programming model supports changes in both universal and specialized interaction behaviors. Changes in universal behavior that do not involve any changes in the specialized interaction behavior of interaction points can be made by constructing suitable event ordering constraint expressions that do not involve any interaction points. Similarly, specialized interaction behavior of an interaction point is changed by specifying an event ordering constraint expression that includes the interaction point and represents the modified interaction behavior.

The methods, $M(S)$, of S of section 3.2 are, therefore, defined by:

$$M(S) = \{\ m\ |(m \in \Delta M_c) \vee \\ ((m \notin \Delta M_c) \wedge (m \in M(Superclass(S))))\}$$

The program $\mathcal{P}(O_s)$ associated with an object O_s of S is:

$$\mathcal{P}(O_s)\ =\ \left(\mathop{\Big\|}_{\substack{k\in\{1\ldots\infty\}\\ m_j\in M(S)}} (m_j, k)\right)\ \texttt{where}\ \ \phi_s$$

In the above, ϕ_s is defined by expression 10.

The ability to resolve inheritance anomaly in our model depends on i) the ability to capture method invocations inside methods through interaction points; ii) the ability to decompose interaction behaviors of methods in a generalization-specialization relationship (both between classes and subclass definitions, and class and method invocations), define and modify each separately, and combine them; and iii) the ability to represent interaction behaviors, both universal and specialized, *separately* from computational behavior specifications. This suggests that *a necessary condition for the resolution of the inheritance anomaly is that programming languages must provide the ability to change both universal and specialized interaction behaviors of methods, and enforce this change separately from method implementations.*

4 Related work

Several concurrent programming languages have used the concept of encapsulated "object" as a common basis for introducing concurrency. It is evident in i) server-based approaches such as Rendezvous-based languages ADA [9] and RPC-based languages; ii) approaches based on message passing such as CSP [13]; iii) approaches based on abstract data types (ADT) such as Monitors [12], ADT with path expressions [5], and SR [3]; iv) approaches based on sequential objects such as POOL-T [2], ABCL/1 [28], Concurrent SmallTalk [27], CC++ [6], Mentat [11], and Charm++ [15]; and v) actor-based approaches [1]. The languages differ in their support for internal and external concurrency and interaction, and inheritance.

We first look at languages for their support of concurrency within objects. Languages such as Monitor [12], POOL-T [2], ABCL/1 [28], and Concurrent Smalltalk [27] support only a single thread of execution within an object; concurrent invocations of methods are always serialized and scheduled for execution according to the policies of the implementation. Such language-imposed serializations define semantic dependencies among method invocations where there should be none. One must specify concurrent programs with these dependencies in mind; one may otherwise end up defining incorrect programs. For instance, deadlock occurs when nested calls are made in monitors or when a method invokes another method through "self" in serialized concurrent object-oriented programming languages.

Our approach is to focus on aspects that are fundamental to representation of application entities. The implementation concerns are handled within this framework by defining a separate execution model that determines the manner (through fixed or variable number of threads) in which method invocations are scheduled for execution. The mapping between the programming model and the execution model is carried out by specifying additional constraints that serialize different method invocations. In this framework, certain method invocations may be concurrent at the program model level, while their executions may be serialized at the execution model level. Languages such as Path Expression [5], CC++ [6], PO [7], and Mediator [10] also support concurrent method invocations within objects.

Two kinds of synchronization mechanisms are used for specifying interaction behavior of methods. The first, called *interface control mechanism*, is used to select a set of method requests from among concurrent requests for execution. It can be done either explicitly by executing a "select" statement, as in languages such as ADA [9], Mediator [10], and POOL-T [2], or implicitly through *interface control conditions* [19, 17] which must be true (or false [8]) before a method can be executed. The second set of synchronization mechanisms is used to specify interactions among method invocations that satisfy their respective interface control conditions and are executing concurrently. Examples of synchronization mechanisms are semaphores and locks, write-once-read-many shared variables [6], data flow based data dependencies [11], and signal variables [12]. We use event ordering constraint expressions to specify both interface control and interactions during executions.

The problem of inheritance anomaly has been studied in great detail and many solutions [14, 24, 4, 22, 21, 19, 23, 18] have been proposed. Most of these solutions support inheritance of method implementations by separating the interface control conditions from method implementations, and by excluding any synchronization operations from the method implementations. Changes in the interaction behavior of a method is achieved by changing the relevant interface control conditions. However, modifications in the interface control conditions of a method are not sufficient to change the interaction behavior of the method, since it cannot change the specialized interaction behavior of methods that the method invokes. These can be only be changed by embedding synchronization operations such as semaphores, locks, and other primitives. However, their inclusion violates the basis of resolution of the inheritance anomaly, namely, the separation of computational and interaction behaviors.

In addition to the inheritance of method implementations, inheritance of interaction behavior specifications have also been explored in many languages. PCM [4] identifies two kinds of synchronization expressions of an object: the first, called *concurrency constraints*, do not depend on the state of the object, whereas the second, called *state constraints*, do. Concurrency constraints are defined separately in abstract classes, whereas state constraints are defined in classes. The language provides mechanisms for inheriting and combining the two constraints. In DRAGOON [21], interaction expressions are defined separately in classes called behavior classes. The computational behavior of objects are defined in classes, called free classes (methods here are concurrent by default, as they are in our model). A concurrent class is constructed by composing a set of behavior classes and free classes. Synchronizing Action [19] and PLOOC [23] also define and inherit interaction and computational behaviors separately.

In [8], interface control conditions are represented in a manner such that subclasses extend the interface control conditions of their methods by imposing additional interface control conditions. This is similar to our approach in that classes contain more general interaction constraints. Subclasses impose additional interaction constraints, while inheriting general constraints from superclasses.

The support for method invocation in most concurrent programming languages is based either on the call-return based synchronous invocation or future-based asynchronous invocation. The synchronous invocation mechanism does not exploit concurrency between calling and called methods. The asynchronous approach, on the other hand, supports concurrency among the calling and called methods; however, the support for interaction between the two methods is limited in terms of both the number of interactions (fixed) and kinds of interactions (only among reads and writes over shared variables).

5 Conclusion

In this paper, we present a compositional approach for concurrent object-oriented programming. A concurrent object is represented as a composition of two separate entities, namely computational and interaction behavior specifications. The separation of the two behaviors makes it possible to inherit them separately in a subclass and modify either to reflect any changes in the composition of the superclass. Our interaction specification mechanism minimizes changes in interaction behavior specifications in the subclass in order to support reusability by allowing one to i) incrementally add interactions that arise due to the addition of new methods, and ii) extend interactions among existing methods incrementally. Also, changes in interactions are localized to event ordering constraint expressions that involve changed methods and other interacting methods. We also identify the cause for the inheritance anomaly. We show that the inheritance anomaly can be resolved if programming languages provide the ability to change both universal and specialized interaction behaviors of methods, and enforce this change separately from method implementations.

Our future work involves the design and implementation of a concurrent object-oriented programming language, and a general analysis of the inheritance anomaly and the manner in which it can be resolved in concurrent object-oriented programming languages.

Acknowledgement

This work was supported by Texas Advanced Research Program under grant 003658-445 and by DARPA under grant number DABT 63-92-C-0042.

References

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

[2] Pierre America. POOL-T: A Parallel Object–Oriented Language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, 1987.

[3] G. R. Andrews. Synchronizing Resources. *ACM Transaction on Programming Languages and Systems*, 3(4):405–430, 1981.

[4] J. P. Bahsoun and L. Feraud. A Model for Design Reusable Parallel Software Components. In *Parallel Architecture and Languages Europe, LNCS 605*, pages 245–260. Springer Verlag, 1992.

[5] 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.

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

[7] Antonio Corradi and Letizia Leonardi. An Object Model to Express Parallelism. In *Workshop on Object-based Concurrent Programming, ACM SIGPLAN Notices V. 24, No. 4*, pages 152–155. ACM Press, 1989.

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

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

[10] J. E. Grass and R. H. Campbell. Mediators: A Synchronization Mechanism. In *Sixth International Conference on Distributed Computing Systems*, pages 468–477, 1986.

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

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

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

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

[15] 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.

[16] 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.

[17] Ciaran McHale, Bridget Walsh, Seán Baker, and Alexis Donnelly. Scheduling Predicates. In *Object-Based Concurrent Computing Workshop, ECOOP'91, LNCS 612*, pages 177–193. Springer Verlag, 1991.

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

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

[20] 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.

[21] S. Crespi Reghizzi and G. Galli de Paratesi. Definition of Reusable Concurrent Software Components. In *ECOOP '91*, pages 148–165. Springer–Verlag, 1991.

[22] Etsuya Shibayama. Reuse of Concurrent Object Descriptions. In *Concurrency: Theory, Language, and Architecture, LNCS 491*, pages 110–135. Springer–Verlag, 1989.

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

[24] 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.

[25] Peter Wegner. Dimensions of Object–Based Language Design. In *OOPSLA'87*, page 168. ACM Press, 1987.

[26] Peter Wegner and Stanley Zdonik. Inheritance as an Incremental Modification Mechanisms or What Like is and Isn't Like. In *ECOOP'88, LNCS 322*, pages 55–77. Springer Verlag, 1988.

[27] Y. Yokote and M. Tokoto. Concurrent Programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. The MIT Press, 1987.

[28] A. Yonezawa, J. Briot, and E. Shibayama. Modeling and Programming in Object–Oriented Concurrent Language ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–89. The MIT Press, 1987.

[29] Akinori Yonezawa. Panel on Object-Based Concurrent Programming. In *Proceedings of the ECOOP-OOPSLA Workshop on Object-based Concurrent Programming, OOPS Messenger, Vol 2, No 2*, pages 3–4. ACM Press, 1991.