Support for Implementation of Evolutionary Concurrent Systems in Concurrent Programming Languages

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

[1] Computer Science Department, University of California, Davis, CA 95616
[2] Department of Computer Sciences, The University of Texas, Austin, TX 78712

Abstract. In many concurrent programming languages, concurrent programs are difficult to extend and modify: small changes in a concurrent program may require re-implementations of a large number of its components. In this paper a novel concurrent program composition mechanism is presented in which implementations of computations and synchronizations are completely separated. Separation of implementations facilitates extensions and modifications of programs by allowing one to change implementations of both computations and synchronizations. The paper also describes a concurrent programming model and a programming language that support the proposed approach.

1 Introduction

Complex software systems are evolutionary in general. They change during the initial development stage, and often after they have been deployed. These changes may occur due to changes in the requirements, in the hardware configuration, and/or in the execution environment. Programming languages must support methodologies that allow implementations of evolutionary systems. Specifically, small changes in the implementations of such systems should be localized, and should require modifications of a small number of components.

In this paper we show that many concurrent programming languages do not adequately support implementation of evolutionary concurrent systems: changes in the implementations of a small number of components may affect the implementations of a *disproportionately* large number of components. More importantly, concurrent program abstractions cannot be composed easily with existing program abstractions. This has implications on the re-usability of program abstractions and on concurrent programming language design. Specifically, the inability to compose concurrent program abstractions causes breakdowns in many of the programming language composition mechanisms.

A novel structuring scheme for concurrent programs is presented in this paper. In this scheme, *implementations of computations and synchronizations are completely separated.* A concurrent program is, thus, composed from *separate* implementations of computations and synchronizations. This is unlike most existing approaches where implementations of computations and synchronizations are embedded within the implementations of components.

Separation of implementations of computations and synchronizations has direct implications on the extensibility and modifiability of programs. Concurrent programs can be easily extended and modified by adding and modifying implementations of either computations, synchronizations, or both. Further, the approach advocates a programming design methodology where concurrent programs can be quickly constructed from existing implementations of computations and synchronizations. We briefly describe a concurrent programming model and a concurrent programming language that supports this programming methodology. The model defines general mechanisms for representing computations, interactions, and program compositions. The object-oriented programming language, CYES-C++, supports extensibility and modifiability of concurrent programs as well as re-usability of specifications of computations and interactions.

This paper is organized as follows: In Section 2, we show that there is poor support for implementation of evolutionary concurrent systems in many existing approaches. In Section 3, we analyze the reasons for the problems, and show how some of these problems can be resolved. In Section 4, the details of a concurrent programming model and a language that support the programming methodology are presented. A brief survey of the related work is presented in Section 5. Section 6 contains concluding remarks and the status of the research.

2 Modifications of Concurrent Programs

In this section we show that it is difficult to change the implementation of a concurrent system implemented using traditional approaches to concurrent programming. 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 contains operations that implement its computations, synchronization with other components, data decomposition and distributions and task scheduling algorithms. We 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 by showing that extensions and modifications of a simple concurrent program require re-implementation of some or all of its components. Note that the conclusions of this exercise are independent of the example.

Example 2.1. *(Extensibility and Modifiability of Concurrent Programs).* Below we show a concurrent program, `examprog1`, that is composed from two components: `producer` and `consumer`. The `producer` component repeatedly produces data, which are consumed by the `consumer` component. The components interact through the `send` and `receive` primitives over a mailbox [1] in which programs can deposit and retrieve information in a FIFO manner. Primitives `send` and `receive` respectively are non-blocking and blocking.

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

producer(channel buf){                    consumer(channel buf){
   while (TRUE) {                            while (TRUE) {
      info = produce();                         info = receive(buf);
      send(buf, info);                          consume(info);
}}                                        }}
```

A simple extension of `examprog1` involves adding another `consumer` component, for instance because `consumer` is slow relative to `producer`, such that data are now shared between the two `consumer` components *alternately.* There are many possible implementations of the extended program. However, in all implementations, `producer`, `consumer`, or both must be re-implemented in order to implement the altered interaction among the `producer` and the two `consumer` components.

Similarly, a modification of `examprog1` may involve defining additional synchronization constraints — for instance, `producer` must wait after N un-consumed data — between `producer` and `consumer`. Again, as in the case of the extension, either or both components must be re-implemented in order to implement the altered interaction. ■

Even though the above program contains two simple components, implications of simple changes in the program are widespread. *Simple extensions and modifications in a concurrent program may therefore affect implementations of a large number of its components.* Implementations of component are not encapsulated from each other. Changes in a concurrent program may be visible in some or possibly all components.

Also, specifications of components cannot be reused easily. For instance, in three versions of the example program, much of the behavior of `producer` and `consumer` remains unchanged. However, different versions of the components are created by duplicating much of the code from one version to another. In addition, synchronization, task scheduling, data mapping, and data distribution algorithms cannot be reused easily because they are embedded *procedurally* inside the implementations of components.

Further, 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 errors in concurrent programs.

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

Definition 2.1. *(Program Composition Anomaly).* The program composition anomaly denotes the phenomenon in which the concurrent program composition of program abstractions requires changes and modifications in some of all of the program abstractions. ■

Example 2.1 shows an occurrence of the program composition anomaly. The program composition anomaly highlights the inability to compose concurrent program abstractions from existing program abstractions. 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 below.

Object-oriented programming 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. In a concurrent object-oriented programming languages, we can think of a concurrent object as a concurrent program, whose composition is defined in terms of its methods and interactions among the methods. 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. We show that instances of the program composition anomaly occur when defining the two composition mechanisms.

Aggregation anomaly: The aggregation anomaly occurs when an object defines additional interaction behavior for methods of its component objects.

Example 2.2. *(Aggregation anomaly).* Assume that an object of class `TwoBufs` contains two objects: `LarBuf` and `SmBuf` of a concurrent class `AtBuf`. Class `AtBuf` defines two methods: `Read` and `Write`. The two methods synchronize with each other while accessing common data structures of `AtBuf`. Let Class `TwoBufs` define addition constraints on invocations of `Read` and `Write` over `LarBuf` and `SmBuf` objects: `Write` invocations on `LarBuf` have higher priority than `Write` invocations on `SmBuf`. Since the synchronization operations of `Write` are embedded inside the implementation of `Write`, the new synchronization behavior can be specified only by re-implementing the methods in `AtBuf`, thereby requiring redefinition of `AtBuf`. ■

In this example, class `TwoBufs` is used to compose two instances of abstraction `AtBuf` along with additional synchronization constraint. However, such a composition requires changes in the abstraction (`AtBuf`).

Inheritance anomaly: The second problem, termed the *inheritance anomaly* [16], arises due to the diverse synchronization requirements of a class and its subclasses.

Example 2.3. *(Inheritance anomaly).* Let class `NBuf` extend class `AtBuf` by defining a new method `GetLst`. Method `GetLst` interacts with `Read` and `Write` of `AtBuf`. This implies that synchronization properties of `Read` and `Write` change. Since the implementations of `Read` and `Write` include synchronization operations, the interaction behaviors of the methods can be implemented only by re-implementing the methods. This can be achieved either by re-implementing `AtBuf` or by re-implementing `Read` and `Write` in `NBuf`. In the latter case, implementations of `Read` and `Write` cannot be inherited in `NBuf`. ■

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 require changes in the composition, which, in this case, means redefinition of methods.

3 Support for Extensibility and Modifiability

We first examine the reason for 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, computational behavior of the `producer` component 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 a semantic relationship among components. For instance, the interaction behavior of `consumer` (example 2.1) specifies that every invocation of `consume` depends on a preceding invocation of `produce`, representing a data dependency relationship among the operations.

The program composition anomaly arises because implementations of both — computational and interaction — behaviors of a component are embedded within an implementation of the component. Any changes (either through extension or modification) in a concurrent program tend to change the existing interaction relationships among the components. Since implementations of the relationships are distributed in the implementation of the components, changes in an interaction relationship can be effected only by re-implementing all components that implement the relationship.

3.1 Concurrent Program Composition

Our approach, which we call *evolution through separation*, is based on a novel structuring technique for concurrent programs. It advocates a programming methodology 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.

Definition 3.1. *(Constrained concurrent program composition).* The expression

$$C = (C_1 \parallel C_2 \parallel \ldots \parallel C_n) \ \texttt{where} \ \phi$$

specifies a concurrent program C. Program C is composed from components C_1, C_2, …, and C_n and expression ϕ that represents relationship among the operations of the components. ■

The semantics of the composition is that during an execution of C, operations of components C_1, C_2, ..., and C_n occur in parallel by default. However, there are invocations of operations that interact. Executions of these invocations must satisfy all interaction relationships specified by ϕ. Concurrent program `examprog1` is thus defined as:

```
examprog1 = (producer || consumer) where consexp1
```

In this definition, components `producer` and `consumer` define only their computational behavior. Expression `consexp1` defines interaction among operations of `producer` and `consumer`.

3.2 Implications of separation

Separation of implementations of computational and interaction behaviors have direct implications on extensibility and modifiability of concurrent programs, as well as re-usability of components.

Concurrent programs can be extended easily. Additions of components may require definition of new interaction behaviors, and possible modifications of existing ones. For instance, `examprog1` can be extended easily:

```
examprog2 = (producer || consumer || consumer) where consexp2
```

Expression `consexp2` represents the new interaction relationship among the three components. Implementations of either `producer` or `consumer` do not change.

A concurrent program can be modified easily either by modifying computational behavior of its components or their interaction behaviors. For instance, the following program

```
examprog3 = (producer || consumer) where consexp3
```

is composed from the same components as `examprog1` except that `consexp3` implements a different interaction behavior among the components. The approach supports encapsulation of implementations of both computational and interaction behaviors. For instance, `producer` can be re-implemented, in isolation, from the implementations of `consumer` and the interaction behavior. Even if this implementation implies changes in concurrent program, only the implementations of interaction behaviors needs to be changed. The computational behavior of `consumer` remains unaffected. Separation of implementations therefore *localizes* the effects of changes in a concurrent program. Further, it supports re-usability of implementations of both computational and interaction behaviors. For instance, different versions of `examprog1` can be constructed by combining `producer` and `consumer` in many different ways. Indeed, it advocates a programming design methodology in which concurrent programs can be quickly constructed from existing implementations of computational and interaction behaviors.

Verification of concurrent programs is also facilitated by the separation of the implementations. The approach allows one to verify properties of the system by looking at the implementations of computational and interaction behaviors in isolation.

Separation also forms the basis for the resolution of the aggregation and inheritance anomalies. In the case of inheritance anomaly, interaction behavior of inherited methods can be extended and/or modified by defining interaction behaviors in a subclass [21]. The inheritance anomaly has been studied in great detail and many solutions [14, 26, 23, 25] have been proposed. Most of these solutions are based on the separation of synchronization constraints from the method specifications as well.

Separation of implementations facilitates programming language design as well. By supporting mechanisms for defining abstractions for computational and interaction behaviors, a concurrent programming language can provide support for constructing powerful concurrent program abstractions by simply extending the existing composition mechanisms. The design of CYES-C++(Section 4.3) clearly benefited from this approach.

4 Support for Concurrent Programming

We now describe a concurrent programming model and a programming language that support the proposed programming methodology. We first present a model of concurrent computation, called the C-YES model [20]. The C-YES model defines representation mechanisms for computational and interaction behaviors. It has been used to define a compositional model for concurrent object-oriented languages [21], and a concurrent object-oriented programming language, CYES-C++ [22]. Due to the lack of space, we outline only the fundamental aspects of the model and the language. The details can be found in [19].

4.1 Representation of computational behavior

Given that implementations of components do not include implementations of interaction behaviors, the question is: how are component programs implemented so that their interaction behaviors can be specified in a concurrent program?

The execution behavior of a component is to repeatedly execute operations, and occasionally interact with its environment (other components) during the execution of certain operations. For instance, `producer` interacts with its environment during executions of `produce` operations. We call such operations *interaction points*. An interaction point denotes a set of possible invocations of operations where interaction may occur. A component in the C-YES model is therefore represented by its computations and interaction points. We call each invocation of an operation an *event*. An interaction point therefore denotes a set of possible events.
We represent an event by `Operation[Selector]`. Here, the term `Selector` is used to uniquely identify an occurrence of `Operation`. 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 ith invocation of an operation in a computation. For instance, term `produce[0]` denotes the first invocation of `produce`.

Components are represented by extending the interfaces of procedures to incorporate the notion of interaction points. In CYES-C++, interaction points of a component are derived from the parameter variables: all methods on objects denoted by the variables are the interaction points of the component. (We assume that the parameters represent objects). For instance, the implementations of `producer` and `consumer` are shown below:

```
producer(buffer info){
    while (TRUE) {
          info.produce();
    }
}
```

```
consumer(buffer info) {
      while (TRUE) {
           info.consume();
      }
}
```

Interaction points of `producer` are represented by the term `info.produce()`, which denotes the set of all possible invocations of `produce` during an execution of `producer`. Interaction behaviors of components are defined in terms of their interaction points.

4.2 Interaction specification

Interaction among programs is specified by an expression, called the *event ordering constraint expression*. An event ordering constraint expression 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*.

Primitive event ordering constraint expression: A primitive event ordering constraint expression `(e1 < e2)` specifies the constraint that event `e1` must occur before event `e2`

Interaction composition operators: There are four operators for composing event ordering constraint expressions:
i) And constraint operator (&&): An execution of a program satisfies event ordering constraint expression `(E1 && E2)` containing `&&` if it satisfies *both* `E1` and `E2`.
ii) Or constraint operator(||): An execution of a program satisfies event ordering constraint expression `(E1 || E2)` if it satisfies *at least one* of event ordering constraint expressions `E1` or `E2`.
iii) forall operator: The `forall` operator extends `&&` in order to specify ordering constraints over sets of events. There are two ways in which the `forall` operator can be specified. The first

```
forall var v in S { E(v) }
```

specifies that event ordering constraint expression `E(v)` holds true for all events `v` in event set `S`. In this expression, variable `v` iterates over the events of S. The second

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

specifies that event ordering constraint expression `E(S[exp(i)])` holds true for all events `S[exp(i)]` of `S`. In this expression, variable `i` ranges over the occurrence numbers of events of `S`. Expression `exp(i)` determines the occurrence number of the event for which `E` must hold.
iv) Exists operator: The `exists` operator is similar to `forall` in that it extends the `||` constraint operator over a set of events.

The interaction specification mechanism is declarative in nature. Its power stems from the ability to decompose global interactions among programs into a set of local interactions, each represented by event ordering constraint expressions, and combined with suitable interaction composition operators. One of the implications of the modularity property of event ordering constraint expressions is that interaction behaviors of programs can be changed by modifying only the relevant and local interaction specifications. Further, the interaction specification mechanism is is not based on the semantic properties of any synchronization primitive. It can be used to specify any interaction behavior for any invocation of any operation.

Example 4.1. *(Interaction specification).* We now present an example that illustrates the manner in which event ordering constraint expressions can be used for specifying interaction relationships. In this example, we show different instances of event ordering constraint expressions for the producer/consumer example.

Simple data dependency: In example 2.1, the synchronization constraint specifies that the ith invocation of `consume` cannot execute until the ith invocation of `produce` has occurred. Let the terms `produce` and `consume` respectively denote the interaction points of `producer` and `consumer`. The following expression implements the data dependency relationship between `producer` and `consumer`:

```
ConsExp1 =  forall occ i in produce { (produce[i] < consume[i]) }
```

Extended concurrent program: In this example, we consider interaction between a single producer and two consumers. Assume that the data produced by `producer` are shared between the two `consumer` components *alternately.* Also, assume that `consume1` and `consume2` denote the interaction points of the two `consumer` components. The interaction relationship between the components is derived by implementing two relationships: one between *odd* events of `produce` and events of one `consumer`, and the other between *even* events of `produce` and events of the other `consumer`. The two relationships are implemented by the following expression:

```
TwoRel = (produce[2*i-1] < consume1[i])&&(produce[2*i] < consume2[i])
```

The above relationship is true for all events of `produce`, which represents the interaction relationship among the components:

```
ConsExp2 =  forall occ i in produce { TwoRel }
```

Modification of concurrent program: In this example, the interaction relationship between `producer` and `consumer` of example 2.1 is modified by defining an additional constraint: there are at most N unconsumed values. Component `producer` therefore must wait for `consumer` if there are N unconsumed values. The modified interaction relationship among the events of `producer` and `consumer` can be implemented by simply extending the existing interaction relationship (as implemented by `ConsExp1`) with suitable event ordering constraint expression that represents the additional constraint:

```
ConsExp3 = ConsExp1 &&
           forall occ i in consume { (consume[i] < produce[i+N]) }
```

■

4.3 Design of a Programming Language

The C-YES model is a general model of concurrent computation in that it can be applied to define many concurrent programming languages. In our research, we combined it with the object-oriented model [27] in order to design a concurrent extension of C++ [24], called CYES-C++ [22]. The design of CYES-C++ is facilitated, and in parts driven, by the notion of separation. In CYES-C++, both computations and interactions are defined as abstractions. CYES-C++ supports powerful concurrent programming abstractions by extending existing C++ abstractions that combine computational and interaction behavior abstractions in different ways. We briefly enumerate them below (See [22] for detail):

Concurrent class: CYES-C++extends the notion of a C++ class in order to define concurrent objects. In CYES-C++, a concurrent object is represented as a composition of a set of methods and a set of event ordering constraint expressions. The event ordering constraint expressions represent interaction relationships such as semantic dependencies, data consistency, and priority among the methods. Concurrent classes allow one to model concurrent objects that permit multiple concurrent activities to occur at the same time.

Inheritance: In CYES-C++, inheritance is a mechanism for extending the program composition of concurrent objects. Separation of implementations of computational and interaction behaviors allows one to extend and modify either components of a concurrent class. CYES-C++ supports inheritance of implementations of both computational and interaction behaviors.

Genericity: C++ provides the template mechanism for implementing generic data structures. CYES-C++ extends the notion of template classes in order 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 computational and interaction behaviors with user defined abstractions. Separation of implementations of computational and interaction behaviors allows either or both behaviors to be instantiated with a class.

Coordination Structure: Open software systems are often characterized by sets of autonomous and distributed objects whose execution behaviors must be coordinated. We have developed a coordination structure, called *object space*. An object space is a composition of a set of objects and a set of event ordering constraint expressions that define coordinate constraints among invocations of methods on the objects of an object space.

5 Related Work

In most approaches to concurrent programming, implementations of computations and synchronization are embedded within the implementation of components. Separation of implementation of computational and interaction behaviors has been proposed for the resolution of the inheritance anomaly [16]. 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 [10] 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 [8] 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.

There has been extensive work done in the area of concurrent programming. Most of this work has focussed on developing methodologies, languages, and tools for implementing concurrent programs. Most languages have added constructs for specifying concurrency and synchronization in a base languages. An extensive survey of these constructs is given in [19]. Examples of synchronization mechanisms are: semaphores [5, 3], write-once-read-many variables [6], data flow based data dependencies [13], signal variables, enable-based approaches [11, 18, 25, 17, 7, 12], disable based approaches [9], and behavior abstraction based approaches [14, 15].

Our proposed interaction specification mechanism differs from most approaches in that it is declarative, and compositional. It supports abstractions for defining interaction behaviors. The abstractions can be modified and extended in isolation from other abstractions. Further, 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 [4]. 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 [2] 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 [22].

6 Conclusion and Status

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.

A prototype implementation for CYES-C++ currently runs on a network of RS/6000 workstations.

References

1. Gregory R. Andrews. *Concurrent Programming.* The Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.
2. Toby Bloom. Evaluating Synchronization Schemes. In *Proc. 7th Symposium on Operating Systems Principles*, pages 24–32. ACM, 1979.
3. 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.
4. 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.
5. R. Chandra, A. Gupta, and J. L. Hennessy. COOL: A Language for Parallel Programming. In *Languages and Compilers for Parallel Computing Conference*, pages 126–147. Springer Verlag, 1992.
6. K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional Parallel Programming. Technical Report Caltech-CS-TR-92-13, Cal Tech, 1992.
7. D. Dechouchant, S. Krakowiak, M. Meyesmbourg, M. Riveill, and X. Rousset de Pina. A Synchronization Mechanism for Typed Objects in a Distributed Systems. In *Workshop on Object-based Concurrent Programming*, pages 105–107. ACM SIGPLAN, ACM, Sept. 1989.
8. Ian T. Foster. Information Hiding in Parallel Programs. Technical Report MCS-P290-0292, Argonne National laboratory, 1992.

9. Svend Frolund. Inheritance of Synchronization Constraints in Concurrent Object–Oriented Programming Languages. In *ECOOP '92, LNCS 615*, pages 185–196. Springer Verlag, 1992.
10. Svend Frolund and Gul Agha. A Language Framework for Multi-Object Coordination. In *Proceedings of the ECOOP'93*, pages 346–360, 1993.
11. Narain H. Gehani. Capsules: A Shared Memory Access Mechanism for Concurrent C/C++. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):795–810, July 1993.
12. J. E. Grass and R. H. Campbell. Mediators: A Synchronization Mechanism. In *Sixth International Conference on Distributed Computing Systems*, pages 468–477, 1986.
13. Andrew S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, 26(6):39–51, 1993.
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. Satoshi Matsuoka. *Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming.* PhD thesis, The University of Tokyo, Japan, June 1993.
16. Satoshi Matsuoka, Keniro Taura, and Akinori Yonezawa. Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages. In *OOPSLA'93*, pages 109–126. ACM SIGPLAN, ACM Press, 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. Christian Neusius. Synchronizing Actions. In *ECOOP '91*, pages 118–132. Springer Verlag, 1991.
19. Raju Pandey. *A Compositional Approach to Concurrent Programming.* PhD thesis, The University of Texas at Austin, August 1995.
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. Raju Pandey and James C. Browne. A Compositional Approach to Concurrent Object-Oriented Programming. In *IEEE International Conference on Computer Languages.* IEEE Press, May 1994.
22. Raju Pandey and James C. Browne. Support for Extensibility and Reusability in Concurrent Object-Oriented Programming Languages. In *Proceedings of the International Parallel Processing Symposium*, pages 241–248. IEEE, 1996.
23. S. Crespi Reghizzi and G. Galli de Paratesi. Definition of Reusable Concurrent Software Components. In *ECOOP '91*, pages 148–165. Springer–Verlag, 1991.
24. Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, Second Edition edition, 1991.
25. Laurent Thomas. Extensibility and Reuse of Object-Oriented Synchronization Components. In *Parallel Architecture and Languages Europe, LNCS 605*, pages 261–275. Springer Verlag, 1992.
26. 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.
27. Peter Wegner. Dimensions of Object–Based Language Design. In *OOPSLA'87*, page 168. ACM Press, 1987.