A Compositional Approach to Concurrent Programming

by

Raju Pandey, B.Tech.,M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 1995

A Compositional Approach to Concurrent Programming

Approved by
Dissertation Committee:

____

____

____

____

____

To

my late mother

Acknowledgments

This has been a long journey, a journey where there was ample time for me to feel for hope, despair, enthusiasm and sheer sloth. I certainly could not have traveled it all on my own. A lot of people have helped me along the way. First and foremost I would like to thank my advisor, Professor J.C. Browne, for his guidance, his patience with my uncanny ability to make simple things complex, and his intellectual and emotional support. I am indebted to him in many ways. I would also like to thank Dr. John Werth for giving many helpful comments on the dissertation. I also thank the other members of the dissertation committee for their comments.

I am grateful to my colleagues and office mates for cheerfully tolerating my whims. I thank Tim Collins and Mike Kleyn for being cell mates all these years. A lot of thanks to Sowmya Ramachandran for cheerfully listening to me complain about anything and everything. I would also like to thank Charles Callaway for keeping the M&M jar full, though I do hold him responsible for my newfound girth and burgeoning cavities.

My dissertation could not have been completed without the help of the superb administrative and system staff. I want to especially thank Nancy MacMahon and Gloria Ramirez for knowing everything there is to know about this university. I also want to thank the system staff, especially Becky Badgett and Kay Nettle, for helping me by installing everything I kept asking for.

My family has always been a source of strength for me. They have always been supportive about everything I have ever wanted to do. They have loved me, supported me, and have only asked once when I was going to finish. I know that they are proud of me and I am grateful to my father, my brother, and my sister for that. My wife, Poornima, has always been with me in this, always with insights better than anyone else, with patience, and with support. I have been lucky. Finally, the last word is for our daughter, Midushi. Midushi has made me smile, always. She has also helped me find myself in her. I know it is terrible when Papa is never there. But I promise I will make up for the lost time.

RAJU PANDEY

The University of Texas at Austin
August 1995

A Compositional Approach to Concurrent Programming

Publication No. ____________________

Raju Pandey, Ph.D.
The University of Texas at Austin, 1995

Supervisor: James C. Browne

There is widespread interest in concurrent programming. Many concurrent programming languages have been proposed. However, there are modularity and extensibility problems associated with many of these languages: concurrent programs are difficult to extend and modify. Also, specifications of programs cannot be easily reused.

This thesis presents a compositional approach to concurrent programming that aims to resolve these problems. The approach is based on the observation that there are two orthogonal behaviors of components of a concurrent program: computational behavior and interaction behavior. The computational behavior of a component specifies the operations performed during an execution of the component. Its interaction behavior determines the manner in which the component affects or is affected by other components. In the compositional approach a concurrent program is composed from *separate* specifications of computational and interaction behaviors. We call this approach "separation of concerns."

One of the implications of separation is that concurrent programs can be easily extended by adding component programs. Only the specifications of the interaction behaviors may need to be changed. Also, a concurrent program can be modified by changing only its interaction behavior specification. Further, separation of concerns supports reusability of both computational and interaction behavior specifications. More importantly, in this approach representations of computations and interactions are both programming language abstractions. These abstractions can be composed with other abstractions such as inheritance and genericity in order to define generic concurrent program abstractions.

In the first part of the thesis, we use separation of concerns as the basis for developing a model of computation. The model contains a concurrent program composition mechanism, a representation mechanism for component programs, and a declarative interaction specification mechanism. In the second part of the thesis, we apply the model to design a concurrent object-oriented programming language, called CYES-C++. CYES-C++ supports highly concurrent objects and a general concurrent method invocation mechanism, fully integrates the notion of inheritance and genericity with concurrency and interaction, and supports reusability of method and interaction specifications. The feasibility of our approach is demonstrated by developing a prototype implementation for CYES-C++. The implementation runs on a network of workstations.

Contents

Acknowledgments **v**

Abstract **vii**

List of Tables **xiv**

List of Figures **xv**

Chapter 1 Introduction **1**

1.1 Approaches to Concurrent Programming 2

1.2 Our Approach 5

1.2.1 Conceptual Basis for a Model of Computation 6

1.2.2 The C-YES Model 7

1.2.3 Design of a Language Based on the C-YES Model 14

1.2.4 Language Implementation and Experimental Evaluation 15

1.3 Dissertation Overview 16

1.4 Research Contributions 16

Chapter 2 Concepts and Background **19**

2.1 Introduction 19

2.2 Background 20

2.3 Related Work 22

2.3.1 Specification of Concurrency 23
2.3.2 Interaction Specification 27
2.4 Analysis 32
2.4.1 Extensibility and Modifiability of Concurrent Programs 34
2.4.2 Interaction Specification 38
2.5 Summary 39

Chapter 3 The C-YES Model of Concurrent Computation 40

3.1 Introduction 40
3.2 Definitions 42
3.3 Resolution of Program Composition Anomaly 45
3.3.1 Separation of Concerns 46
3.3.2 Implications of Separation of Concerns 46
3.4 The C-YES Model 48
3.4.1 Concurrent Program Composition 48
3.4.2 Representation of Component Programs 49
3.4.3 Interaction Specification Mechanism 53
3.4.4 Formal Semantics of Concurrent Program Composition 61
3.5 Examples 67
3.5.1 Mutual Exclusion 67
3.5.2 Producer/Consumer 67
3.5.3 Dining Philosopher 70
3.5.4 Gaussian Elimination 73
3.6 Discussion 76
3.6.1 Concurrent Program Composition 77
3.6.2 Interaction Specification 82
3.7 Summary 85

Chapter 4 A Compositional Approach to Concurrent Object-Oriented Programming 87

4.1 Introduction . . . 87
4.2 Intra-object Concurrency and Interaction . . . 89
4.2.1 Interaction Behavior Specification . . . 92
4.2.2 Event Set . . . 94
4.2.3 Representation of Interaction Behaviors . . . 95
4.2.4 Examples . . . 95
4.2.5 Representation of Synchronization Primitives . . . 99
4.2.6 Aggregation Anomaly . . . 102
4.3 Inter-object Concurrency and Interaction . . . 104
4.4 Inheritance . . . 105
4.4.1 Inheritance Anomaly . . . 105
4.4.2 Model of Inheritance . . . 107
4.5 Incremental Specification of Interaction Behavior . . . 111
4.6 Summary . . . 114

Chapter 5 CYES-C++ 116

5.1 Introduction . . . 116
5.2 Concurrent Objects . . . 119
5.2.1 Definition of Concurrent Objects . . . 120
5.2.2 Declaration of Concurrent Objects . . . 121
5.2.3 Special Member Functions of Concurrent Classes . . . 123
5.3 Method Invocation . . . 124
5.3.1 Synchronous Method Invocation . . . 125
5.3.2 Asynchronous Method Invocation . . . 125
5.4 Interaction Specification . . . 126
5.4.1 Event Sets . . . 126
5.4.2 Events . . . 128

5.4.3 Event Ordering Constraint Expressions 129
5.5 Inheritance 131
5.6 Generic Concurrent Classes 136
5.7 Summary 140

Chapter 6 Implementation of CYES-C++ 141

6.1 Introduction 141
6.1.1 Problem 141
6.1.2 Design goals and approach 144
6.2 Runtime System 147
6.2.1 Model of Parallel Machines 148
6.2.2 Model of Execution of a CYES-C++ Program 150
6.2.3 Runtime Abstractions 155
6.3 Transformation of CYES-C++ Programs 158
6.3.1 Transformation of Concurrent Class Specifications 159
6.3.2 Transformation of Object Declarations and Method Invocations . . 161
6.3.3 Implementation of Concurrent Objects 164
6.3.4 Execution Behaviors of Method Invocations 165
6.4 Implementation of Event Ordering Constraint Expressions 166
6.4.1 Code Generation 166
6.4.2 Implementation of Normalized Expressions 168
6.5 Summary 172

Chapter 7 Experimental Results 174

7.1 Introduction 174
7.1.1 Experiment Execution Environment 175
7.2 Applications 175
7.2.1 System Overhead Measurements 175
7.2.2 Execution Time Measurements 177

7.2.3 Queue 180
7.2.4 The Gaussian Algorithm 181
7.2.5 The Barnes-Hut Algorithm 183
7.2.6 Parallel Evaluation of value of π 188
7.3 Summary 192

Chapter 8 Summary and Conclusion **194**
8.1 Future Work 197

Bibliography **202**

Vita **215**

List of Tables

5.1 Syntactical representation of event ordering constraint expressions 130

7.1 Thread and synchronization primitive overheads 176

7.2 Execution times of operations such as system termination, object creation and deletion 177

7.3 Method invocation costs 181

List of Figures

1.1 An example concurrent program 3

1.2 Representation of an extension of the example concurrent program 4

2.1 An example concurrent program 33

2.2 A representation of an extended concurrent program 35

2.3 A representation of a modified concurrent program 36

3.1 Representations of interacting programs 51

3.2 Explicit specification of interaction points 52

3.3 A sequential program and one of its computations 54

3.4 Representation of concurrency and interaction in event dependency graphs . 55

3.5 Interaction among programs as a set of relationships among invocations of actions 56

3.6 Application of and constraint operator and a valid computation 58

3.7 Application of or constraint operator and valid computations 59

3.8 Application of `forall` constraint operator and a valid computation 60

3.9 BNF expression for event ordering constraint expressions 62

3.10 Valid computations of two concurrent events 65

3.11 Domination of `produce` events over `consume` events 69

3.12 Single buffer interaction between `producer` and `consumer` 70

3.13 Configuration of dining philosophers and shared forks 71

3.14 Abstract specifications of three dining philosophers 72
3.15 States of dining philosophers 73
3.16 Transformation during a pivot step in the forward elimination step 74
3.17 Representation of the ith pivot step 75
3.18 Interaction between reads and writes of pivots over a matrix element 75
3.19 Specification of forward elimination step 76
3.20 Interaction relationship among events in a computation of extended program 78
3.21 Interaction relationships among events in a computation of modified concurrent program 79
3.22 Parallel implementation of a pivot 82
3.23 Modular derivation of representation of interaction 84

5.1 Concurrent class specification of concurrent `queue` objects 120

6.1 Compilation and linking steps of a CYES-C++ program 145
6.2 The runtime model of a parallel machine 148
6.3 Steps in mapping of an object from one execution space to another 149
6.4 Abstraction of memory models of parallel machines 150
6.5 Execution behavior of a CYES-C++ program 153
6.6 Mapping and unmapping functions for a user defined class 158
6.7 Transformation of `queue` concurrent class 160
6.8 Transformation of concurrent object declarations 162
6.9 Transformation of method invocation expressions 163
6.10 Execution behavior of an object controller 164
6.11 Examples of normalized event ordering constraint expressions 167
6.12 Prefix and postfix execution behaviors of two ordered events 168
6.13 Prefix and postfix execution behaviors of methods 170
6.14 Prefix and postfix execution behaviors of methods 171

7.1 Execution times associated with creation of concurrent objects 178

7.2 Execution times associated with deletion of concurrent objects 179

7.3 Execution times associated with termination of runtime system 180

7.4 Interface of a concurrent class `Matrix` 182

7.5 Execution times for the Barnes-Hut algorithm 187

7.6 Concurrent class specification of a component of a parallel algorithm for evaluation of value of π 190

7.7 Execution times for parallel π algorithms 191

Chapter 1

Introduction

There is widespread interest in concurrent programming due to the emergence of many parallel processing systems. Parallel systems allow one to improve the runtime performance of certain applications by supporting the execution of multiple tasks simultaneously. In some cases, they have made it possible to solve certain grand challenge problems that would have been impossible otherwise. In recent years, a number of parallel systems have been introduced. Examples of such systems include massively parallel systems such as Thinking Machines' CM-5, IBM SP-2 and Intel's Paragon, shared memory systems such as Kendall Square Research's KSR1, Stanford University's DASH and SGI's Challenge, SIMD systems such as Maspar's MP-2, and distributed systems such as collections of Sun and IBM workstations. The machines differ widely in their architecture, their scope, and the target problem domain.

The design and implementation of concurrent programs for this wide range of machines has proven to be extremely difficult. There are a number of factors — nondeterminism, complex interactions among programs, program granularity, data partitioning, data distribution, load balancing, program scheduling, and target machine configurations — that make the design of a concurrent program. An efficient and correct implementation of a concurrent program involves careful analysis of these factors, their interactions, and their representation. The difficulty is exacerbated by the fact that many parallel applications are large and complex.

The complexity associated with the development of concurrent software systems can be reduced by software development methodologies that allow modular and extensible development. Modularity allows one to partition complex systems into smaller and simpler modules, each of which can be developed independently and combined. Support of extensibility is especially important since a complex concurrent system may evolve incrementally through additions of component subsystems. Also, the concurrent system may need to be extended in order to implement new application requirements. Another important aspect of software development is easy modifiability of concurrent programs, since changes in a concurrent program may occur quite often during its lifetime, especially during the program development process. Portability of programs is an important issue in the parallel domain. Since parallel machines vary widely in architecture, and since there is no consensus about a converging architecture, it is important that concurrent programs be easily modified to be able to run on different parallel architectures.

1.1 Approaches to Concurrent Programming

In reality many concurrent programming languages do not support modular and extensible approaches to concurrent programming. In practice, concurrent programs are often difficult to extend and modify. In addition, porting a concurrent program often involves re-implementing the components of the program. There is poor support for reusability of programs as well. These problems occur because of the manner in which concurrent programs are specified.

We observe that in the majority of concurrent programming languages, the general approach to designing and implementing a concurrent program involves partitioning a problem into a number of independent components, each of which is represented as a program. A representation of a component program includes specifications of operations and of synchronization primitives. For instance, in figure 1.1 we show an example of a concurrent program `examprog1`. Program `examprog1` contains two components: `producer`

```
examprog1()
{
   channel buf;

   producer(buf) || consumer(buf);
}
```

(a) Concurrent program

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

(b) Producer component

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

(c) Consumer component

Figure 1.1: An example concurrent program

and `consumer`. The `producer` component produces a value and sends it to `consumer` over a communication channel. The operations `send` and `receive` are synchronization primitives.

We note that concurrent programs that are specified in this manner are difficult to extend or modify. We show this by extending `examprog1` such that it contains an additional `consumer` component. In the extended program, the information produced by the `producer` is now shared between the two `consumer` programs alternately. The extended program can be implemented by modifying `producer`, `consumer`, or both component programs. We show one possible implementation of the extended program in figure 1.2. Here, the `consumer`1 program is a modified version of the `consumer` program of figure 1.1(c). In `consumer`1, the two `consumer` components synchronize with each other by accessing a `sync` mailbox. We note the following:

- Interaction between the `consumer` components is implemented procedurally by an

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

(a) Modified consumer program

```
examprog2()
{
  channel buf;
  channel sync;

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

(b) Extended examprog1

Figure 1.2: Representation of an extension of the example concurrent program

additional synchronization primitive, program variables, and language constructs.

- The extension of the program requires that some or all component programs be re-implemented.

The above illustrates some weaknesses in these concurrent program design methodologies:

- Concurrent programs are difficult to modify and extend since changes and extensions may require that some of its components be re-implemented.

- Specifications of component programs are not encapsulated. Changes in a concurrent program often impact component specifications.

- This phenomena underlines the problem associated with constructing new concurrent program abstractions in terms of existing program abstractions. Concurrent program abstractions cannot be composed easily with other program abstractions. Such compositions may often require changing the abstraction itself. We call this phenomenon the *program composition anomaly*: the inability to easily construct new concurrent program abstractions from existing program abstractions. Also, since programming languages use many composition mechanisms for defining abstractions in terms of other abstractions, presence of the program composition anomaly causes a breakdown in many of these composition mechanisms. As we shall see later, one example is the breakdown of inheritance in concurrent object-oriented languages.

- Specifications of components cannot be reused easily. In addition, synchronization code cannot be reused because it is embedded inside the body of the component program.

The process of modifying or extending a concurrent program becomes difficult also because of the procedural approaches to interaction specification. Any modification in the interaction of a component involves operational reasoning with the specification of the component followed by procedural manipulation of its source code in order to represent the desired interaction. Since components can interact in complex and nondeterministic ways, such modifications are difficult to make, and are often major sources of bugs in concurrent programs.

Program composition mechanisms are needed that resolve the program composition anomaly and that facilitate extensibility and modifiability of concurrent programs.

1.2 Our Approach

This research is aimed at facilitating development of concurrent programs. The focus is on support for software development techniques for minimizing the possible changes that need to be made when a concurrent program is modified.

The conceptual foundations of our approach are based on the observation that modifications and extensions of concurrent programs are facilitated if specifications of computations and interactions are separated. We call this approach “separation of concerns.” This separation of concerns concept forms the basis for the development of a concurrent programming system. The steps in the research include:

- Development of the conceptual and theoretical foundation of a model of concurrent computation.
- Application of the model of computation to design a concurrent object-oriented programming language.
- A prototype implementation of the programming language.
- An experimental evaluation of the implementation.

We briefly describe these various aspects of our research below.

1.2.1 Conceptual Basis for a Model of Computation

There are two distinct behaviors of components of a concurrent program: *computational behavior* and *interaction behavior*. The computational behavior of a component specifies the operations performed during an execution of the component. Its interaction behavior, on the other hand, determines the manner in which the component affects or is affected by other components. *Concurrent programs are difficult to extend and modify because specifications of component programs intermingle specifications of both behaviors*. The result is that changes in a concurrent program (either by extension or modification) may induce changes in interaction behaviors of components. However, since specifications of components include specifications of both behaviors, changes in the interaction behaviors can be effected only by re-implementing the components.

Concurrent programs can be easily extended or modified if specifications of computational and interaction behavior are separated. The requirement for the separation highlights the orthogonality of the two behaviors. Conceptually, the computational behavior of a component is an intrinsic property. It exists independently from the component's possible compositions with other components. Its interaction behavior, on the other hand, is dependent on other components of a concurrent program. It should therefore be defined separately from the computational behavior.

We use the concept of "separation of concerns" to define a model of concurrent computation, called the *Composition bY Event Specification*(C-YES) model, which we describe below.

1.2.2 The C-YES Model

The C-YES model supports a modular and extensible approach to concurrent programming. There are three elements of the C-YES model: i) a concurrent program composition mechanism, ii) a representation of component, and iii) an interaction specification mechanism.

Concurrent Program Composition

The concurrent program composition mechanism of the C-YES model is based on the observation that the role of a concurrent program composition mechanism is to establish two kinds of relationships among operations of components: *concurrency* and *interaction*. Concurrency represents semantic independence among specific invocations of operations (called events). Interaction, on the other hand, represents semantic dependencies among events. A concurrent program written in the C-YES model has the following form:

$$\boxed{\begin{array}{ll} C = & (C_1 \parallel C_2 \parallel \ldots \parallel C_n) \\ & \quad\quad \texttt{where} \\ & \quad\quad\quad\quad \phi \end{array}}$$

This expression specifies that program C is composed from components $C_1, C_2, \ldots,$ and C_n and an interaction specification ϕ. The $\parallel$ operator is used to establish the concurrency relationship whereas expression ϕ represents the interaction relationships among the components. The semantics of the expression is that during an execution of C, events of $C_1, C_2, \ldots,$ and C_n occur in parallel, except for those events whose occurrences must satisfy all ordering constraints specified in ϕ.

Example 1.2.1. *(Specification of a concurrent program).* We specify concurrent program `examprog1` of figure 1.1(a) as a constrained concurrent program composition of components `producer` and `consumer`:

```
examprog1 = (producer ∥ consumer)
                    where
                        ConsExp1
```

This expression specifies that during an execution of `examprog1`, events of `producer` and `consumer` occur in parallel except for those which must satisfy the ordering constraints imposed by the expression `ConsExp1` (defined in example 1.2.3). ■

Representation of Component Programs

In the C-YES model, a component program is represented using two elements: operations and *interaction points*. The interaction points of a component are used to name operations of one component that may interact with other components. Interaction among components is represented by specifying the manner in which an interaction point of a component may affect or be affected by an interaction point of another component. We call such a representation of a component program an *interacting program*.

Example 1.2.2. *(Interacting program).* Interacting program representations of components `producer` and `consumer` are shown below:

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

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

The `producer` program interacts with its environment during executions of the `produce` operations. It is at these operations that `producer` may affect execution behaviors of other components. The set of `produce` invocations therefore form its interaction points.

Similarly, the set of all `consume` invocations form the interaction points of `consumer`. Let the terms `produce` and `consume` denote these interaction points. ■

The interacting program model of component programs therefore allows one to include a component program in different concurrent program compositions without making any changes to the component. Each composition may define a different interaction behavior for the component by specifying different interactions to its interaction points.

Interaction Specification Mechanism

Interaction in the C-YES model is defined by an algebraic expression called an *event ordering constraint expression*. The interaction specification mechanism is declarative. It is based on the following observations:

- Interactions among component programs can be defined by a set of interactions among events of the components, and
- interactions between two events can be represented by directly specifying execution orderings between the events.

The interaction specification mechanism therefore contains mechanisms for representing interaction between events, and a set of operators for combining representations of interaction relationships. An event ordering constraint expression (evoce) is constructed from a set of *primitive ordering constraint expressions* and a set of *interaction composition operators*.

Primitive Event Ordering Constraint Expression: A primitive ordering constraint expression represents interaction between two events. The expression

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

specifies that event e_1 must occur before event e_2 in a computation.

Interaction Composition Operators: Interaction composition operators are used to combine primitive and non-primitive event ordering constraint expressions. There are four operators:

- and constraint operator ($\textcircled{\wedge}$): An event ordering constraint expression containing $\textcircled{\wedge}$ is defined:

$$\phi = (\phi_1 \textcircled{\wedge} \phi_2)$$

Intuitively, a computation satisfies event ordering constraint expression ϕ if it satisfies both ϕ_1 and ϕ_2.

- or constraint operator($\textcircled{\vee}$): An event ordering constraint expression containing $\textcircled{\vee}$ is defined:

$$\phi = (\phi_1 \textcircled{\vee} \phi_2)$$

Intuitively, a computation satisfies and event ordering constraint expression ϕ if it satisfies *at least one* of event ordering constraint expressions ϕ_1 or ϕ_2.

- forall operator: The `forall` constraint operator is an extension of the $\textcircled{\wedge}$ operator in that it is used to specify and-constraints over sets of events.

- exists operator: The `exists` operator is similar to `forall` in that it extends $\textcircled{\vee}$ over sets of events.

Example 1.2.3. *(Interaction specification).* We now derive the event ordering expression `ConsExp1` of example 1.2.1. It is specified by first examining the interaction between two specific occurrences of `produce` and `consume`. The constraint specifies that the ith occurrence of `consume` cannot execute until the ith occurrence of `produce` has executed. The following primitive event ordering constraint expression represents the relationship between the two events:

$$\boxed{\texttt{produce[i]} \textcircled{<} \texttt{consume[i]}}$$

In the above terms `produce[i]` and `consume[i]` respectively denote the ith invocations of the `produce` and `consume` operations. Since the above relationship holds true for all invocations of `produce`, the event ordering constraint expression `ConsExp1` is defined in the following manner:

```
ConsExp1 =  forall occ  i in  produce :
                 produce[i] ⧀ consume[i]
```

Expression `ConsExp1` constrains the order of execution of invocations of the `produce` and `consume` operations. ■

We make the following observations about the interaction specification mechanism:

- The interaction specification mechanism is declarative. Interactions are represented as ordering relationships of invocations of operations. An event ordering constraint expression is a predicate over a range of variables and operators. An execution of a concurrent program constrained by the event ordering constraint expression must maintain the expression as "true" throughout its execution.

- The interaction specification mechanism supports a modular approach to interaction specification. Global and complex interactions are specified by decomposing them into a set of simpler interactions between pairs of components. These interactions can then be represented by event ordering constraint expressions, and combined with suitable interaction composition operators to represent the global interaction. This approach allows one to change interaction behavior of a program by changing only the relevant event ordering constraint expression. Also, representations of interactions can be reused in specifications of other interactions.

- The interaction specification mechanism provides an abstraction suitable for defining interaction. It defines interaction by suitable ordering relations among interacting events of programs. It is not based on the semantic properties of a specific synchronization primitive. It does not depend on the semantic properties of the events. It can therefore be used to specify any interaction behavior for any invocation of any operation.

- Since specifications of interactions of components are separated from the component specifications, it can be combined with other abstraction mechanisms such as inheritance and genericity to construct powerful concurrent program abstractions.

- Since the properties of the primitive event ordering constraint expression and the interaction composition operators are well defined, it is possible to formally verify certain safety and progress properties of a concurrent program from its interaction behavior specification. In addition, the verification process is facilitated by the separation of computational and interaction behavior specifications: some properties such a mutual exclusion of events can be verified solely from the interaction behavior specification.

Example 1.2.4. *(Extensibility of concurrent programs).* We now show that concurrent program `examprog1` can extended easily in the C-YES model. Program `examprog2` is an extension of `examprog` in that it contains an additional `consumer` component. In the extended program, the two `consumer` programs share the information produced by the `producer` programs alternately. Program `examprog2` is defined in the following manner:

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

Note that there are no changes in the specifications of either `producer` or `consumer`. Let `consume`1 and `consume2` denote the interaction points of the two consumer components. They denote the set of invocations of the `consume` operation in each `consumer` program.

The event ordering constraint expression `ConsExp2` represents the new interaction relationship among the three components. It is derived by observing that there are two sets of relationships among the events of the `producer` and `consumer` programs. The first is between odd events of `produce` and events of one `consumer`. It is represented in the following manner:

$$\boxed{\phi_1 = (\texttt{produce[2*i-1]} \circledless \texttt{consume1[i]})}$$

The second is between even events of `produce` and events of the other `consumer`:

$$\boxed{\phi_2 = (\texttt{produce[2*i]} \circledless \texttt{consume2[i]})}$$

Since both of these relationships must hold, they are combined with the $\textcircled{\wedge}$ operator:

$$\boxed{\phi_1 \textcircled{\wedge} \phi_2}$$

Since the above relationship holds for all events of `produce`, `ConsExp2` is defined in the following manner

$$\boxed{\begin{array}{l}\texttt{ConsExp2 = forall occ } i \texttt{ in produce :} \\ \qquad\qquad\qquad\qquad \phi_1 \textcircled{\wedge} \phi_2\end{array}}$$

■

We make the following observations regarding the above approach to concurrent programming:

- A concurrent program can be extended by adding new components and changing interaction specifications. No changes need to be made in the specifications of the existing component programs. In certain cases, redefinition of interactions may only involve adding new event ordering constraint expressions or modifying only a small subset of the event ordering constraint expressions.

- A concurrent program can be easily modified by modifying interaction behavior specifications of the components.

- The approach supports encapsulation of specifications of component programs. It allows one to change implementation of components in isolation from other components. For instance, it is possible to change the implementation of the `producer` program without making any changes in the `consumer` program or the interaction behavior specification as long as the nature of computation, interaction behavior, and the interaction points do not change.

- Specifications of computational and interaction behaviors can be reused.

1.2.3 Design of a Language Based on the C-YES Model

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 object-oriented concepts in order to derive a compositional model for concurrent object-oriented programming. The object-oriented compositional model forms the basis for the design of a concurrent extension of the C++ programming language, called CYES-C++. CYES-C++ supports truly concurrent objects, implements a general concurrent method invocation mechanism, fully integrates the notion of inheritance with concurrency, and supports reusability of both method and interaction specifications. We briefly describe the characteristics of CYES-C++ below:

Intra-object Concurrency and Interaction: In CYES-C++, a concurrent object is represented as a composition of a set of methods and a set of event ordering constraint expressions. Specifications for methods contain specifications of only their computational behaviors. The event ordering constraint expressions capture the interactions among the invocations of the methods. The semantics associated with the composition is that invocations of methods on a concurrent object execute in parallel by default. However, their executions must satisfy all ordering constraints specified in the event ordering constraint expressions. In CYES-C++, the composition of a concurrent object is represented by a concurrent class type. The interface of a concurrent class contains, in addition to `public`, `private`, and `protected` entities of C++ classes, `interaction` entities. The `interaction` section defines the event ordering constraint expressions.

Inter-object Concurrency and Interaction: Concurrency and interaction among concurrent objects is represented as a concurrent program composition of the invoking and the invoked methods. This defines a concurrent method invocation mechanism. The method invocation mechanism is general in that it subsumes both synchronous and future-based asynchronous method invocation mechanisms.

Inheritance: In CYES-C++, inheritance is a mechanism for extending the composition of concurrent objects. A concurrent class therefore extends the composition of its superclasses by adding new methods, modifying inherited methods, adding interaction behaviors, and/or modifying interaction behaviors. The composition of objects of a concurrent class is defined by the composition of methods and composition of event ordering constraint expressions of the class and its superclasses. This model of inheritance allows us to inherit and reuse both computational and interaction behavior specifications.

Interaction Specification Mechanism: Event ordering constraint expressions represent interactions in terms of event sets and relationships between events of the sets. Abstractions of interactions can therefore be defined by supporting mechanisms for defining and manipulating event sets, and for naming and instantiating event ordering constraint expressions. CYES-C++ defines mechanisms for defining event sets and event ordering constraint expressions. This allows one to capture patterns of interactions among event sets in terms of named event ordering constraint expressions. The named event ordering constraint expressions can be instantiated with different event sets, each representing interactions among different sets of methods. This supports reusability of interaction specifications.

Further, the abstractions of interaction can be combined with other composition mechanisms such as the template mechanism of the C++ programming language. The concurrent program abstractions allow one to capture common computational and interaction behaviors of concurrent classes in generic and abstract concurrent classes. This supports reusability of both computational and interaction behaviors.

1.2.4 Language Implementation and Experimental Evaluation

We implemented a translator for CYES-C++ in order to demonstrate the feasibility of implementation of the C-YES model of concurrent computation. The current implementation executes on a network of IBM RS/6000 workstations. It supports creation and distribution of concurrent objects both on local and distributed nodes. In addition, both synchronous and asynchronous method invocations on local and remote objects have been implemented. Event ordering constraint expressions containing `forall`, `&&`, and primitive event ordering constraint expressions are also supported in the current version.

We evaluated the implementation by writing several applications and measuring their execution behavior. The measurements show that the performance of the CYES-C++ implementation can be comparable to other approaches to concurrent object-oriented programming.

1.3 Dissertation Overview

In the rest of the dissertation we develop and apply the compositional approach to concurrent programming. In Chapter 2 we present concepts of and background information on concurrent programming. We also discuss several approaches to concurrent programming. We show that concurrent programs are difficult to extend and modify in some of these approaches. The reasons for the difficulties are established in Chapter 3. In that chapter we also present the conceptual and theoretical basis for the model of computation, the C-YES model. We present a concurrent program composition mechanism, a representation for components, and a representation for interaction specification. In Chapter 4 we present an application of the C-YES model to derive a model for object-oriented concurrent programming. We develop a model for concurrent objects, a general mechanism for method invocation, and a model of inheritance in the presence of concurrency. The object-oriented model forms the basis for the design of a concurrent programming language called CYES-C++. The language features of CYES-C++ are described in Chapter 5. The design of an implementation of CYES-C++ is described in Chapter 6. We evaluated the implementation by writing a number of applications, and by measuring their performance. The applications and the analysis of the performance results are described in Chapter 7. We summarize our research in Chapter 8. We also describe ideas for future work in that chapter.

1.4 Research Contributions

The following are the contributions of our research:

- The concept of the program composition anomaly and its resolution. We show that a concurrent program is difficult to extend and modify if specifications of its components contain specifications of both computational and interaction behaviors. The concurrent program can be extended and modified easily if the specifications of the two behaviors are completely separated.

- The formulation of a model of concurrent computation that is based on the concept of "separation of concerns." The model includes a concurrent program composition mechanism, a representation mechanism for component programs, and a declarative interaction specification mechanism.

- The design of a modular and compositional approach to interaction specification mechanism. The interaction specification mechanism captures fundamental abstractions of interaction. It specifies interaction by suitable ordering relations and their compositions. It does not depend on the semantics of specific primitives. It can be used to specify any interaction behavior for any invocation of any operation.

- Demonstration of the utility of the "separation of concerns" and the declarative approach to interaction specification through examples. We show that the approach supports a modular and extensible approach to concurrent programming. In addition, it supports reusability of both computational and interaction behavior specifications.

- An application of the C-YES model to derive a model for concurrent object-oriented programming. It includes the following:

  - Development of a concurrent program representations of both intra- and inter-object concurrency through the concurrent program composition mechanism of the C-YES model.

  - Presentation of occurrences of two instances, *inheritance anomaly* and *aggregation anomaly*, of the program composition anomaly in concurrent object-oriented programming languages. We show that their resolutions can be derived from the principles used for resolving the program composition anomaly, namely separation of computational and interaction behavior specifications.

  - Development of a model of inheritance which defines the manner in which concurrent program composition of a class can be derived from superclass and class specifications. The model can be used to represent the model of inheritance of most concurrent object-oriented programming languages.

- Development of an object-oriented programming language. The programming language CYES-C++ is derived from the concurrent object-oriented model. It supports truly concurrent objects, implements a general concurrent method invocation mechanism, fully integrates the notion of inheritance with concurrency, and supports reusability of both method and interaction specifications. The language supports definition of abstractions for interactions and concurrency that can be composed with other abstraction mechanisms such as inheritance and template to construct generic and abstract concurrent classes. Such classes support reusability of both method and interaction specification. In addition, they raise the level of abstraction at which concurrency is expressed.

- Design and implementation of a portable translator for CYES-C++. The implementation includes a front end parser, a code generator, and a runtime library. It currently runs on a network of IBM RS6000 workstations.

- Evaluation of the execution characteristics of the CYES-C++ implementation by developing a number of applications and measuring the performance of the applications. Careful analysis shows that the performance behavior of the implementation can be improved significantly by modifying different aspects of the implementation.

Chapter 2

Concepts and Background

2.1 Introduction

There are two goals of this chapter: the first is to present background information on concurrent programming and some of the approaches to concurrent programming. The second is to analyze the existing approaches, and present the motivation for our research.

Design and implementation of concurrent programs is inherently complex. There are a number of factors such as nondeterminism, complex interactions among programs, program granularity, data partitioning, data distribution, load balancing, program scheduling, and target machine configurations that drive the design of a parallel program. An efficient and correct implementation of a concurrent program involves careful analysis of the factors, the interaction among them, and their representation. In addition, concurrency is generally used to represent problems that are large and computationally intensive. It is therefore imperative that concurrent programming languages support software development methodologies that allow modular and extensible development of such large and complex concurrent software systems.

We observe that in most concurrent programming languages, the general approach to designing and implementing a concurrent program involves partitioning a problem into a number of independent components, each of which is represented as a program. A representation of a component includes specifications of both operations and interactions with other components. Also, interactions are specified in terms of synchronization primitives and combined through operational mechanisms.

We show that concurrent programs specified in this manner are difficult to extend and modify. Changes and extensions of a concurrent program may require that some or all components be re implemented. Specifications of component programs are therefore not encapsulated. More importantly, this phenomena underlines the problem associated with composing new concurrent program abstractions in terms of existing program abstractions. Such compositions may often require changing the abstraction itself. We call this phenomenon the *program composition anomaly*: the inability to easily construct new concurrent program abstractions from existing program abstractions. Also, since programming languages use many composition mechanisms for defining abstractions, presence of the program composition anomaly causes breakdown in many of these composition mechanisms. One example is the breakdown of inheritance in concurrent object-oriented languages. An implication of the program composition anomaly is that specifications of components may not be easily reused. Different composition mechanisms are needed that resolve the program composition anomaly and that facilitate extensibility and modifiability of concurrent programs.

This chapter is organized as follow: in Section 2.2, we present background information on concurrent programming. Section 2.3 contains a brief survey of the related work. In Section 2.4, we analyze the various approaches with respect to extensibility, modularity, and reusability of components of a concurrent program. Section 2.5 contains a summary of the chapter.

2.2 Background

The process of programming can be viewed as that of constructing new abstractions in terms of existing abstractions. Programming languages support this process by defining a set of i) primitives, and ii) mechanisms, which we will call *composition mechanisms*, that can be used for composing the primitives and other existing abstractions for specifying new abstractions. Most programming languages support two kinds of composition mechanisms: *data composition* and *program composition*. Examples of data composition mechanisms are aggregation and inheritance. An example of a program composition mechanism is the sequential program composition mechanism used in most imperative sequential programming languages [Set94].

There are two aspects of a composition mechanism: one is the syntactical mechanisms for representing a composition. Both textual and graphical representation schemes have been used for specifying syntax of data and program compositions. The second is specification of semantics. Semantics of a composition assigns a "behavior" to an abstraction in terms of behaviors of its components. This behavior can describe memory layout, execution orderings, or other properties of an abstraction. For instance, the semantics associated with a sequential program composition of two programs `progA` and `progB`

```
[ progA; ProgB]
```

specifies that during an execution of the composition, `progA` executes before `progB`. Also, certain properties of the program composition can be derived from the properties of `progA` and `progB` [Hoa69].

We now look at the notion of concurrent program composition. A concurrent program models a collection of interacting entities. The entities exist independently, perform operations, and may cooperate by exchanging information to work on a common goal, or may compete to access shared resources. In either case entities may affect execution behaviors of other entities. We call this aspect of a concurrent system *interaction* — ability of an entity to affect or be affected by other entities.

A concurrent program composition mechanism defines mechanisms for representing such entities as programs as well as mechanisms for combining programs such that both concurrency and interaction among the programs can be specified. In addition, many concurrent programming languages provide mechanisms for representing attributes such as data distribution, data partitioning, program granularity, scheduling, underlying machine configuration, and memory hierarchy. Many of these attributes implicitly determine concurrency and interaction patterns among programs. Concurrency and interaction are the two fundamental aspects of concurrent program composition. We will focus our attention only on them.

We observe that the role of a concurrent program composition mechanism is to establish two kinds of relationships among operations of component programs. The first, *concurrency*, represents semantic independence among operations of components. During an execution of a program, concurrent operations can execute in parallel. The second, *interaction*, defines semantic dependencies among operations. Semantic dependencies specify the manner in which the components cooperate or compete. For instance, an operation of one component may rely on information produced by an operation of another component. Similarly, policies associated with implementation of a resource shared among different components may ensure that requests for resource are accepted fairly. In all of these cases interaction among components represents relationships among operations of the components. The relationships are application specific, and — implicitly or explicitly — affect execution behaviors of components. A correct execution of a concurrent program is one that preserves all relationships specified by its concurrent program composition.

We use concurrent program composition as a specification of concurrency and interaction relationships as a basis for analyzing different approaches to concurrent program specification below.

2.3 Related Work

We have divided this section into two subsections. We first present the approach taken by some other programming languages for specifying concurrency among components. We then examine the different approaches for specifying interaction.

2.3.1 Specification of Concurrency

Concurrency among components can be specified either *explicitly* by annotating components as concurrent, or by deriving concurrent relationships among components *implicitly* from the semantics of a representation mechanisms. We look at the explicit approaches first.

Explicit Specification of Concurrency

The simplest approach for specifying concurrency is to support creation of independent execution threads of control. Such threads, once created, execute independently and in parallel. For instance, the UNIX operating system [RT78] supports creation of independent streams of execution, called *processes*, by `fork` and `join` primitives. A process consists of a private address space and an execution control. Since process creation and process context-switching costs are high, this notion of processes cannot be used in massively parallel applications that tend to create thousands of independent streams of control of execution. Lightweight "threads" decouple the notion of address space from control of execution. They can therefore be created, manipulated, and destroyed cheaply. Examples of shared memory based thread packages are Presto [Ber91], Mercury [Ber91], and CThreads [CR88]. Examples of thread packages that allow threads to be created on distributed nodes and that support communication among these threads are Nexus [FGT94] and Chant [HCM94]. Both processes and lightweight threads support powerful mechanisms for specifying concurrency among programs. However, they are low level mechanisms and are difficult to use.

A structured approach for specifying concurrency is the parbegin-parend [Dij65] construct. Here a statement of the form

```
parbegin
        s1  s2  ...  sn
parend
```

explicitly identifies statements s_1, s_2 $\cdots$ and s_n as concurrent. The parbegin-parend statement terminates once all s_i have terminated. Languages such as CSP [Hoa78], PCN [FOT92, CT92], and CC++ [CK92] have similar constructs. The parbegin-parend construct is limiting in the sense that it allows only a static number of concurrent components. Constructs such as `parfor` [CK92] and `doall` [AG94, Ost86] extend the parbegin-parend construct in that they allow creation of dynamic number of components.

Another approach that explicitly specifies concurrency is the Path expression [CH74, And79]. Here, an operator explicitly identifies procedures of an abstract data type that can execute in parallel. For instance, the expression

$$\boxed{\{ \texttt{ P1 } \}}$$

specifies that different invocations of procedure `P1` are concurrent.

Implicit Specification of Concurrency

In many programming languages, concurrency among component programs is implicit in the semantics of the programming language. Examples of such languages are logic, functional, object-oriented, and visual concurrent programming languages.

Logic Programming Languages: In logic programming languages [Rob92], programs are represented in terms of goals and sets of clauses. Each clause has the form:

$$\boxed{A \leftarrow B_1, \ldots, B_n}$$

Here A is a predicate and is called head of the clause. Terms B_1, B_2, $\ldots$, and B_n form the body of the clause. They are represented by predicates as well. The semantics of the clause is that A is true if all body predicates are true. A computation involves determining if a goal (a predicate) is a logical consequence of a set of clauses. Concurrency among components of logic program occurs in two ways [Con87, CG86]:

- *And parallelism* is based on the observation that predicates such as B_i are independent and can be evaluated in parallel.

- *Or parallelism* arises from the fact that validity of a goal may be determined by a number of clauses. Evaluations of the clauses can therefore occur in parallel.

Examples of logic-based languages are Prolog and Concurrent Prolog [Sha86]. A comprehensive survey of logic programming languages can be found in [Sha89].

Functional Programming Languages: Concurrency is specified implicitly in functional programming languages [Bac78] as well. Functional programming languages are based on lambda calculus in that programs are defined as pure functions. Different evaluations of a function with same parameters always return identical results. There are no side effects. Evaluations of parameters are therefore semantically independent: they can be occur in parallel. Examples of functional programming languages are Miranda [Tur85] and Haskell [HWA$^+$90]. Hudak [Hud89] provides a survey of different functional programming languages.

Object-oriented Programming Languages: Object-oriented programming languages [Weg87] also provide a natural basis for modeling concurrency among entities of applications. Such entities exist independently. Also, they 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 [TS89a]. Several concurrent programming languages have used the concept of encapsulated "object" as a common basis for specifying concurrency. For instance, the concept is used in i) rendezvous-based languages such as ADA [Geh84] and server-based approaches such as RPC-based languages [BN84] and; ii) approaches based on message passing such as CSP [Hoa78]; iii) approaches based on abstract data types (ADT) such as Monitors [Hoa74], ADT with path expressions [CH74], and SR [And81]; iv) approaches based on objects such as POOL-T [Ame87], ABCL/1 [YBS87], Concurrent SmallTalk [YT87], CC++ [CK92], Mentat [Gri93], and Charm++ [KK93]; and v) actor-based approaches [Agh86]. Most ADT- or object-based languages have used either a *passive* or an *active* approach to represent concurrency among programs:

- Passive view: In passive object approaches [Hoa74, CH74, Car93b], the notion of concurrent threads of executions is added to a sequential object-oriented programming language. Here, objects are passive entities and are used solely for data abstraction and encapsulation purposes. Languages support mechanisms for specifying concurrency explicitly.

- Active view: In active object-based approaches [CK92, Ame87, KK93, CL89] objects are both units of execution and encapsulation. Active object-based languages are more natural since they model entities which are independent and which permit concurrent activities. External concurrency is supported by concurrently executing objects (inter-object concurrency), whereas internal concurrency is supported by allowing multiple execution threads within an object(intra-object concurrency):

  - Intra-object concurrency: Languages such as POOL-T [Ame87], Concurrent Smalltalk [YT87], and ABCL/1 [YBS87] 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. Languages such as Path Expression [CH74], CC++ [CK92], PO [CL89], and Mediator [GC86], on the other hand, support concurrent method invocations within objects.
  - Inter-object concurrency: Concurrency among objects is specified by method invocation mechanisms. In most concurrent programming languages, method invocation 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, allows calling and called methods to execute in parallel. The calling method blocks if it tries to access a future object which has not yet been created by the called method. The interaction between the calling and the called methods is limited to reads and writes over the future variable.

Visual Programming Languages: In visual programming languages such as CODE [Bro85, BAS85, NB92] and VISAGE [KZ93]. concurrent programs are represented as directed graphs. Nodes of a graph denote component programs, whereas edges represent interactions among the components. Concurrency is modeled implicitly by the fact that programs associated with nodes are independent, and hence can execute in parallel.

Summary: Concurrent programming languages use both explicit and implicit approaches for specifying concurrency among programs. Implicit approaches are more natural in that they capture more semantic information about an application. However, explicit approaches provide more control over concurrent program structures that can be specified.

2.3.2 Interaction Specification

We view interaction among components of a concurrent program as a set of semantic relationships among operations of the components. An interaction specification mechanism therefore contains two parts: i) mechanisms for specifying interaction relationships between specific operation, and ii) mechanisms for combining the relationships in order to represent the interaction between programs. In this section we look at various interaction specification mechanisms by highlighting the different approaches to specifications of the two parts. We first look at the mechanisms used for representing specific interaction relationships.

Interaction Relationships Specification

We classify interaction relationship specification approaches into two types: *primitive-based interaction specification* and *ADT interaction specifications*.

Primitive-based Interaction Specification: In primitive-based approaches, interaction is defined by identifying a synchronization primitive of a type, say S, and a set of operations, say o_1 and o_2, on an instance of S. The semantics of the primitive determines a specific interaction relationship between invocations of operations o_1 and o_2. Interaction among different programs is specified by using objects of type S and by invoking operations o_1 and o_2 on the variables suitably. We present many instances of such primitives below:

- Semaphores: In Semaphores [Dij65, Dij68b], operations o_1 and o_2 respectively are `P` and `V` operations on semaphore variables. Interaction between invocations of `P` and `V` is determined by the constraint that invocations of `P` are delayed until corresponding `V` invocations have occurred.

- Communication primitives: Many languages support operations such as `send` and `receive` for communication. These operations serve two purposes: i) transfer information from one node to another, and ii) act as a synchronization primitive. For instance, in synchronous communication channels [Hoa78, Per87, Mil80] invocations of `send` and `receive` on a channel variable are delayed until both can occur. In asynchronous channels [Sun90, AO93], on the other hand, every `receive` invocation is delayed for a corresponding `send` invocation. There are no constraints on invocations of `send` invocations.

- Logic variables: In logic-based concurrent programming languages, variables shared among different clauses form the basis for specifying interaction relationships. The clause

  $$\boxed{A(X) \leftarrow B_1(X), \ldots, B_n(X)}$$

  holds true for all values of variables X. Hence, during an evaluation of a predicate B_i, if the variable X is assigned a certain value, it must be assigned the same value in the head and the body predicates of the clause. This implies that a shared variable is written only once in an evaluation. Further, concurrent threads trying to read X are delayed until X has been written. Many programming languages such as PCN [FOT92] and CC++ [CK92] have used the write-once shared variables to specify interaction among concurrent programs.

- Dataflow: A data dependency-based interaction relationship occurs in functional programming languages. Here, reads of a parameter are delayed until the expression associated with the parameter has been evaluated. Here, interaction occurs between reads and writes over parameter variables.

- Tuple space operations: In Linda [Gel85, ACG86] interaction among programs is specified by operations over a global tuple space. Programs place information in a tuple form in the tuple space. They retrieve tuples by searching the tuple space. Interaction is specified by relationships between reads (`in`) and writes (`out`) of tuples: i) `in` and `out` invocations are atomic, and ii) invocations of `in` are delayed until a corresponding `out` has occurred.

- Critical regions: Critical regions and conditional critical regions [Hoa72, Bri72b, Bri73, Bri72a] are used to assign unconditional or conditional mutual exclusion properties to a region with respect to other regions.

ADT Interaction Specifications: Here, synchronization primitives are used for specifying interaction among procedures/methods of abstract data types [Par72] or objects. Interaction among methods arises when they access common resources and data structures. They represent certain semantic dependencies such as data dependency, data consistency, and priority among method invocations. We place the different approaches to ADT interaction specification into three categories:

- The first is the languages that use traditional synchronization primitives such as locks and semaphores [CGH92, BS93], write-once-read-many variables [CK92], data flow based data dependencies [Gri93], and signal variables [Hoa74] for specifying interaction among methods. In these approaches, methods include these primitives in order to specify interaction.

- The second is approaches such as enable-based approaches [Geh93, Neu91, Tho92, MWBD91, DKM$^+$89, GC86], disable based approaches [Fro92, SG91], and behavior abstraction based approaches [KL89, TS89b, Mat93]. Here a set of boolean conditions is associated with each method. A method is delayed until the associated boolean conditions are true (in enable based approaches) or false (in disable based approaches). Here, a conditional interaction relationship exists between a method invocation and invocations of other methods of a class.

- The third is approaches that use regular expression and temporal logic expressions for specifying interaction. For instance, in Path Expression [CH74, And79] interaction among methods is represented as regular expressions. A regular expression here can be thought of as representing a set of strings containing invocations of methods as symbols. A string represents a labeled total order of executions of methods [LSC81]. In SYSL [RK83], temporal logic operators such as `always`, `until`, and `eventually` are used to assert relationships among method invocations.

Composition of Interaction Relationships

We now look at different mechanisms for combining interaction relationships. We emphasize that it is essential that interaction specification mechanisms support techniques for representing complex interactions as compositions of simpler interaction specifications. This allows one to develop representations of interactions by partitioning a complex interaction into a set of simple interactions, developing representations for the simple interactions and combining them with suitable composition mechanisms. The modular approach to interaction specification forms the basis for both modifiability and reusability of interaction specifications. We enumerate the different approaches below:

- Most concurrent programming languages use procedural mechanisms for combining different interactions among operations. Here, operations on synchronization primitives are like any other procedure call. Different interaction relationships therefore are represented by procedurally manipulating the synchronization primitives, program variables and the sequential program composition mechanisms such as `if` and `while`. However, since most procedural composition mechanisms are deterministic, many programming languages support additional constructs such as `select` and `accept` in order to compose interaction relationships nondeterministically. For instance, in ADA [Geh84] the `accept` statement is used to select, *nondeterministically,* one entry for execution from among many possible entries. Nondeterministic selection constructs are derived from the notion of guarded commands first proposed by Dijkstra. They occur in different forms in languages such as SR [AOC+88], Concurrent C [GR86], and Mentat [Gri93].

- In [Rep88] synchronization is a first class value. It is constructed from primitives (`send` or `receive` over a channel), and composition operators such as “select” and “filter.” The select operator allows one to choose a synchronization value from many values, whereas the filter operator is used to select synchronization values on the basis of certain boolean conditions. Once a synchronization value has been constructed, it can then be applied with a `sync` operator. This approach allows one to construct interaction relationships by constructing synchronization values and combining them with suitable operators. This approach is limited in that synchronization values can only be constructed from a fixed set of primitives (send and receive) which have predefined synchronization properties. This limits the possible kinds of interaction behaviors that can be constructed.

- In path expressions, the composition mechanism is based on regular expression operators. However, Bloom [Blo79] shows that path expressions do not support modular development of interaction specifications because path expressions do not contain general mechanisms for directly representing certain states of objects, and for specifying interactions that depend on the states.

2.4 Analysis

We now analyze the various approaches to concurrent programming. We do so by first establishing the nature of parallel software development support that concurrent programming languages must provide.

Design and implementation of concurrent programs is inherently complex. There are a number of factors such as nondeterminism, complex interactions among programs, program granularity, data partitioning, data distribution, load balancing, program scheduling, and target machine configurations that drive the design of a parallel program. An efficient and correct implementation of a concurrent program involves careful analysis of the factors, interactions among them, and their representation. In addition, concurrency is frequently used to represent problems that are large and computationally intensive. It is therefore imperative that concurrent programming languages support software development methodologies that allow modular and extensible development of large and complex concurrent software systems.

Modularity allows one to partition complex systems into smaller and simpler modules, each of which can be developed independently and then combined suitably. Support for extensibility is especially important because a complex concurrent system may evolve incrementally through additions of component subsystems. Also, it may be extended in order to implement new application requirements. Another important aspect of software development is easy modifiability of concurrent programs, since changes in a concurrent program may occur quite often during its lifetime, especially during the program development process. In all of these cases, a concurrent programming language must support a software development methodology that allows for changes and extensions in concurrent programs to be easily carried out.

Portability of programs is also an important issue in the parallel domain. Since parallel machines vary widely in architecture, and since there is no consensus about a converging architecture, it is important that concurrent programs be easily modified to be able to run on different architectures. Our view of portability of concurrent programs is that both

```
examprog1()
{
   channel buf;

   producer(buf) || consumer(buf);
}
```

(a) Concurrent program

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

(b) Producer component

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

(c) Consumer component

Figure 2.1: An example concurrent program

portability and optimal efficiency of concurrent programs cannot be achieved at the same time. The reason is that the optimal execution behavior of a concurrent program is driven by a number of factors such as program scheduling, machine configuration, memory hierarchy, and interconnection network. These factors vary significantly from one machine to another, and hence influence the design and structure of a concurrent program. The problem of portability in such cases is, therefore, reduced to that of minimizing the possible changes one needs to make in order to port a program from one parallel machine to another.

We focus our discussion on two aspects of concurrent software development in traditional concurrent programming languages: i) approaches for concurrent software development, and ii) approaches for interaction specification. Both have direct implications for modular and extensible development of programs as well as modifiability of the programs.

2.4.1 Extensibility and Modifiability of Concurrent Programs

We observe that in a majority of concurrent programming languages, the general approach to designing and implementing a concurrent program involves partitioning a problem into a number of independent components, each of which is represented as a program. A representation of a component includes specifications of both operations and interactions with other components. We examine the extensibility and modifiability of such concurrent programs. We do so by taking a simple example concurrent program, and trying to extend and modify it in various ways.

A Simple Concurrent Program

Let `examprog1` be a concurrent program containing two components. One of the components produces certain information, while the other consumes information. Semantic dependency exists between the components. It specifies that information can consumed only after it has been produced. We show a representation of `examprog1` and its components `producer` and `consumer` in figure 2.1.

The components interact with each other through `send` and `receive` primitives over a communication channel. The channel is a FIFO asynchronous channel; it is a mailbox [And91] where programs can deposit and retrieve information in a FIFO manner. Note that specifications of the components include operations such as `produce` and `consume`, and synchronization operations such as `send` and `receive`.

Extensibility of Concurrent Program

We extend `examprog1` by adding another `consumer` component, for example because the `consumer` program is slow relative to the `producer` program. In the extended program, information produced by the `producer` is now shared between the two `consumer` programs alternately. The extended program can be implemented by modifying `producer`, `consumer`, or both components. We show one possible implementation of the extended program in figure 2.2. Here, `consumer1` is a modified version of `consumer` of fig-

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

(a) Modified `consumer` program

```
examprog2()
{
   channel buf;
   channel sync;

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

(b) Extended `examprog1`

Figure 2.2: A representation of an extended concurrent program

ure 2.1(c). Operations send and receive on the `sync` mailbox are used for representing the interaction between the two `consumer` programs. We make note of the following:

- Interaction between the `consumer` components is implemented procedurally through additional synchronization primitives, program variables, and language constructs.
- The extension requires that the `consumer` component be re-implemented.

Modifiability of Concurrent Programs

We now look at the modifiability of `examprog1`. We do that by defining additional constraints between the `producer` and `consumer` components: there are at most N unconsumed values. The `producer` component therefore must wait for the `consumer` component if there are N unconsumed values. This program is implemented by re-implementing both `consumer` and `producer` components. One possible implementation is shown in figure 2.3. In the implementation, the `consumer` program sends an acknowledgement after every received message.

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

(a) Modified concurrent program

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

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

(b) Modified `producer` (c) Modified `consumer`

Figure 2.3: A representation of a modified concurrent program

Conclusion: *Changes and extensions of a concurrent program may require that some or all components be re-implemented.* For instance, a modification of `examprog1` required that both `producer` and `consumer` (figure 2.3) be modified.

Implications

The above underlines many weaknesses in the concurrent program design methodology. The first is that concurrent programs are difficult to modify and extend since changes and extensions may require that some of components be re-implemented. A concurrent programming language must provide mechanisms that encapsulate different components of a concurrent system so that changes in components or additions of components do not involve changing some or all existing components of the concurrent program. However, as we saw earlier, changes in concurrent programs are often visible in most components. Specifications of component programs are therefore not encapsulated. Also, changes in components often involve making modifications in existing source code. Such modifications in source programs are error prone. Indeed, they are one of major sources of bugs in concurrent programs.

This example also illustrates the problem associated with constructing new concurrent program abstractions in terms of existing program abstractions. A concurrent program represents an abstraction of an activity of the application world, just as a sequential program does. However, unlike other composition mechanisms such as sequential program composition mechanism, concurrent program abstractions cannot always be composed easily with other program abstractions. Such compositions may often require changing the abstraction itself. We call this phenomenon the *program composition anomaly*: the inability to easily construct new concurrent program abstractions from existing program abstractions. Also, 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. One example is the breakdown of inheritance in concurrent object-oriented languages. We describe these breakdowns in Chapter 4.

One of the major impacts of the program composition anomaly is that specifications of components cannot be reused easily. For instance, in three different versions of `examprog1`, much of the behavior of the producer and consumer component remains unchanged. However, we were forced to create different versions of the components by duplicating much of code from one version to another. In addition, synchronization code cannot be reused because it is embedded inside the body of the program.

The program composition anomaly also inhibits porting concurrent programs across parallel architectures. It is usually not possible to define a concurrent program that executes *optimally* on parallel machines with different architectures. Different versions of the program must be constructed that use architecture-specific parameters such as program granularity, interaction patterns, and underlying machine configuration to achieve efficiency. Development of concurrent programs for different architectures can be facilitated if the programs can be derived easily by composing components that are common to the different architectures. However, in the traditional concurrent programming approaches, porting the program from one architecture to another often requires major changes in many components.

Finally, the conventional approach to development of concurrent systems is not modular. Specifications of components are closely coupled. Design and implementations of components therefore must be done with other components in mind.

2.4.2 Interaction Specification

We now analyze the approaches to interaction specification. We note that in most concurrent programming languages, interaction is usually specified in terms of synchronization primitives such as send-receive, semaphores, and write-once variables. Also, procedural mechanisms are used for composing the synchronization primitives so that a specific relationship can be specified. For instance, in the implementation of `examprog2` (figure 2.2), variables such as `myTurn` and `myId` along with certain procedural control flow constructs are used for implementing additional constraints between the two consumer components.

Interaction relationships among operations of components are therefore specified indirectly through synchronization primitives. For instance, in order to define an interaction relationship between operations o_1 and o_2, one must specify suitable primitives that execute before and/or after o_1 and o_2 in order to represent the interaction relationship between the operations. This approach to interaction specification requires one to reason about the specifications of the operations in order to specify an interaction. In cases where concurrent systems may contain large number of components interacting in complex ways, this approach is clearly inadequate. Also, formal reasoning with programs specified in such a way is difficult. In addition, most approaches do not support modular development of interactions because of lack of a powerful composition mechanisms for combining interaction relationships. Interactions are therefore difficult to specify, modify, and cannot be easily reused.

2.5 Summary

We presented concepts of concurrent programming. We also presented a brief survey of different approaches to concurrent programming. In the majority of concurrent programming approaches, specifications of components of a concurrent program include both operation and synchronization primitives. We showed that concurrent programs specified in this manner are difficult to extend and modify: changes in such programs require that some or all components be re-implemented. Also, concurrent program abstractions cannot be easily constructed from other concurrent program abstractions. One of the implications of this phenomenon is that specifications of component and interaction may not be reused easily.

Chapter 3

The C-YES Model of Concurrent Computation

3.1 Introduction

In the previous chapter we showed that concurrent programs are difficult to modify and extend. Changes and extensions in a concurrent program require that some or all components be re-implemented. More importantly, the phenomenon underlines the problem of the program composition anomaly: concurrent program abstractions cannot be composed easily from other program abstractions. Such compositions may often require changing the program abstractions themselves. In addition, presence of the program composition anomaly causes breakdowns in other composition mechanisms such as aggregation and inheritance. We also showed that most approaches to interaction specification are based on a set of synchronization primitives. Interaction specifications often involve manipulation of program variables, synchronization primitives, and computations operationally. It is difficult to specify interaction through such interaction specification mechanisms. In this chapter, we examine the reasons for the program composition anomaly. We also present a solution for its resolution. We then develop the conceptual and theoretical foundation of a model of concurrent computation that is based on the solution.

The program composition anomaly arises because specifications of component programs include specifications of interactions. Changes in a concurrent program may change interactions among its components. Since interactions are embedded inside the components, such changes require modifying the implementations of the components. The program composition anomaly can therefore be resolved by separating the specifications of interaction from the specifications of the components.

We use the concept of separation to derive a model of concurrent computation, called the C-YES model. In the C-YES model, specifications of computations and interactions are completely separated in a concurrent program representation. There are three elements of the C-YES model:

(a) Concurrent program composition: The model defines a composition mechanism for specifying concurrent programs. Here a concurrent program is specified by composing *separate* specifications of components and their interactions. Such a specification of concurrent program is easier to modify and extend. For instance, additions of components may only require changes in the specifications of interaction.

(b) Representation of component programs: The C-YES model defines a representation of component programs. In this representation, a component program is represented by operations it performs, and a set of entities called *interaction points*. Interaction points denote operations where a component may interact with other components. Interaction among components is represented by specifying the manner in which the interaction points (operations) of a component may affect or be affected by the interaction points of other components. Note that such a model of component programs forms the basis for including a component program in different concurrent program compositions, where each program composition defines a different interaction for the component by specifying suitable interactions for its interaction points.

(c) Interaction specification mechanisms: The C-YES model contains a declarative mechanism for specifying interactions. The interaction specification mechanism contains a primitive for specifying execution ordering relationships, and a set of logical operators (such as and, or, forall, and exists) for combining the relationships. The approach to interaction specification here is to first partition complex interactions among component programs into a set of simpler interactions between pairs of component programs. The simpler interactions can be represented as relationships among the interaction points of the programs, and then combined to represent the global interaction. This approach is the basis for easy modifiability and reusability of interactions specifications.

This chapter is organized as follows: We define a number of terms in Section 3.2. In Section 3.3 we examine the reasons for the modularity and extensibility problems. We also present the manner in which they can be resolved. In Section 3.4, we give the details of the C-YES model. Section 3.5 contains a number of examples that illustrate the manner in which concurrent programs can be specified in the C-YES model. We analyze the C-YES model with respect to extensibility and modifiability of concurrent programs in Section 3.6. Section 3.7 provides a summary.

3.2 Definitions

There are three components [Sch86] of a programming language - i) syntactic mechanisms for representing programs, ii) semantics for interpreting a user program, and iii) pragmatics used for evaluating applicability, suitability, and efficiency of the language. In this section we establish the syntactic and semantic mechanisms that we use for specifying and interpreting the C-YES model.

Most programming languages identify a set of primitive abstractions and composition mechanisms for defining program or data abstractions in terms of other — primitive or non-primitive — abstractions. An example of a composition mechanism is the sequential program composition mechanism. We use the following terms to denote syntactic representations of programs.

Definition 3.2.1. *(Action).* An action is an identifiable operation. ■

Actions represent the alphabet that a programmer uses for constructing a program. For instance, operation `produce` denotes an action.

Definition 3.2.2. *(Program).* A program represents composition of primitive and nonprimitive actions. ■

An example of a program is the `producer` program shown in figure 2.1. The `producer` program is composed from the `produce` action and the `while` program composition mechanism. A program may contain both sequential and concurrent compositions of actions. We use the syntactic mechanisms of the C++ programming language [Str91] for specifying the sequential aspects of a program or an action.

We now define a number of terms that we use for specifying executions of programs.

Definition 3.2.3. *(Computation).* A computation is an execution of a program. ■

Every program has a (possibly infinite) set of computations associated with it.

Notation: Let terms $\xi_s(C)$ and $\xi(C)$ respectively denote the set of computations and some computation of program C. Note that $\xi(C) \in \xi_s(C)$.

Definition 3.2.4. *(Event).* An event is an identifiable occurrence of an action in a computation. ■

The relationship between an action and an event is shown by the following program:

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

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

Notation: We represent an event in a computation by `Action[Selector]`. Here, the term `Selector` is used to uniquely identify an occurrence of `Action`.

We use the notion of *event occurrence number* as a selector. An event occurrence number, i, of an event specifies that the event is the ith invocation of an action in a given computation. For instance, term `produce[i]` denotes the ith invocation of `produce` action in a computation of `examprog`. Note that relative orderings in occurrence numbers merely specify the order in which invocations of an action occur; they do not suggest that executions of the invocations are serialized. For instance, it is possible for an event, say `X[5]`, to occur before an event, `X[4]`, in a computation.

Definition 3.2.5. *(Event Ordering).* The event ordering relation $<$ between two events is an asymmetric, non-reflexive, and transitive relation. The relation $e_1 < e_2$ specifies that e_1 occurs before e_2. ■

Relation $<$ models execution ordering between two events. It is identical to relation *happens before* [Lam78].

Definition 3.2.6. *(Concurrent Event).* Events e_1 and e_2 are concurrent if there is no ordering relation between the events. ■

Notation: We use $(e_1 \parallel e_2)$ to denote concurrency between events e_1 and e_2.

$$e_1 \parallel e_2 = \neg((e_1 < e_2) \vee (e_2 < e_1))$$

Definition 3.2.7. *(Pomset [Pra86, Gai88]).* A labeled partial order is a 4-tuple $(V, \Sigma, <, \mu)$ consisting of i) a set V of events, ii) an alphabet Σ of actions, iii) a partial order $<$ on set V, and iv) a labeling function $\mu : V \to \Sigma$ assigning symbols to events, each labeled event representing an occurrence of the action labeling it. Each $(V, \Sigma, <, \mu)$ arises from a computation. ■

Notation: We will write $(V, <)$ for pomsets when Σ and μ can be derived from the context.

Notation: We will use $e \in V$ to mean that e is an event in computation $(V, <)$.

Definition 3.2.8. *(Event Dependency Graph).* An event dependency graph $G = (V, E)$ is a directed graph such that i) V is a set of events, and ii) $(e_1, e_2) \in E$ iff $e_1 < e_2$. ■

Each event in V denotes a vertex, whereas each $<$ relation between two events is represented as an edge between two vertices. We use event dependency graphs to visually represent computations.

3.3 Resolution of Program Composition Anomaly

We first explore the reason for the program composition anomaly. We observe that there are two distinct behaviors of components of a concurrent program: *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 in `examprog` is to produce certain values. The interaction behavior of a component determines the manner in which the component affects or is affected by other components. For instance, the interaction behavior of the consumer component in `examprog` specifies that every invocation of the `consume` action is dependent on a preceding invocation of the `produce` action.

The program composition anomaly arises because specifications of component programs contain specifications of both — computational and interaction — behaviors. The reason is that changes in a concurrent program (either by extension or modification) may induce changes in interaction behaviors of its components. However, since specifications of the components include specifications of both behaviors, changes in the interaction behaviors can be effected only by re-implementing the components. For instance, computational behaviors of the `producer` and `consumer` components remains unchanged in different variations of the `examprog` program; only their interaction behaviors change. However, one must create different versions of the components because the specifications of the interaction behavior are embedded in the specifications of the components.

3.3.1 Separation of Concerns

We now present the general characteristics that a concurrent program composition mechanism must provide in order to support easy modifiability of concurrent program as well as a resolution of the program composition anomaly. *The program composition anomaly can be resolved if specifications of computational and interaction behavior are separated*. The requirement for the separation highlights the orthogonality of the two behaviors. We think of the computational behavior of a component as its intrinsic property. It exists independently from the component's possible inclusions in different concurrent programs. For instance, the role of the `producer` component is to produce certain value. It is independent of the fact that it can be combined with a single consumer, multiple consumers, or even another producer. The intrinsic property — producing information — does not change. Its interaction behavior, on the other hand, is dependent on other components of a concurrent program. It should therefore be specified separately from the specifications of the computational behavior, and when the concurrent program is defined.

On the basis of separation of concerns, we can define a concurrent program as a composition of two entities: specifications of computational behavior of its components and specifications of interaction behavior among the components. Let program `examprog` be represented in the following manner:

$$\texttt{examprog} = \langle S, I \rangle$$

In this expression, $S = \{\texttt{producer, consumer}\}$. The expression specifies that program `examprog` is composed from components `producer` and `consumer`, and interaction I between the components. Specifications of `producer` and `consumer` contain specifications of only their computational behaviors.

3.3.2 Implications of Separation of Concerns

Separation of specifications of the two behaviors has direct implications on the modifiability of concurrent programs, reusability of both computational and interaction behavior specifications, and the concurrent program design methodology. For instance, `examprog` can be easily extended by defining a composition of the form:

$$\texttt{examprog1} = \langle S \cup \{\texttt{consumer}\}, I_1 \rangle$$

In the above, I_1 represents interaction among the `producer` and the two `consumer` programs. Neither the `producer` nor the `consumer` component needs to be modified. Similarly, `examprog` can be easily modified:

$$\texttt{examprog2} = \langle S, I_2 \rangle$$

In the above, I_2 represents the modified interaction behavior between the components.

The above approach to concurrent program composition allows us to encapsulate the specifications of component programs. For instance, it is possible to change the implementation of a component in isolation from other components. Modifications of a concurrent program are therefore localized in that only specifications of interaction behaviors may change when computational behaviors of components change.

Separation of specifications of the two behaviors also supports reusability of component programs. For instance, once separation of specifications of computational and interaction behavior is made, specifications of `producer` and `consumer` are reused in different versions of `examprog`. Also, since specifications of interaction behaviors are not embedded inside the bodies of components, they can be reused in definition of concurrent program compositions as well.

The separation of concerns approach to concurrent program composition emphasizes a concurrent program design methodology where program design involves identifying components whose computational behaviors do not change over different implementations of a concurrent program. Such an approach allows construction of different versions of a program quickly from core components. Also changes in components can be isolated to a subset of computational and interaction behavior specifications, thereby reducing the possibility of introducing bugs.

3.4 The C-YES Model

The C-YES model is a model of concurrent computation. It is based on the concept of separation of concerns. It supports a modular and extensible approach to concurrent programming. There are three elements of the model:

- It defines a concurrent program composition mechanism in which specifications of computational and interaction behaviors are completely separated.
- It contains an extended model of component programs in order to incorporate specifications of their interaction behavior.
- It contains a declarative approach to interaction specification.

We first describe the composition mechanism.

3.4.1 Concurrent Program Composition

Definition 3.4.1. *(Constrained concurrent program composition).* The expression

$$\boxed{\begin{array}{l} C = \; (C_1 \parallel C_2 \parallel \ldots \parallel C_n) \\ \qquad\qquad \texttt{where} \\ \qquad\qquad\quad \phi \end{array}}$$

specifies a concurrent program C. Program C is composed from components $C_1, C_2, \ldots,$ and C_n and interaction specification ϕ. ■

The above definition underlines the separation of computational and interaction behavior specifications. Components $C_1, C_2, \ldots,$ and C_n contain specifications of their computational behaviors only. Interaction among the components is defined *separately* by an expression ϕ. The above definition is based on the observation that the role of a concurrent program composition mechanism is to establish two kinds of relationships among events of computations: *concurrency* and *interaction*. Concurrency represents semantic independence among the events. These events can occur in parallel. Interaction, on the other hand, represents semantic dependencies among the events. Semantic dependencies arise because an event may depend on the information produced by another event (data dependency), events must access a resource in an ordered manner (resource consistency and fairness), or events must satisfy other application specific semantic constraints.

The $\|$ operator in the constrained concurrent program composition expression is used to establish the concurrency relationship whereas expression ϕ is used to define the interaction relationships among the components $C_1, C_2, \ldots,$ and C_n. The semantics of the composition, therefore, is that events of computations of $C_1, C_2, \ldots,$ and C_n are concurrent by default. Hence, during an execution of C, they may occur in parallel. However, there are certain events that interact with each other. Occurrences of these events must satisfy all ordering constraints specified by the expression ϕ.

The approach to program design in the C-YES model therefore is to partition a concurrent program into a set of components, define their computational behaviors, identify events that interact, and specify interaction relationships among them.

3.4.2 Representation of Component Programs

Given that specifications of component programs do not include specifications of interaction behavior, we focus our attention on formulating a model of component programs that provides an answer to the question: how are component programs represented so that their interaction behaviors can be specified?

We construct a model of component programs by observing the execution behavior of a component: during execution of a concurrent program, the component repeatedly performs certain action. Occasionally, it interacts with its environment (other components) during the execution of certain actions. We call these actions *interaction points*. For instance, in programming languages such as CSP [Hoa78] components are represented by processes. A process interacts with other processes by sending and receiving messages on communication channels. The sends and receives are the actions where processes interact and, thus, form the interaction points of the processes.

We observe that there are two elements of an interaction point: the first is its identification, which determines the action at which a program may interact. In CSP, for instance, names of communication channels along with the send and receive actions identify the interaction points of a process. The second is its *role* in an interaction. The role of an interaction point determines the manner in which a program participates in an interaction at the interaction point. For instance, the role associated with a synchronous "receive" interaction point determines a process's behavior at the interaction point: the process is delayed until a message arrives.

In the C-YES model, the two elements of an interaction point — its identification and its role — are separated. Interaction points of a program are identified when the program is specified. However, the roles of the interaction points are determined when the program is composed with other programs by the concurrent program composition expression. A component program in the C-YES model is represented by its computations and interaction points. We call such programs *interacting programs*.

We now look at the mechanisms that we use for defining interacting programs. The approach taken here is to extend the notion of sequential programs in order to capture interaction points of programs. A specification of a sequential program has two components: i) interface of the form $f_s(p_1,\ p_2,\ \ldots,\ p_n)$, where variables $p_1, p_2, \ldots,$ and p_n parameterize the execution behavior of f_s; and ii) an implementation specifying its computational behavior. We extend this specification in the following manner:

1. Interface: An interacting program f_c has an interface of the form

$$f_c(p_1, \ldots, p_n\ ;\ i_1, \ldots, i_m)$$

   Here $p_1, p_2, \ldots,$ and p_n are parameter variables. Interaction parameters $i_1, i_2, \ldots,$ and i_m denote interaction points of f_c.

2. Implementation: An implementation of an interacting program is partitioned into two parts. The first specifies computational behavior of the program. The second is a

mapping between the interaction parameters $i_1, i_2, \ldots,$ and i_m and the actions of the program.

Note that a sequential program is a special instance of an interacting program in that it identifies — implicitly — two interaction points: i) *entry* point, where control and values of the parameters are passed and ii) *exit* point, where the block terminates and returns any values. All other interaction points are encapsulated.

We now look at the nature of interaction points and how they are specified. Since events form the basis for specifying interactions in the C-YES model, an interaction point represents a set of events. There are two ways in which the mapping between the interaction parameters and set of events can be specified:

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

(a) Representation of `producer`

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

(b) Representation of `consumer`

Figure 3.1: Representations of interacting programs

- Implicit: In this approach, interaction points of a component are derived from the parameter variables: all actions on objects denoted by the variables are the interaction points of the component. (We assume that the parameters represent objects). For instance, we show interacting program representations of `producer` and `consumer` in figure 3.1. Note that they do not contain any interaction behavior specifications. Interaction points of the `producer` program are represented by the term `info.produce()`, which denotes the set of all `produce` events associated with the `producer` program in a computation.

- Explicit: In this approach, mappings between the interaction parameters and the sets of events are specified by *explicitly* labeling the sets of events. Interacting program representations of `producer` and `consumer` are shown in figure 3.2. The interaction parameters `prod` and `cons` explicitly name sets of events of the programs.

```
producer(buffer info; prod)
{
 computation:
  while (TRUE) {
   info.produce();
  }
 interaction point:
  prod names info.produce;
}
```

(a) Representation of `producer`

```
consumer(buffer info; cons)
{
 computation:
  while (TRUE) {
   info.consume();
  }
 interaction point:
  cons names info.consume;
}
```

(b) Representation of `consumer`

Figure 3.2: Explicit specification of interaction points

The implicit mapping approach has the advantage that additional language mechanisms for specifying interaction points are not needed. Also, the scope rules of the parameters can be used to specify the scope rules of the interaction points. However, approaches that use implicit mappings do not have precise control over the visibility of interaction points: either all method invocations on an object are visible or none are. Also, it is not possible to capture an event that is not a method of a parameter object. This limits the kinds of interactions that can be specified. The explicit approach, on the other hand, provides precise control over the visibility of interaction points. However, this approach requires additional language mechanisms for specifying the labeling and the scope rules of interaction points.

We chose the implicit approach for specifying interaction points in CYES-C++ (see Chapter 5) for two reasons: the first is that we wanted to minimize the possible changes in the base language C++ of CYES-C++. The second is that since most interactions occur among actions over objects that are either parameter objects or global objects, the implicit approach can be used to represent all such actions as interaction points.

Two interacting programs $f_c(v_1,\ldots,v_n\ ;\ i_1,\ldots i_m)$ and $g_c(w_1,\ldots,w_l\ ;\ j_1,\ldots,i_k)$ can thus be composed by the constrained concurrent composition mechanism to specify a concurrent program:

```
$f_c(v_1,\ldots,v_n) \parallel g_c(w_1,\ldots,w_l)$
    where
        $\phi$
```

Here, interaction specification ϕ is defined over the interaction points of programs f_c and g_c.

3.4.3 Interaction Specification Mechanism

We now describe the interaction specification mechanism of the C-YES model. Unlike many interaction specification approaches, where interaction among programs is specified in terms of a set of synchronization primitives and is composed through procedural mechanisms, the approach here is declarative and compositional: complex interactions among programs are specified by first partitioning the interactions into a set of simpler interactions between pairs of component programs. Each of these interactions is individually represented and then combined to represent the global interaction. We will show that this approach leads to a modular development of interaction specification. Also, it forms the basis for reusing specifications of interactions.

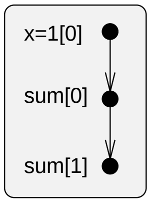
We first present the conceptual foundations of the interaction specification mechanism. We then describe the interaction specification mechanism.

Approach

We describe our approach to interaction specification by examining the concept of concurrent program composition in terms of computations of programs, and the role synchronization primitives play in a concurrent program composition. We first look at the sequential program composition mechanism. A sequential program, such as the one shown in figure 3.3(a), contains a set of actions and combines these actions by the `;` and `for` composition mechanisms. A computation of this program is shown as an event dependency graph

```
x = 1;
for (i = 1; i < 2; i++)
    sum();
```

(a) A sequential program



(b) A computation

Figure 3.3: A sequential program and one of its computations

in figure 3.3(b). The role of the sequential composition mechanisms is to order occurrences of events. For instance, operator "`;`" determines that event `(x:= 1)[0]` occurs before the `sum` events, whereas operator `for` determines that `sum[0]` occurs before `sum[1]`.

We now look at the role that concurrent program composition plays in defining a concurrent program: assume that a concurrent program C is composed from components C_1 and C_2. Specifications of the components contain both actions and synchronization primitives. Figure 3.4 shows an event dependency graph associated with a specific computation of C. Event dependency graphs $\xi(C_1) = (V_1, <_1)$ and $\xi(C_2) = (V_2, <_2)$ respectively represent specific computations of C_1 and C_2 in the computation of C. Note that there are two kinds of edges in G: one is the edges between events of a single computation and other is the edges between events of different computations. Edges between the events of a single computation are shown in the figure by the solid lines. For instance, the edge between e_1 and e_3 shows the order in which they occur. These orderings are specified by the computational behavior specifications of the component programs.

The second are the edges that exist between the events of $\xi(C_1)$ and $\xi(C_2)$ are shown by the dashed lines. These edges specify the order in which the events of $\xi(C_1)$ and $\xi(C_2)$ execute with respect to each other. For instance, occurrences of the events e_1 and f_1 are ordered. We call such edges *interaction edges*. The interaction edges represent orderings among events so that occurrences of the events preserve certain data dependency, data consistency, mutual exclusion, and other semantic constraints. For instance, events e_2

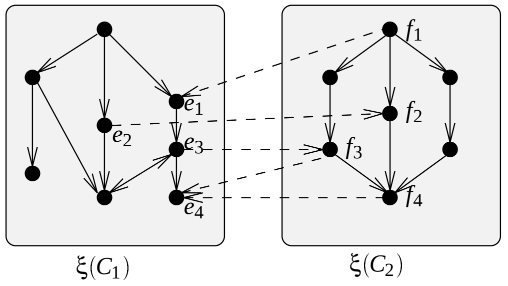


Figure 3.4: Representation of concurrency and interaction in event dependency graphs

and f_2 may respectively be read and write events over common data. Also, the application may specify the constraint that a read event happens only after a write event (thereby representing data dependencies among the events). We observe that the notion of execution orderings among events can be extended to the concurrent domain as well: in addition to the orderings between the events of a computation, orderings may exist between events of computations of different components as well. One of the goals of operations over synchronization primitives is to create these orderings. The program therefore must include synchronization primitives such that

$$e_2 < f_2$$

in a computation of the program.

Our approach to interaction specification, therefore, is based on the following two observations:

- *interaction between two events can be represented by directly specifying execution orderings between them,* and
- interaction among programs can be defined by a set of relationships among events. For instance, interaction among C_1 and C_2 is defined by relationships between e_1 and f_1, e_2 and f_2, e_3 and f_3, e_4 and f_3, and e_4 and f_4.

Further, we can partition the interaction edges among the events according to the actions that the events belong to. As shown in figure 3.5, a_1, a_2, b_1 and b_2 denote the actions

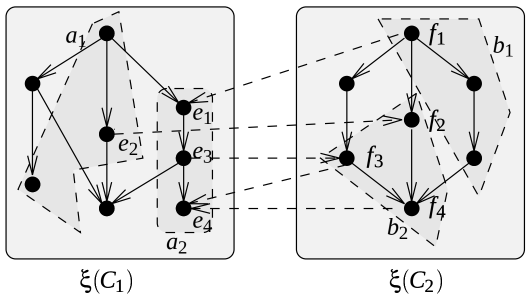


Figure 3.5: Interaction among programs as a set of relationships among invocations of actions

that the events correspond to. Interaction among programs can, therefore, be defined by specifying relationships among the occurrences of the actions. For instance, occurrences of a_2 are related to occurrences of both b_1 and b_2. This approach to interaction specification can be generalized for specifying interaction among programs in the following manner:

- Decompose interaction among components of a concurrent as a set of local relationships between pairs of programs.

- Identify local relationships among sets of events of two components. Each local relationship is a set of relationships between event sets. For instance, the interaction between C_1 and C_2 can be represented by a set of relationships among the event sets a_1, b_1, a_2 and b_2 as shown in figure 3.5.

- Identify and represent the relationships among the events of the events sets. Combine the relationships between the events to represent the relationship among the event sets.

- Combine the relationships between the event sets to define the local interactions.

- Finally, combine the representations of the local interactions to represent the global interaction.

The interaction specification mechanism can therefore be defined by defining mechanisms for representing relationships between events, and by a set of operators for combining representations of different relationships. We present one such interaction specification mechanism below.

Event Ordering Constraint Expressions

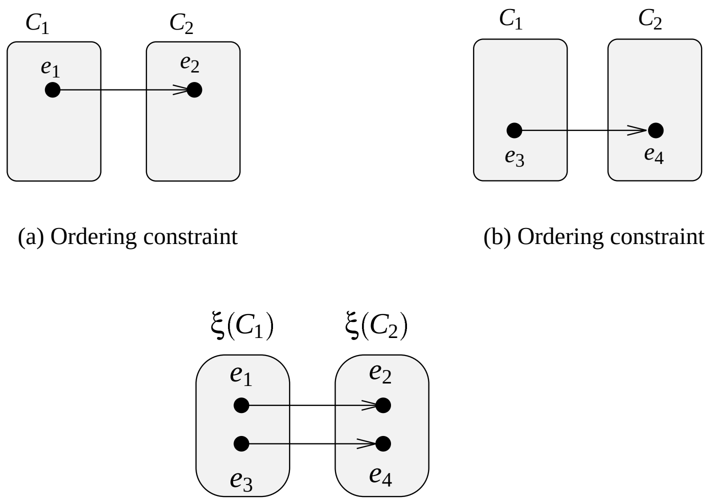
Interaction among programs in the C-YES model is specified by an algebraic expression, called the *event ordering constraint expression*. It is used to represent semantic dependencies among events of component programs by specifying execution orderings — deterministic or nondeterministic — among the events. An event ordering constraint expression (evoce) is constructed from a set of *primitive ordering constraint expressions* and a set of *interaction composition operators*. A primitive ordering constraint expression captures interaction between two events, whereas the interaction composition operators are used to represent nondeterministic interactions as well as interactions among sets of events.

Primitive Event Ordering Constraint Expression: A primitive event ordering expression defines the interaction relationship between specific occurrences of two actions. It imposes constraints on execution orderings of two events. A primitive event ordering constraint expression is represented as:

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

A computation satisfies ϕ if e_1 occurs before e_2 in the computation. The order of execution between the events is determined by issues such as data dependency, safety, and progress properties. Primitive ordering constraint expressions allow one to capture interactions when they are translated to the most primitive level of computations, that is, events. For instance, e_1 may denote an occurrence of a write action, whereas e_2 may denote an occurrence of a read action. The ordering relationship therefore represents data dependency between the events.

Interaction Composition Operators: Interaction composition operators are used to combine primitive and non-primitive event ordering constraint expressions to construct more complex expressions. A primitive ordering constraint expression specifies a deterministic ordering relationship between two events. In real applications both nondeterministic interac-



(a) Ordering constraint (b) Ordering constraint

(c) A computation

Figure 3.6: Application of and constraint operator and a valid computation

tions, and interaction among sets of events are possible. We present a set of operators that can be used to construct event ordering constraint expressions that represent such interactions.

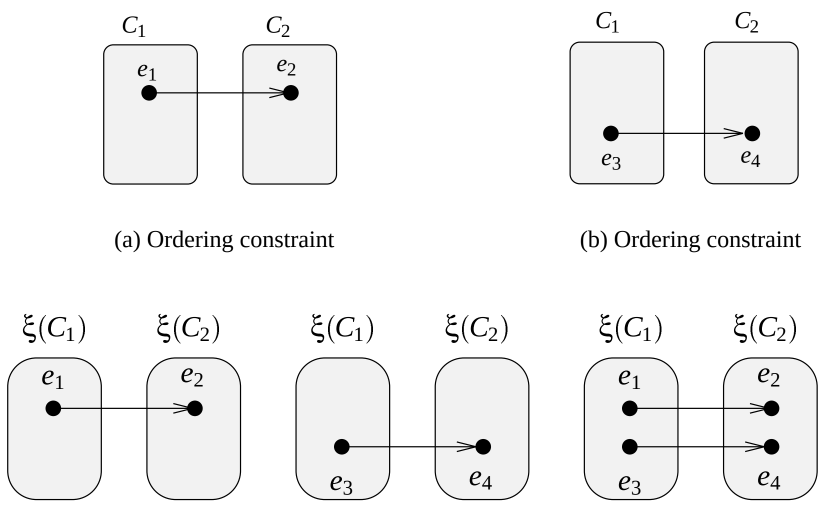
- And constraint operator ($\owedge$): The and constraint operator $\owedge$ is used for combining event ordering expressions in order to represent interactions among sets of events. It allows one to express the fact that a set of interaction relationships holds in a computation. An event ordering constraint expression containing $\owedge$ is defined:

  $$\phi = (\phi_1 \owedge \phi_2)$$

  Intuitively, a computation satisfies event ordering constraint expression ϕ if it satisfies both ϕ_1 and ϕ_2.

  Example 3.4.1. *($\owedge$ operator).* Let events of program C_1 and C_2 be constrained by two ordering relationships as shown in figure 3.6(a) and 3.6(b):

  $$\phi = (e_1 \olessthan e_2) \ \owedge \ (e_3 \olessthan e_4)$$



(a) Ordering constraint (b) Ordering constraint

(c) A computation (d) A computation (e) A computation

Figure 3.7: Application of or constraint operator and valid computations

A computation containing C_1 and C_2 satisfies ϕ if event e_1 occurs before event e_2 *and* event e_3 occurs before event e_4 in the computation. A valid computation is shown in figure 3.6(c). ■

- Or Constraint Operator($\ovee$): The or constraint operator $\ovee$ is used to incorporate nondeterminism in the orderings of events. An event ordering constraint expression containing $\ovee$ is defined:

$$\phi = (\phi_1 \ovee \phi_2)$$

Intuitively, a computation satisfies event ordering constraint expression ϕ if it satisfies *at least one* of event ordering constraint expressions ϕ_1 or ϕ_2.

Example 3.4.2. *($\ovee$ operator).* Let events of program C_1 and C_2 be constrained by nondeterministic ordering relationships as shown in figure 3.7(a) and 3.7(b):

$$\phi = (e_1 \olessthan e_2) \ovee (e_3 \olessthan e_4)$$

There are three possible computations as shown in figure 3.7. In the first, event

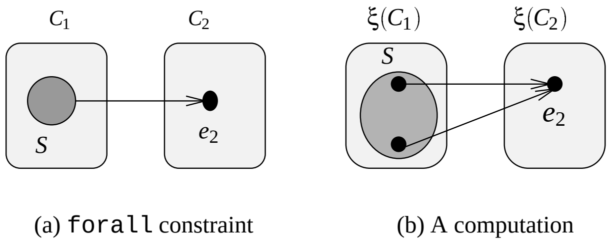


Figure 3.8: Application of `forall` constraint operator and a valid computation

e_1 occurs before event e_2 (figure 3.7(c)), whereas in the second e_3 occurs before e_4(figure 3.7(d)) . In figure 3.7(e), both relationships hold true. ■

- Forall operator: The and constraint operator $\textcircled{\wedge}$ is used for combining two event ordering constraint expressions. Forall extends $\textcircled{\wedge}$ in order to specify ordering constraints over sets of events. Let S be a set of events and $\phi(e)$ be an event ordering constraint expression over event e. The relationship between `forall` and $\textcircled{\wedge}$ is shown below:

$$\begin{array}{l} \texttt{forall var } e \texttt{ in } S\texttt{:} \\ \quad \phi(e) \end{array} \quad = \quad \underset{e\in S}{\textcircled{\wedge}} \phi(e)$$

Example 3.4.3. *(Forall operator).* Let events of program C_1 and C_2 be constrained by the expression (see figure 3.8(a)):

$$\boxed{\begin{array}{l} \texttt{forall var } e \texttt{ in } S\texttt{:} \\ \quad e \textcircled{<} e_2 \end{array}}$$

A valid computation is shown in figure 3.8(b). Here, all events of set S occur before event e_2. ■

- Exists operator: The `exists` operator is similar to `forall` in that it extends the or constraint operator over a set of events. Let S be a set of events and $\phi(e)$ be an event ordering constraint expression over event e. The relationship between `exists` and $\textcircled{\vee}$ is shown below:

```
exists var $e$ in $S$:
    $\phi(e)$
```

$$= \quad \bigcirc\!\!\!\!\vee_{e \in S} \phi(e)$$

3.4.4 Formal Semantics of Concurrent Program Composition

We now present the formal semantics of the C-YES model. We do so by defining computational structures that are created when a concurrent program is executed. A computational structure associated with a computation of a concurrent program identifies events in the computation, and the ordering relationships among the events. A computational structure can be represented in two ways:

- Interleaving model: In the interleaving model, a computation of a concurrent program is represented by an interleaved execution of the atomic actions of the components. Here, concurrency among events is represented as nondeterministic ordering relationships among the events. The interleaving model has been widely used for defining semantics of many concurrent programming languages [Hoa78, Mil80, Hen88]. It has also been used for proving safety and progress properties of programs [CM88, MP92].

- True concurrency model: In the true concurrency model, concurrency among events of a computation is represented by lack of execution orderings among the events.

We chose the true concurrency model for defining the semantics of the C-YES model because it is more natural for modeling entities that are hierarchical, relativistic, and non-atomic. Since our main interest here is in examining the computational structures of concurrent programs that result from constrained concurrent program composition and event ordering constraint expressions, the true concurrency model allows us to focus our attention on only those events that have causal relationships due to interaction.
Let a concurrent program C be defined:

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

```
<evoce>      ::=       <primEvoce>
                  |   <nonPrimEvoce>

<primEvoce> ::=      ( <eventId> ⓢ <eventId> )
<nonPrimEvoce> ::=  <evoce> ⓥ <evoce>
                  |   <evoce> ⓐ <evoce>
                  |   forall var <varIter> in <eventSet>
                                <evoce>
                  |   forall occ <varOcc> in <eventSet>
                                <evoce>
                  |   exists var <varIter> in <eventSet>
                                <evoce>
                  |   exists occ <varOcc> in <eventSet>
                                <evoce>
                  |   (<evoce>)
```

Figure 3.9: BNF expression for event ordering constraint expressions

The computational structure associated with a computation of program C is represented by a pomset (see Definition 3.2.7). It is specified by defining computational structures associated with computations of C_1 and C_2 in the computation of C, and the ordering relationships that exist among the events of the computational structures. A computation $\xi(C) = (V_C, <_C)$ of C is thus defined:

$$\begin{aligned} V_C &= V_{C_1} \cup V_{C_2} \\ <_C &= <_{C_1} \cup <_{C_2} \cup <_\phi \end{aligned}$$

In the above equation, terms $\xi(C_1) = (V_{C_1}, <_{C_1})$ and $\xi(C_2) = (V_{C_2}, <_{C_2})$ respectively represent computations of C_1 and C_2 in the computation $\xi(C)$. The set $<_\phi$ specifies the set of ordering relationships between events of $\xi(C_1)$ and $\xi(C_2)$. It represents interactions between the components, and is defined by the event ordering constraint expression ϕ. We now describe the manner in which $<_\phi$ can be evaluated from event ordering constraint expressions.

The BNF expression for specifying event ordering constraint expressions is shown in figure 3.9. Here, terms <eventId> and <eventSet> respectively denote events and event sets. Terms <`varIter`> and < `varOcc`> denote variables used for iterating over event sets. Term <`varOcc`> is different from <`varIter`> in that it ranges over occurrences numbers of events in an event set. Events can thus be represented by specifying integer expressions over <`varOcc`> variables. The term <`varIter`> on the other hand denotes events without making any references to their occurrence numbers. We develop the semantics associated with event ordering constraint expressions by defining a term, called the *Ordering Constraint Set*:

Definition 3.4.2. *(Ordering Constraint Set).* An ordering constraint set S is a set of sets of ordered pairs (e_1, e_2) such that e_1 and e_2 are events and

$$(e_1 \circledless e_2)$$

■

Notation: We use $S(\phi)$ to denote the ordering constraint set associated with the event ordering constraint expression ϕ.

We now specify the mechanisms for deriving ordering constraint sets, $S(\phi)$, from an event ordering constraint expression, ϕ.

- Primitive event ordering constraint expression: For expression

$$\texttt{<evoce>::= event\_id1} \circledless \texttt{event\_id2}$$
$$S(\texttt{<evoce>}) = \{\{\texttt{event\_id1, event\_id2}\}\}$$

- And constraint operator: For expression <`evoce`>`::=` $\phi_1 \circleddash\!\!\wedge\, \phi_2$

$$S(\phi_1 \circledwedge \phi_2) = \bigcup_{s_i \in S(\phi_1)} \left(\bigcup_{s_j \in S(\phi_2)} s_i \cup s_j \right)$$

- Or constraint operator: For expression <`evoce`>`::=` $\phi_1 \circledvee \phi_2$

$$S(\phi_1 \circledvee \phi_2) = S(\phi_1) \cup S(\phi_2)$$

- forall operator: For expression

```
ϕ = forall var e in S:
          ϕ1(e)
```

$$S(\phi) = S(\bigwedge_{e \in S} \phi_1(e))$$

- exists operator: For expression

```
ϕ = exists var e in S:
          ϕ1(e)
```

$$S(\phi) = S(\bigvee_{(e \in S)} \phi_1(e))$$

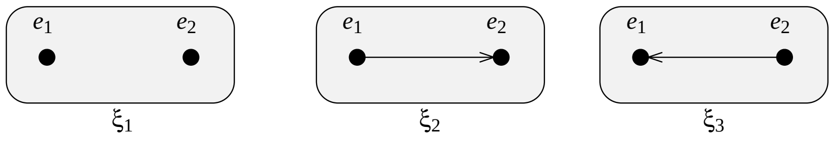
We now formalize the semantics associated with concurrent constraint composition expressions.

Definition 3.4.3. *(Sat).* For a computation $(V, <_v)$ and an event ordering constraint expression ϕ, the term $Sat((V, <_v), \phi)$ is true if

$$\begin{aligned} &\langle \exists\, s : s \in S(\phi) : \\ &\quad \langle \forall\, (e_i, e_j) : (e_i, e_j) \in s : \\ &\qquad (e_i \in V) \wedge (e_j \in V) \wedge (e_i <_v e_j) \rangle \rangle \end{aligned}$$

We say that $(V, <_v)$ satisfies ϕ. ■

Intuitively, a computation satisfies ϕ, if all orderings specified in at least one $s_j, s_j \in S(\phi)$, are preserved in the computation. Note that the notion of `Sat` is used merely to ensure that ordering constraints specified in ϕ are preserved in the computation. It does not ensure that computations do not contain additional orderings. For instance, figure 3.10 shows three computations ξ_1, ξ_2, and ξ_3, each of which satisfy the constraint that e_1 and e_2 are concurrent. In figure 3.10(a) the events are concurrent, whereas in figure 3.10(b) e_1 occurs before e_2. Similarly, e_2 occurs before e_1 in figure 3.10(c).



(a) Concurrent e_1 and e_2 (b) e_1 before e_2 (c) e_2 before e_1

Figure 3.10: Valid computations of two concurrent events

Definition 3.4.4. *(Constrained Concurrent Composition).* The constrained concurrent composition

$$(C_1 \parallel C_2 \parallel \ldots \parallel C_n) \text{ where } \phi$$

of programs $C_1, C_2, \ldots, C_n$ denotes a program C such that for every execution $\xi(C)$ of C,

$$\begin{array}{l}
\langle\, \forall\, i, j :\ i, j \in \{1 \ldots n\} \wedge i \neq j : \\
\quad \langle\, \forall\, e_k, e_l :\ e_k \in \xi(C_i) \wedge e_l \in \xi(C_j) : \\
\quad\quad (\neg e_k \parallel_e e_l) \Rightarrow \\
\quad\quad\quad \langle \exists\, s : s \in S(\phi) : \\
\quad\quad\quad\quad \langle \forall\, (e_i, e_j) : (e_i, e_j) \in s : \\
\quad\quad\quad\quad\quad (e_i \in V) \wedge (e_j \in V) \wedge (e_i <_v e_j) \\
\quad\quad\quad\quad \rangle \wedge ((e_k, e_l) \in s) \\
\quad\quad\quad \rangle \\
\quad \rangle \\
\rangle
\end{array}$$

Computations $\xi(C_i)$ and $\xi(C_j)$ respectively are the specific executions of program C_i and C_j that occur during the computation $\xi(C)$. ■

The constrained concurrent composition

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

of programs $C_1, C_2, \ldots, C_n$, therefore, denotes a program C such that

$$\langle\ \forall\ (V, <_v) : (V, <_v) \in X_s(C) : Sat((V, <_v), \phi)\ \rangle$$

Properties of Event Ordering Constraint Expressions

Theorem 1. *(Relationship between $\bigcirc\!\!\!\!\wedge$ and $\wedge$)* $Sat(V, \phi_1 \bigcirc\!\!\!\!\wedge\ \phi_2) = Sat(V, \phi_1) \wedge Sat(V, \phi_2)$
A proof for the theorem follows directly from the definitions of `Sat` and event ordering constraint set.

Theorem 2. *(Relationship between $\bigcirc\!\!\!\!\vee$ and $\vee$)* $Sat(V, \phi_1 \bigcirc\!\!\!\!\vee\ \phi_2) = Sat(V, \phi_1) \vee Sat(V, \phi_2)$
A proof for the theorem follows directly from the definitions of `Sat` and ordering constraint set.

The following define algebraic properties of the different operators:

$$p \bigcirc\!\!\!\!\wedge\ p = p \tag{3.1}$$

$$p \bigcirc\!\!\!\!\wedge\ q = q \bigcirc\!\!\!\!\wedge\ p \tag{3.2}$$

$$p \bigcirc\!\!\!\!\wedge\ (q \bigcirc\!\!\!\!\wedge\ r) = (p \bigcirc\!\!\!\!\wedge\ q) \bigcirc\!\!\!\!\wedge\ r \tag{3.3}$$

$$p \bigcirc\!\!\!\!\vee\ p = p \tag{3.4}$$

$$p \bigcirc\!\!\!\!\vee\ q = q \bigcirc\!\!\!\!\vee\ p \tag{3.5}$$

$$p \bigcirc\!\!\!\!\vee\ (q \bigcirc\!\!\!\!\vee\ r) = (p \bigcirc\!\!\!\!\vee\ q) \bigcirc\!\!\!\!\vee\ r \tag{3.6}$$

$$p \bigcirc\!\!\!\!\wedge\ (q \bigcirc\!\!\!\!\vee\ r) = (p \bigcirc\!\!\!\!\wedge\ q) \bigcirc\!\!\!\!\vee\ (p \bigcirc\!\!\!\!\wedge\ r) \tag{3.7}$$

$$p \bigcirc\!\!\!\!\vee\ (q \bigcirc\!\!\!\!\wedge\ r) = (p \bigcirc\!\!\!\!\vee\ q) \bigcirc\!\!\!\!\wedge\ (p \bigcirc\!\!\!\!\vee\ r) \tag{3.8}$$

In addition, the forall operator distributes over the $\bigcirc\!\!\!\!\wedge$ operator:

```
forall var  e in  S :  (ϕ1(e) ⩕ ϕ2(e)) =
        ( forall var  e in  S :  ϕ1(e)) ⩕
        ( forall var  e in  S :  ϕ2(e))
```

Similarly, `exists` distributes over $\vee$. All of the above properties can be proved by evaluating corresponding ordering constraint sets. In addition we have the notion of the T event ordering constraint expression:

$$S(T) = \{\{\}\}$$

All computations of a concurrent program constrained by the event ordering constraint expression T are valid computations.

3.5 Examples

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

3.5.1 Mutual Exclusion

Let S_1 and S_2 be sets of events. We want to specify that events of S_1 and S_2 are mutually exclusive. For events e_1 and e_2 such that $e_1 \in S_1$ and $e_2 \in S_2$, mutual exclusion between the two events can be represented as nondeterministic orderings of occurrences of e_1 and e_2:

```
MutexEvents($e_1$,$e_2$) = $(e_1 \olessthan e_2) \ovee (e_2 \olessthan e_1)$
```

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

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

3.5.2 Producer/Consumer

We construct a concurrent program that is composed from the `producer` and `consumer` components. The `produce` and `consume` actions of the components interact with each other. All other actions are noninteracting and hence may execute in parallel during the execution of the composite program. Let terms `produce` and `consume` denote the interaction points of `producer` and `consumer` respectively. Recall that an interaction point denotes a set of events. Term `produce[i]` therefore denotes the ith invocation of `produce` in a computation. The following expression defines a constrained concurrent program composition of the two components:

```
examprog1 = (producer ∥ consumer)
                   where
                       ConsExp
```

This expression specifies that during an execution of `examprog1`, events of `producer` and `consumer` occur in parallel except for those that must satisfy the ordering constraints specified by the event ordering constraint expression `ConsExp`. We now derive different event ordering constraint expressions for `ConsExp`.

The simplest interaction arises from the mutual exclusion constraint between the actions: no occurrences of `produce` and `consume` execute in parallel. We use the event ordering constraint expression of Section 3.5.1 to specify mutual exclusion among events of `produce` and `consume` events:

```
ConsExp = MutuallyExclusive(produce, consume)
```

Note that there are many possible executions of `examprog1` that satisfy the event ordering constraint expression `ConsExp`. The event dependency graph for one such execution is shown in figure 3.11. We have only shown the interacting events of the programs in the figure. Here all occurrences of `produce` dominate the first occurrence of `consume`, thereby causing the starvation of `consumer`. The event dependency graph in the figure is captured by the following event ordering constraint expression:

```
Starv =  forall occ  i in  produce :
                   produce[i] ⧀ consume[0]
```

Note

$$\langle \forall v :\ v \in \xi_s(\texttt{examprog1}) :\ Sat(v, \texttt{Starv}) \Rightarrow Sat(v, \texttt{ConsExp}) \rangle$$

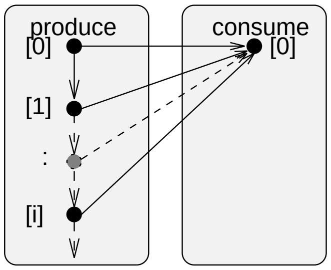


Figure 3.11: Domination of `produce` events over `consume` events

Another possible interaction constraint between events of `produce` and `consume` is that ith occurrence of `consume` cannot execute until the ith occurrence of `produce` has executed. The following primitive event ordering constraint expression represents the relationship between the two events:

$$\boxed{\texttt{produce[i]} \ \circledless\ \texttt{consume[i]}}$$

Since the above relationship holds true for all invocations of `produce`, event ordering constraint expression `ConsExp1` is defined in the following manner:

```
ConsExp1 =  forall occ  i in  produce :
                    produce[i] ⧀ consume[i]
```

A different interaction semantics for the `producer` and `consumer` programs can be specified by constructing a different event ordering constraint expression. For instance, assume that `producer` and `consumer` access a buffer of size one. The semantic constraints on accessing and modifying the buffer are: i) No data is consumed until it is produced, and ii) No data is produced until the previously produced data has been consumed. The following event ordering constraint expression captures these constraints:

```
SingleBuffer =     forall occ  i in  produce :
                        (produce[i] ⧀ consume[i]) ⦶
                   forall occ  i in  consume :
                        (consume[i] ⧀ produce[i+1])
```

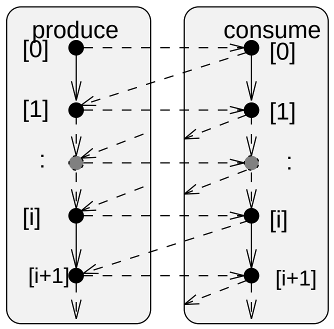


Figure 3.12: Single buffer interaction between `producer` and `consumer`

In figure 3.12, we show the event dependency graph associated with a computation of `examprog1` constrained by `SingleBuffer`. Note that there are no starvation or deadlocks.

3.5.3 Dining Philosopher

We now present a solution for the classical dining philosopher problem [Dij68a]. The problem occurs in different forms in many applications, especially in cases where multiple entities try to access a common resource. Also, it has often been used to illustrate the expressibility of an interaction specification mechanism.

Let $G = (V, E)$ be a graph such that nodes of the graph represent philosophers, whereas an edge represents a fork shared between two philosophers. The abstract behavior of a philosopher is:

```
while (TRUE) {
    think;
    get hungry;
    pick up both adjacent forks;
    eat;
    put down forks;
}
```

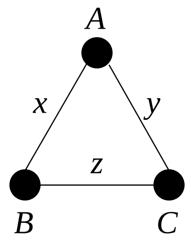


Figure 3.13: Configuration of dining philosophers and shared forks

There is a conflict between two philosophers when they try to access a common adjacent fork. Any solution to the dining philosopher problem must satisfy the following constraints:

- A philosopher can eat only if he has acquired both of his forks.
- All philosophers must be able to eat in finite amount of time.
- Philosophers think and eat for finite amount of time.

The solution in [CM84] satisfies these constraints by first converting the graph G into a directed graph. A directed edge (v_1, v_2) specifies that node v_1 contains the fork associated with edge (v_1, v_2). It then assigns priorities to the philosophers such that the directed graph is always acyclic. Also, the priorities are changed in a manner that the graph remains acyclic. The solution is both deadlock and starvation free. (See [CM88] for a proof.)

We represent this solution of the dining philosopher by event ordering constraint expressions. We assume that there are three philosophers, *A*, *B*, and *C*. The graph in figure 3.13 shows the forks that the philosophers share. The abstract behavior of each of the philosophers is shown in figure 3.14.

We derive the solution by examining the interaction between two philosophers, say X and Y (X and Y can be A, B, or C). The first part of the solution is to assign higher priority to node X over node Y in such a way that i) *i*th invocation of `get` and `put` of X over the shared fork has higher priority than ith invocation of `get` of Y. We can represent this by the following event ordering constraint expression:

```
A:
 while (TRUE) {
  think;
  hungry;
  x.get;
  y.get;
  eat;
  x.put;
  y.put;
 }
```

(a) Philosopher A

```
B:
 while (TRUE) {
  think;
  hungry;
  x.get;
  z.get;
  eat;
  x.put;
  z.put
 }
```

(b) Philosopher B

```
C:
 while (TRUE) {
  think;
  hungry;
  y.get;
  z.get;
  eat;
  y.put;
  z.put;
 }
```

(c) Philosopher C

Figure 3.14: Abstract specifications of three dining philosophers

$$\boxed{\begin{array}{l}\texttt{Prior1(X, Y, fork) = forall occ } i \texttt{ in X.fork.put :}\\ \qquad\qquad\qquad\texttt{X.fork.put[}i\texttt{]} \olessthan \texttt{Y.fork.get[}i\texttt{]}\end{array}}$$

In this expression, term `X.fork.put[i]` denotes philosopher X's ith invocation of `put` operation on fork. Once philosopher X has finished with the fork, the priority should now reside with Y. The following expression represents this priority:

$$\boxed{\begin{array}{l}\texttt{Prior2(X, Y, fork) = forall occ } i \texttt{ in Y.fork.put :}\\ \qquad\qquad\qquad\texttt{Y.fork.put[}i\texttt{]} \olessthan \texttt{X.fork.get[}i+1\texttt{]}\end{array}}$$

Since both of the above priorities must be satisfied, the following event ordering constraint expression captures the relationship between X and Y:

$$\boxed{\texttt{Priority(X, Y, fork) = Prior1(X, Y, fork)} \owedge \texttt{Prior2(X, Y, fork)}}$$

Interaction between A and B is defined by expression `Priority(A, B, x)`. Similarly interaction between A and C, and between B and C respectively are `Priority(A, C, y)`, and `Priority(B, C, z)`. The constraint over the philosophers therefore is

$$\boxed{\texttt{Priority(A, B, x)} \owedge \texttt{Priority(B, C, z)} \owedge \texttt{Priority(A, C, y)}}$$

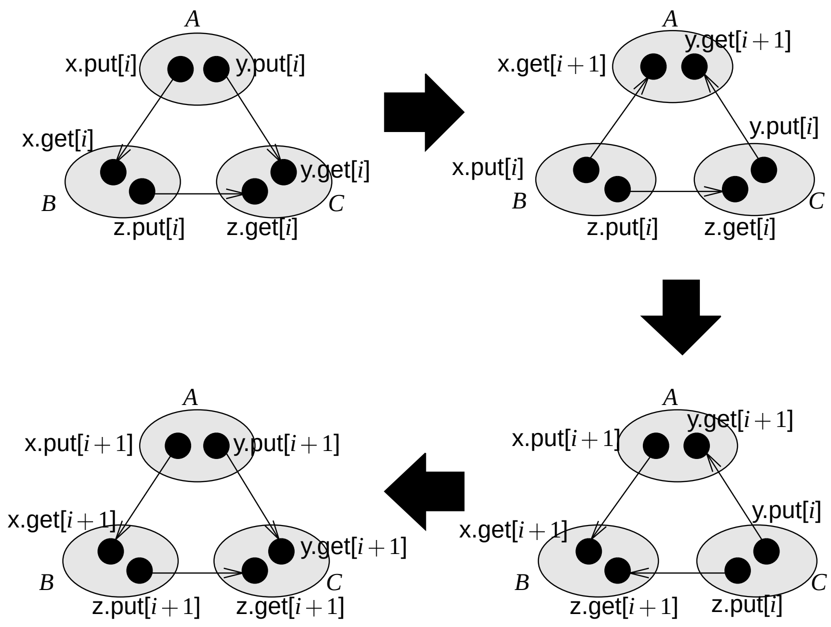


Figure 3.15: States of dining philosophers

Figure 3.15 shows a computation. Note that the priority assignment to philosophers is static in that the philosophers always eat alternately. The solution therefore does not guarantee maximum concurrency. For instance, in the case when it is *A*'s turn to eat, and if A thinks for a long time, both *B* and *C* must starve even if the forks are not being used.

3.5.4 Gaussian Elimination

We now develop a concurrent program for the Gaussian elimination algorithm. For a $n \times n$ matrix *A*, the Gaussian elimination algorithm [Ste73] is used to solve the linear equation

$$Ax = b$$

There are two steps in the algorithm: *forward elimination* and *back substitution*. In the forward elimination step, the matrix *A* is transformed into an upper triangular matrix, whereas in the back substitution step the transformed matrix is used to derive solutions of the unknowns. We focus our attention here on the forward elimination step only.

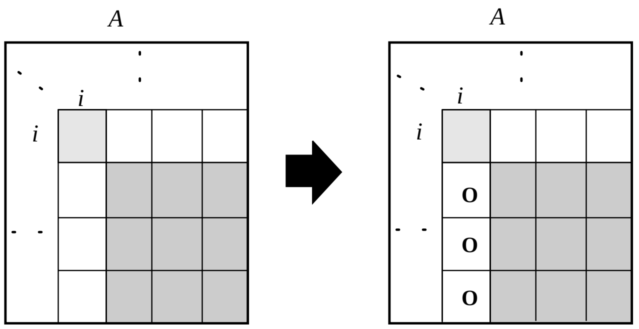


Figure 3.16: Transformation during a pivot step in the forward elimination step

There are $(n-1)$ steps, called *pivots*, in the forward elimination step. In the ith pivot step, elements $A[i+1,i]$ through $A[n,i]$ are reduced to zero, while modifying other elements of A as shown in figure 3.16. We represent each pivot step by an interacting program. The forward elimination program is a constrained concurrent program composition of the $(n-1)$ pivot programs. Interaction among the pivots is represented by an event ordering constraint expression.

The primary motivations for structuring the forward elimination program in this manner are to show that i) there is concurrency among the different pivot steps, and ii) this concurrency can be easily expressed by the composition mechanism of the C-YES model. The representation of the ith pivot step is shown in figure 3.17. This implementation of the pivot step is sequential.

We now derive the interaction points of `P(i)`: all read and write events over the elements of matrix `A` form its interaction points. Let the term `Access(i, j, k)` denote the set of all `P(i).A[j][k].read` invocations. The set contains all invocations of reads over the element `A[j][k]` in pivot `P(i)`. Similarly assume that the term `Modify(i, j, k)` denotes the set of all `P(i)`'s writes over the element `A[j][k]`. We derive an expression for the interaction among the pivots in terms of these sets of events.

We use a bottom up approach to specify the interaction among the pivots. We first derive an expression for the interaction among two events of the pivots `P(i)` and `P(i+1)`. We then extend this expression for all interacting events of `P(i)` and `P(i+1)`. Finally, we

```
P(int i, Matrix A)
{
   int j, k;
   for (j = i+1; j < n; j = j + 1)
      for (k = i+1; k < n; k = k+1)
         A[j][k] = A[i][k] - (A[j][k]*(A[i][i]/A[j][i]));
}
```

Figure 3.17: Representation of the ith pivot step

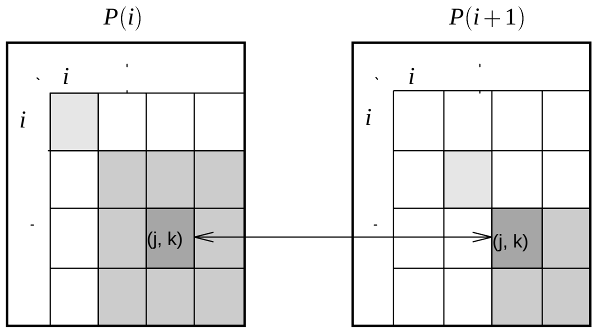


Figure 3.18: Interaction between reads and writes of pivots over a matrix element

define interaction for all pivots.

The first step is to examine the interaction between the ith and (i+1)th pivot steps (see figure 3.18): Pivot `P(i+1)` cannot read or write to any `A[j][k]` until `P(i)` has modified `A[j][k]`. There is an explicit ordering between the reads and writes of the two pivots and an event ordering constraint expression must represent this data dependency. The following event ordering constraint expression captures the relationship between events at `A[j, k]`:

```
ReadCons(i, j, k) = Change(i, j, k)[0] ⧀ Access(i+1, j, k)[0]
WriteCons(i, j, k) = Change(i, j, k)[0] ⧀ Change(i+1, j, k)[0]
```

The above constraints specify that `P(i+1)`'s first read and write events at `A[j][k]` must occur after `P(i)`'s first write event at `A[j][k]`. We extend the above relationship over all array elements that interact:

```
ForwardElimination(Matrix A)
{
   $\|_{i=1}^{n-1}$ P(i, A)
         where
            PivotInt
}
```

Figure 3.19: Specification of forward elimination step

```
IntExp(i) =
        forall var j in $\{(i+1),\ldots,n\}$:
            forall var k in $\{(i+1),\ldots,n\}$:
                ReadCons(i, j, k)
                            $\textcircled{\wedge}$
        forall var j in $\{(i+2),\ldots,n\}$:
            forall var k in $\{(i+2),\ldots,n\}$:
                WriteCons(i, j, k)
```

The above defines interaction between pivots P(i) and P(i+1). The next step is to extend the expression for all pivots:

```
PivotInt = forall var i in $\{1,\ldots,n-2\}$:
              IntExp(i)
```

The forward elimination program is shown in figure 3.19.

3.6 Discussion

We now discuss the different aspects of the C-YES model. We show that the C-YES model supports extensibility and modifiability of concurrent programs. In addition, the model supports reusability of both computational and interaction behavior specifications. In this section, we also highlight the modular and declarative nature of event ordering constraint expressions. We note that the interaction specification mechanism is general and is independent of the underlying memory model.

3.6.1 Concurrent Program Composition

The C-YES model is based on the concept of separation of concerns: specifications of computational behavior are separated from specifications of interaction behavior. We showed in Section 3.3 that such a separation allows one to easily extend and modify a concurrent program. We now show the manner in which the concurrent program composition and interaction specification mechanisms of the C-YES model can be used to support easy extensibility and modifiability of concurrent programs.

Extensibility of Concurrent Programs

In order to show that specifications of a concurrent program can be easily extended, we derive a concurrent program for `examprog2` of Section 2.4.1. Program `examprog2` is an extension of `examprog` in that it contains an additional `consumer` component. In the extended program, the two `consumer` programs share the information produced by the `producer` alternately. Program `examprog2` is defined in the following manner:

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

Note that there are no changes in specifications of either `producer` or `consumer`. The event ordering constraint expression `ConsExp2` represents the new interaction relationship among the three components. It is derived by observing that there are two sets of relationships among the events of the `producer` and `consumer` components (See figure 3.20). The first is between odd events of `produce` and events of one `consumer`, and the second is between even events of `produce` and events of the other `consumer`. Let `consume1` and `consume2` denote the interaction points of the two `consumer` components. The two relationships can be defined by the following event ordering constraint

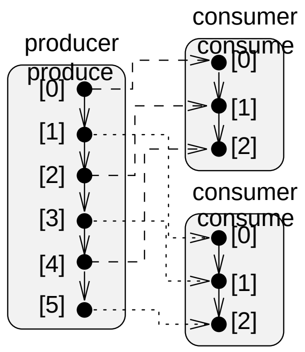


Figure 3.20: Interaction relationship among events in a computation of extended program

expressions:

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

Since the above relationship holds for all events of `produce`, `ConsExp2` is defined as:

```
ConsExp2 =  forall occ  i in  produce :
                (produce[2*i-1] <⃝ consume1[i])  ∧⃝
                (produce[2*i] <⃝ consume2[i])
```

A concurrent program can therefore be extended by adding new components, and by redefining interaction among the components. Note that in certain cases, redefinition of interaction may only involve adding new event ordering constraint expressions or modifying only a small subset of event ordering constraint expressions. An example of such a case is presented in Section 4.4.

Modifiability of Concurrent Programs

We now look at the modifiability of concurrent programs in the C-YES model. We present two examples: one is the representation of program `examprog3` of Section 2.4.1, and the other is redefinition of the forward elimination step of the Gaussian elimination algorithm (Section 3.5.4). In both cases, we construct different concurrent programs by composing

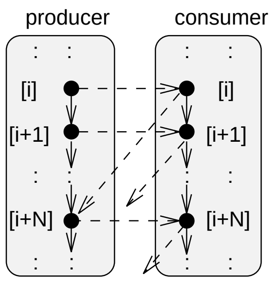


Figure 3.21: Interaction relationships among events in a computation of modified concurrent program

existing component programs with different event ordering constraint expressions. The examples highlight a program design methodology where concurrent programs can be constructed quickly from existing core components.

Modification of producer/consumer program: We first derive a representation for `examprog3`. In this program, there is additional constraint between the `producer` and `consumer` components: there can be at most N outstanding un-consumed values. We show a representation of `examprog3` below:

```
examprog3 = (producer ∥ consumer)
                 where
                     ConsExp3
```

In this program, there is relationship — in addition to the one defined by event ordering constraint expression `ConsExp1` of Section 3.5.2 — between `produce` and `consume` interaction points: a `produce` event cannot occur until a certain `consume` event has occurred (see figure 3.21). Event ordering constraint expression `ConsExp3` is thus defined:

```
ConsExp3 = ConsExp1 ⊙∧
                forall occ  i in  consume :
                   (consume[i] ⊙< produce[i+N])
```

Forward Elimination: We now define a different concurrent program for the forward elimination step of the Gaussian elimination algorithm. The focus here is to show that different concurrent programs can be constructed, each suitable for a different parallel machine, from existing components.

In the Gaussian elimination algorithm (Section 3.5.4), pivot `P(i)` waits for `P(i+1)` to modify the matrix element `A[j][k]` before it can read or write this element. The different pivot steps, therefore, interact at the level of single elements of the matrix. In many systems (such as distributed systems), such closely coupled interactions among programs can be very inefficient. We modify the `ForwardElimination` program in such a way that the pivots interact with each other only after they have modified a certain row. We construct the modified program by changing the event ordering constraint expression that defines interaction between the different pivots. The specifications of the pivots remain unchanged.

We derive an expression for the interaction between the pivots by observing that writes to the last element of a row by `P(i)` must occur before reads and writes to the first element of that row by `P(i+1)`. These ordering relationships can be represented by the following event ordering constraint expression:

```
PivotInt1 =
   forall var $i$ in {1, ..., (n-2)}
      forall var $j$ in $\{(i+1),\ldots,n\}$:
            Change(i, j, n)[0] $\circledless$ Access(i+1, j, i+1)[0]
                     $\circledwedge$
      forall var $j$ in $\{(i+2),\ldots,n\}$:
            Change(i, j, n)[0] $\circledless$ Change(i+1, j, i+1)[0]
```

The modified `ForwardElimination` is thus represented as:

```
ForwardElimination1(Matrix A)
{
      $\|_{i=1}^{n-2}$ P(i, A)
         where
            PivotInt1
}
```

Note that the composition relies on the fact that a row `j` is modified sequentially within `P(i)`. It exploits the ordering relationships between the reads and writes within a single row. The above composition expression, therefore, cannot be used if the rows are not modified sequentially in `P(i)`.

Encapsulation of Component Programs

Another implication of the separation of the two behaviors is that specifications of component are encapsulated: changes to the specifications of the components or the interaction behaviors do not affect the specifications of other components. It is, therefore, possible to change the implementation of a component without changing other components or their interaction behaviors as long as the nature of computation, interaction, and interaction points do not change. We show this by defining an alternate implementation of the pivot step. The change in the implementation of the pivot component does not require any changes in the specification of the interaction behavior or the `ForwardElimination` program.

The idea is to change those aspects of the program that do not affect either the interaction points or the semantic relationships among interaction points of the pivots. In `P(i)` it is easy to do because we can parallelize aspects of the pivot without changing either the interaction points (reads and writes to individual matrix elements) or the ordering constraints between them. Figure 3.22 shows such an implementation. The definitions of the `ForwardElimination` program (figure 3.19) and the `PivotInt` event ordering constraint expression (Section 3.5.4) remain unchanged, since there are no changes in either the interaction points or the orderings among interacting events. The program in figure 3.22 is optimally concurrent in that no event is delayed unless it is semantically dependent on some other event.

Reusability

In addition to the extensibility and modifiability of concurrent programs, the C-YES model supports reusability of both computational and interaction behavior specifications. Both

```
P(int i, Matrix A)
{
  int j, k;
    $\|_{j=i+1}^{j=n-1}$
      $\|_{k=i+1}^{k=n-1}$
          A[j][k] = A[i][k] - (A[j][k]*(A[i][i]/A[j][i]));
}
```

Figure 3.22: Parallel implementation of a pivot

computational and interaction behavior specifications can be used in defining different concurrent program compositions. For instance, we derived different concurrent programs from the `producer` and `consumer` programs.

3.6.2 Interaction Specification

We now analyze the interaction specification mechanism. We discuss its support for modular development of interaction specification, the abstractions it captures, and support for formal verification of certain classes of properties.

Modularity

Event ordering constraint expressions are declarative and compositional. The power of the interaction specification mechanism stems from the ability to decompose global interactions among programs into a set of local interactions. The local interactions can then be represented by event ordering constraint expressions, and combined with suitable interaction composition operators to represent the global interaction. One of the implications of the modularity property of event ordering constraint expressions is that it allows one to change interaction behavior of programs by changing only the relevant and local interaction expression. Also, it forms the basis for reusability of interaction behavior specifications.

We show the modularity property of event ordering constraint expressions by extending the producer consumer example (Section 3.5.2). Assume that `producer` and consumer programs access a buffer such that

```
produce = P(buffer[0]); P(buffer[1])
consume = C(buffer[0]); C(buffer[1])
```

Here, every produce event creates information in buffer[0] and buffer[1] and every consume event retrieves information from buffer[0] and buffer[1]. Note that produce and consume here are not atomic actions anymore; they are composed (with the sequential operator ';') from actions P and C respectively.

Expression SingleBuffer (Section 3.5.2) can be used to specify the interaction between the produce and consume actions. However, it over-constrains the executions of P and C events. It introduces unnecessary ordering among events when none is required. In order to derive an event ordering constraint expression that does not impose any ordering constraints among the events that occur at different buffers, we need to identify P and C as the basis of interaction — not produce and consume.

We derive an event ordering constraint expression by first specifying single buffer interactions at buffer[0] and buffer[1], and then by combining the two expressions to construct an expression for both the buffers. Let P1 and P2 respectively denote P events at buffer[0] and buffer[1]. Similarly, let C1 and C2 respectively denote C events at buffer[0] and buffer[1].

The derivation of the event ordering constraint expression for the interaction between the events are shown in figure 3.23. Note that

$$\langle \forall i : i \in \{ 1 \ldots \} : (\mathtt{P}, i+1) \parallel_e (\mathtt{C}, i) \rangle$$

Intuitively, it means that consumer can access buf[0] while producer is modifying buf[1]. Also, the above interaction expression can be extended to define interaction over a buffer of size N:

```
forall occ $i$ in P:
        P[$i$] $\olessthan$ C[$i$] $\owedge$ C[$i$] $\olessthan$ P[$i+N$]
```

$$\texttt{P1}[i] = \texttt{P}[2i-1] \tag{3.9}$$
$$\texttt{P2}[i] = \texttt{P}[2i] \tag{3.10}$$
$$\texttt{C1}[i] = \texttt{C}[2i-1] \tag{3.11}$$
$$\texttt{C2}[i] = \texttt{C}[2i] \tag{3.12}$$

```
$E_1$
= {interaction at buffer[0]}
  forall $i$ in {0 ...}
     P1[$i$] $\olessthan$ C1[$i$] $\owedge$
     C1[$i$] $\olessthan$ P1[$i+1$]

= { eqn 3.9, eqn 3.11 }
  forall $i$ in {0 ...}
     P[$2i-1$] $\olessthan$ C[$2i-1$] $\owedge$
     C[$2i-1$] $\olessthan$ P[$2i+1$]

= { $k=2i-1$ }
  forall $k$ in {1, 3, ...}
     P[$k$] $\olessthan$ C[$k$] $\owedge$
     C[$k$] $\olessthan$ P[$k+2$]
```

```
$E_2$
= {interaction at buffer[1]}
    forall $j$ in {0 ...}
        P2[$j$] $\olessthan$ C2[$j$] $\owedge$
        C2[$j$] $\olessthan$ P2[$j+1$]

= { eqn 3.10, eqn 3.12 }
    forall $j$ in {0 ...}
        P[$2j$] $\olessthan$ C[$2j$] $\owedge$
        C[$2j$] $\olessthan$ P[$2j+2$]

= { $l=2j$ }
    forall $l$ in {0, 2, ...}
        P[$l$] $\olessthan$ C[$l$] $\owedge$
        C[$l$] $\olessthan$ P[$l+2$]
```

```
{ Combine $E_1$ and $E_2$ }
ConsExp3 = $E_1 \owedge E_2$ =
            forall $i$ in {0 ...}
                P[$i$] $\olessthan$ C[$i$] $\owedge$
                C[$i$] $\olessthan$ P[$i+2$]
```

Figure 3.23: Modular derivation of representation of interaction

Abstractions of Interaction

The interaction specification mechanism captures fundamental abstractions of interaction. It specifies interaction by suitable ordering relations among the interacting events of programs. It does not depend on the semantics of these events. Hence, the model can be used for both shared and distributed memory programs.

Also, the mechanism is general in that it is not based on the semantic properties of any specific synchronization primitive. It can be used to specify any interaction behavior for any invocations of any action. The mechanism therefore can be used to represent many different synchronization mechanisms. In addition, the abstraction of interaction can be combined with other abstractions such as inheritance and genericity to construct powerful concurrent program abstractions. (See Section 5.6.)

Formal Verification

Event ordering constraint expressions are algebraic in nature. An event ordering constraint expression is constructed from a set of primitive event ordering constraint expressions and the interaction operators. Since the properties of the primitives and the operators are well defined, it is possible to verify certain safety and progress properties of the system from interaction behavior specifications in a rigorous manner. In addition, the verification process is facilitated by the separation of computational and interaction behaviors: many properties can be verified — in isolation from component programs — solely from the event ordering constraint expressions.

3.7 Summary

In this chapter we presented the conceptual foundation of a model of concurrent computation. We showed that concurrent programs are difficult to modify and extend because their components include specifications of both computational and interaction behaviors. Changes in a concurrent program (either by extension or modification) may induce changes in interaction behaviors of components. However, since specifications of components include specifications of both behaviors, changes in the interaction behaviors can be effected only by re-implementing the components. Concurrent programs can be easily modified and extended by separating the specifications of computational and interaction behaviors. Also, computational and interaction behavior specifications can be reused.

The C-YES model of computation is based on the above separation of concerns. It defines a concurrent program composition mechanism where specifications of computational and interaction behaviors of component programs are represented separately. In the model, the role of a concurrent program composition mechanism is to establish two kinds of relationships: *concurrency* and *interaction*. Concurrency represents semantic independence among the events. Interaction, on the other hand, represents semantic dependencies among operations of components. The semantic dependencies can be represented as a collection of ordering relationships. The C-YES model also contains an interaction specification mechanism that includes a primitive for representing individual ordering relationships as well as a set of operators for combining the relationships. The interaction specification mechanism supports modular development of interaction specification.

Chapter 4

A Compositional Approach to Concurrent Object-Oriented Programming

4.1 Introduction

In this chapter we present a compositional approach to concurrent object-oriented programming. The approach that we describe here forms an evolutionary middle point between the ideas presented in the previous chapter and a realization of these ideas in terms of a concurrent programming language described in the next chapter. We describe only the conceptual foundation of a concurrent object-oriented programming model here since it allows us to focus on the mechanisms for specifying explicit or implicit concurrent program compositions, and on the role of inheritance within the object-oriented model. The design of the programming language described in the next chapter is derived completely from the model described here.

In the previous chapter we described the C-YES model of concurrent computation. The approach includes a concurrent program composition mechanism that separates specifications of interaction behavior from specifications of computational behavior, an interacting model of component programs, and a declarative approach to interaction specification. We also showed that such an approach supports extensibility and modifiability of concurrent programs as well as reusability of computational and interaction behavior specifications. We use these components of the C-YES model to describe a compositional approach for concurrent object-oriented programming. The approach here is to identify the concurrent program compositions that exist, either implicitly or explicitly, within the concurrent object-oriented paradigm, and to represent them by the mechanisms specified of the C-YES model.

We chose to apply the C-YES model to the concurrent object-oriented model because concurrent objects provide a natural basis for modeling entities of applications. Such entities exist independently, and many allow multiple activities to occur in parallel. The notion of concurrency, both within entities and among different entities, exists naturally and can be modeled in the concurrent object-oriented model through inter-object and intra-object concurrency. In addition, there is a natural correspondence between the notions of actions and events, and the notions of methods and method invocations. Also, the concurrent object-oriented model supports a natural model for extensibility of abstractions through inheritance. They therefore provide a suitable ground where we can test the concepts of the C-YES model with respect to extensibility and modifiability of concurrent program.

We represent both intra-object and inter-object concurrency by the constrained concurrent program composition mechanism of the C-YES model. We model concurrency and interaction within a concurrent object in terms of a concurrent program composition. The concurrent program composition of a concurrent object is represented as separate specifications of computational and interaction behavior specifications of methods of the object. The interaction behavior of the methods is represented by event ordering constraint expressions. The semantics of the composition is that invocations of the methods execute in parallel except for those constrained by the event ordering constraint expressions.

We view concurrency and interaction among objects as a constrained concurrent program composition of invoking and invoked methods. Interaction among two methods represent the manner in which one method affects another (for instance, by returning certain result objects synchronously or asynchronously), and is represented by event ordering constraint expressions.

Within the framework of concurrent objects as concurrent program compositions of computational and interaction behaviors, inheritance can be defined as a mechanism for extending the concurrent program compositions of the concurrent objects. We show that a problem associated with inheritance of methods, called the *inheritance anomaly*, is an instance of the program composition anomaly in that a subclass extends a concurrent program abstraction (defined in a superclass) by defining additional program abstractions. However, the breakdown in the inheritance mechanism occurs because such compositions require that the program abstraction itself be modified. Separation of specifications of computational and interaction behavior of methods of concurrent objects allow us to avoid the inheritance anomaly. Also, it supports reusability of method and interaction behavior specifications.

This chapter is organized as follows: in Section 4.2 we present the concurrent program composition of concurrent objects. We describe the semantics associated with this composition. We also present the mechanisms that allow us to define interactions among methods in terms of event ordering constraint expressions. Interaction among objects is described in Section 4.3. In Section 4.4 we present inheritance as a mechanism for extending the concurrent program composition of a concurrent class by adding/modifying methods and their interaction behavior. We also look at the notion of the inheritance anomaly and present a resolution in terms of separation of concerns. In Section 4.5, we analyze the nature of interaction behaviors specified during class and method invocation definitions.

4.2 Intra-object Concurrency and Interaction

Several concurrent programming languages have used the concept of an encapsulated "object" as a basis for introducing concurrency. In some of these languages, objects are both units of executions and encapsulation. There are, thus, two views of concurrent objects: passive view and active view. In the passive view, an object encapsulates a data structure.

In the active view, on the other hand, objects represent a concurrent program whose execution behavior can be defined in terms of concurrently executing methods within objects. In this section, we will concentrate solely on the active view of concurrent objects — objects as concurrent programs.

Within the active view of concurrent objects, each object supports an execution environment where method invocations may occur 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. The behavior of the execution environment can either be defined explicitly by the programmer as in [GC86, Car93a] or can be implicitly determined from the interaction specification mechanism and from the semantics of concurrent objects. In this section we present a model of a concurrent object whose execution environment is derived *implicitly* from the specification of the object. Indeed, we use specifications of concurrent objects to generate an implementation for concurrent objects that supports concurrent executions of method invocations as well as synchronization among the invocations (see Chapter 6).

We represent concurrent objects by a *concurrent program composition* of two separate behaviors, namely computation and interaction behavior specifications. Formally, let the tuple $\langle M, \Phi \rangle$ represent the concurrent program composition of instances 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 $\phi_1, \phi_2, \ldots,$ and ϕ_l. Methods of set M are represented as interacting programs (see Section 3.4.2): they contain specifications of their computational behaviors and interaction points. Note that every invocation of a method denotes an event.

Notation: Let the term $m_i[j]$ denote the jth invocation of method m_i.

We specify the semantics associated with the concurrent program composition by defining a concurrent program. Let the term $P(O)$ denote the concurrent program associated with an object O of class C. Program $P(O)$ is derived from the specification of the composition:

$$P(O) = \left(\mathop{\Big\|}_{\substack{k \in \{1 \ldots \infty\} \\ m_j \in M}} m_j[k] \right) \ \texttt{where} \ \left(\mathop{\bigcirc\kern-0.75em\wedge}_{\phi_k \in \Phi} \phi_k \right) \tag{4.1}$$

The above expression specifies that all invocations of the methods of object O execute in *parallel*, except for those whose executions must satisfy *all* ordering constraints specified in the event ordering constraint expressions of set Φ.

Note that method invocations are concurrent by default. 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 concurrent program composition of concurrent objects can be viewed in terms of defining concurrency and interaction relationships among invocations of methods. In the absence of any knowledge about the application domain, concurrency is the fundamental relationship among the methods since it captures semantic independence among them. Interaction, on the other hand, represents semantic dependencies (such as data dependency, data consistency, and priority) among the methods. It is, therefore, application specific, and should be specified explicitly when the objects are specified.

In many languages, concurrent invocations of methods are always serialized and scheduled for executions according to the policies of the implementation. Such language-imposed serializations among methods define semantic dependencies among method invocations when there should be none. Programmers must specify concurrent programs with these dependencies in mind; programs otherwise may contain certain errors. For instance, two independent and concurrent programs may deadlock if one program invokes a method on an object B from a method of an object A, whereas another program invokes a method on object A from a method of object B. The programs deadlock because the language imposed orderings create cyclic dependencies among the two independent programs. In our approach, relationships are explicitly specified by event ordering constraint expressions, and hence can be examined, possibly formally, for the presence of certain kinds of errors.

In addition to the implementation mechanisms used for creating and scheduling different execution threads for method invocations, the behavior of the execution environment of a concurrent object is driven by the manner in which event ordering constraint expressions are evaluated. We evaluate an event ordering constraint expression by constructing a boolean *interaction constraint*, and by ensuring that the interaction constraint remains true during the life time of a concurrent object. Let $B(\phi)$ 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 interaction constraint is:

$$B(a \otimes b) \;=\; (a \mapsto b) \tag{4.2}$$

In the above, $(a \mapsto b)$ is true if event a occurs before event b. Operationally, the execution environment keeps the interaction constraint invariant by ensuring that event b is delayed until event a has terminated.

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

$$B(\phi_1 \owedge \phi_2) \;=\; (B(\phi_1) \wedge B(\phi_2)) \tag{4.3}$$

$$B(\phi_1 \ovee \phi_2) \;=\; (B(\phi_1) \vee B(\phi_2)) \tag{4.4}$$

$$B(\texttt{forall } a \texttt{ in } A\texttt{:}\phi) \;=\; \langle \forall a : a \in A : B(\phi) \rangle \tag{4.5}$$

$$B(\texttt{exists } a \texttt{ in } A\texttt{:}\phi) \;=\; \langle \exists a : a \in A : B(\phi) \rangle \tag{4.6}$$

The evaluation of event ordering constraint expressions that define relationships among infinite and dynamic event sets is done by evaluating the boolean interaction constraint incrementally, and by ensuring that the interaction constraint remains true over the lifetime of a concurrent object. The above transformations provide suitable mechanisms for generating an implementation for $P(O)$ from the specification of a concurrent class (see Chapter 6).

4.2.1 Interaction Behavior Specification

We now examine the nature of interaction behavior of methods within a concurrent object, and how they can be represented using event ordering constraint expressions. Interaction among methods arise when they access common resources and data structures. They represent semantic dependencies such as data dependency, data consistency, and priority among method invocations. Representations of the dependencies depend on a number of attributes [Blo79] associated with a concurrent object. We categorize them into two:

1. Naming mechanisms are needed for identifying invocations of methods, and for distinguishing among invocations of a method:

   - Type of a request: It is used to associate a method invocation with a specific method.
   - Relative order of invocation: Since method invocations may occur randomly, relative order of invocations distinguish among different invocations of a method.

2. There are attributes that form the semantic content of semantic dependencies. They determine if interaction relationships exist among method invocations. They are:

   - Request parameters: These denote parameters associated with a method invocation. Values of parameters are often used for specifying semantic dependencies among method invocations.
   - Synchronization state: The synchronization state of a concurrent object captures information such as methods that are waiting or running, or history of method invocations. Many semantic dependencies among method invocations depend on such information.
   - Local state: It denotes states of an entity that a concurrent object models.

Different interaction specification mechanisms differ in their representations of these attributes. Most languages have used different constructs for representing the different aspects. We surveyed some of these constructs in Section 2.3.2. Our approach to interaction behavior specification is to represent the different attributes through the notions of event sets and events. Interaction behaviors of methods can then be represented by event ordering constraint expression that are defined in terms of suitable event sets.

4.2.2 Event Set

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

Event sets allow representations of different kinds of interaction behavior of methods by a single unified mechanism. We identify two kinds of event sets. The first, called *static event sets*, are the event sets whose contents can be determined statically. An example is the event set that we can associate with a method. This set includes all invocations of the method. The second, called *dynamic event sets*, are the event sets whose contents change during program execution. Dynamic event sets associate certain states of concurrent objects with method invocations. They change due to the changes in the states. For instance, an event set that contains all invocations of a method that are currently waiting is dynamic in nature. Its content changes as the execution proceeds.

We enumerate a number of event sets below that we use for specifying interactions. Let *m* denote a method.

- *m*: This term denotes the set of all invocations of *m*.
- *m*`:waiting`: This term denotes the set of all invocations of *m* that are waiting to be executed.
- *m*`:running`: This term denotes the set of all invocations of *m* that are currently executing.

- *m*`:terminated`: This term denotes the set of all invocations of *m* that have terminated.

- *m*`:<B>`: This term denotes the set of all invocation of *m* for which the boolean condition <B> is true.

Terms *m*`:waiting`, *m*`:running`, and *m*`:terminated` are used to capture the synchronization states of a concurrent object. The term *m*:<B>, on the other hand, can be used to represent the application-specific state of the object in terms of a boolean condition so that interaction behavior of an invocation of *m* can be defined when the object is in that state. In addition, occurrence numbers of events allow us to capture relative order of invocations of methods. They provide syntactic mechanisms for representing specific method invocations.

4.2.3 Representation of Interaction Behaviors

We use event ordering constraint expressions for defining interaction behavior of methods. We make one simple extension to the mechanism described in Section 3.4.3. The primitive event ordering constraint expression is extended to include the notion of conditional ordering. It has the form:

$$p \Rightarrow (e_1 \otimes e_2)$$

The above expression specifies that event e_1 must occur before event e_2 if the condition p is true. However, if p is false, the ordering relationship is not enforced, and the events can occur in parallel. The above expression is more general. It captures representations of certain relationships more directly.

4.2.4 Examples

We now present a number of examples that illustrate the manner in which concurrent objects and interactions among methods can be specified:

Example 4.2.1. *(Specification of concurrent objects).* Let the tuple $\langle M, \Phi \rangle$ define a concurrent class `queue`. Here, $M = \{$ `put, get` $\}$. Methods `put` and `get` access `queue` objects, and interact while accessing common data structures. Interaction behavior of the methods is specified by the following constraints: i) `put` invocations are sequential, ii) `get` invocations are sequential, iii) `put` events are delayed if the `queue` is full, and iv) `get` events are delayed if the `queue` is empty.

We represent the constraints by event ordering constraint expressions. Therefore,

```
Φ = { SeqPut, SeqGet, DelPut, DelGet}
```

We first derive event ordering constraint expressions for `SeqPut` and `SeqGet`. We define an expression `Serialize` that orders events of an event set S according to their occurrence numbers:

```
Serialize(S) =
      forall var  i in  S :
          forall var  j in  S :
              $(i < j) \Rightarrow (S[i] \oslash S[j])$
```

Event ordering constraint expressions for `SeqPut` and `SeqGet` can be derived by instantiating the `Serialize` expression over event sets `put` and `get`:

```
SeqPut = Serialize(put)
SeqGet = Serialize(get)
```

We now derive event ordering constraint expressions for `DelPut` and `DelGet`. We do that by defining conditional relationships between events of two sets. The relationships hold as long as a boolean condition remain true. Event ordering constraint expression `WaitWhile` below defines such a relationship:

```
WaitWhile($S_1$, $S_2$, BooleanCond) =
        forall var  $e_1$ in  $S_1$:BooleanCond :
            forall var  $e_2$ in  $S_2$ :
                $(e_2 \oslash e_1)$
```

In the above, the set S_1`:BooleanCond` contains all events of S_1 for which the boolean expression `BooleanCond` is true. Hence, as long as the condition `BooleanCond` is true, events of S_1 are delayed with respect to events of S_2. Expressions `DelPut` and `DelGet` therefore can be defined as:

```
DelPut = WaitWhile(put, get, full)
DelGet = WaitWhile(get, put, empty)
```

`full` and `empty` are boolean conditions that respectively determine if a `queue` object is full or empty. In the above, `DelPut` specifies that all events of set `put:full` are delayed with respect to events of set `get`. Here set `put:full` captures all `put` invocations for which a `queue` object in `full` state. Such events are delayed with respect to `get` events because it is the `get` events that change the `full` state of the `queue` object. Expression `DelGet` is similarly defined.

Interaction behavior of invocations of `get` and `put` events is specified by the following interaction expression:

```
SeqPut ⓥ SeqGet ⓥ DelPut ⓥ DelGet
```

Note that there are no serialization assumptions regarding invocations of methods. Behaviors of method invocations that may affect other method invocations are represented by a single mechanism: event ordering constraint expressions. ■

Example 4.2.2. *(Simple priority).* Let `read` and `write` be two methods of an object. Interaction behavior of the methods is specified by the following constraints: i) `read` and `write` events are mutually exclusive, ii) `write` events are mutually exclusive, and iii) waiting `read` events have higher priority that waiting `write` events. Note that the above constraints permit concurrent executions of `read` methods. The three semantic constraints are represented by three separate event ordering constraint expressions:

```
    1.  MutuallyExclusive(read, write)
    2.  MutuallyExclusive(write, write)
    3.  Priority(read:waiting, write:waiting)
```

Expression `MutuallyExclusive` is defined in example 3.5.1. We define an expression `Priority` that simply orders events of set S_1 over events of set S_2:

```
Priority($S_1$, $S_2$) =
        forall var  $e_1$ in  $m_1$ :
            forall var  $e_1$ in  $S_2$ :
                   $(e_1 \olessthan e_2)$
```

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

```
MutuallyExclusive(read, write) $\owedge$
MutuallyExclusive(write, write) $\owedge$
Priority(read:waiting, write:waiting)
```

If 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:

```
Priority(write:waiting, read:waiting)
```

The above illustrates the support for modular development of interaction behavior in the interaction specification mechanism. 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 event ordering constraint expression. For instance, we could modify interaction behavior of `read` and `write` events by changing only the relevant priority expressions. Expressions that represent other relationships remained unchanged. ■

4.2.5 Representation of Synchronization Primitives

We now discuss the generality of the interaction specification mechanism by showing that many synchronization primitives can be represented in terms of specific event sets and relationships between events of these sets.

Representation of Enable Command

Enable-based approaches to interaction specification are derived from the concept of conditional critical regions [Hoa72] in that a guard, called the *activation condition*, is associated with the methods of a class. Guards determine interaction behavior of a method in that an invocation of a method m_1 can execute if its activation condition, say p, is true. However, if p is false, the invocation is delayed until p becomes true. The form for an enable command is:

$$\boxed{\texttt{enable}\, m_1\, \texttt{when}\, p}$$

Examples of languages that use enable-based approaches for specifying synchronization are Capsule [Geh93], Scheduling Predicates [MWBD91], Synchronization Actions [Neu91], Guide [DKM$^+$89], PLOOC [Tho92], and Mediators [GC86]. The different approaches can be differentiated on the basis of the nature of activation conditions. However, we will treat the activation conditions as abstract entities and will not distinguish among different approaches on the basis of the nature of the activation conditions.

A representation of the enable command in terms of an event ordering constraint expression is specified by examining the interaction relationship between an event, e_1, of method m_1 and an event, say e_2, of event set $M - m_1$. Here set M contains all events of all methods of an object. If condition p is false, e_1 is delayed with respect to e_2. However, if p is true, there is no ordering relationships between the events; they can execute in parallel. The following event ordering constraint expression represent the enable command:

```
enable($p$, $m_1$) =
        forall var $e_1$ in $m_1 : (\neg p)$ :
            forall var $e_2$ in $M - m_1$ :
                $(e_2 \lessdot e_1)$
```

The above expression allows us to compare the conceptual differences in our interaction specification mechanism and other approaches, and to examine the possible implications on implementation costs of enable commands and event ordering constraint expressions. Our approach to interaction specification mechanism allows specification of semantic relationships among any event sets. In most other approaches to interaction specification, relationships are specified among specific event sets. For instance, in the enable command relationship is specified between events of set $m_1 : (\neg p)$ and events of $M - m_1$. This restricts the possible kinds of interaction relationships that can be defined.

Also, in most implementations of the `enable` command, condition p is evaluated every time an event of $M - m_1$ terminates. This is to ensure that a blocked m_1 event is scheduled for execution if p is true. However, in many cases such evaluations may be unnecessary because some events of $M - m_1$ may not affect p. (Conceptually, these events do not interact with events of m_1.) However, in our approach interactions can be specified only among those event sets that interact, thereby avoiding unnecessary evaluations of the activation conditions.

Representation of Disable Command

Disable-based approaches [Fro92, SG91] are similar to enable-based approaches in that they associate guards with the methods of a class. However, the disable-based approaches interpret the guard differently in that a method is disabled (as opposed to enabled) if the guard is true. We use the term *deactivation condition* for guards. The form of disable commands is:

```
disable $m_1$ when $p$
```

The semantics of a disable command in terms of an event ordering constraint expression is specified by examining the relationship between an occurrence, e_1, of method m_1 and an occurrence, say e_2, of event set $M - m_1$. If p is true, e_1 is delayed with respect to e_2. However, if p is false, there are no relationships among the two events; they can execute in parallel. The following event ordering constraint expression represents a disable command:

```
disable($p$, $m_1$) =
        forall var $e_1$ in $m_1 : p$ :
            forall var $e_2$ in $M - m_1$ :
                $(e_2 \oslash e_1)$
```

Representation of Behavior Abstraction

In behavior abstraction-based approaches [KL89, TS89b, Mat93], interaction behavior of methods is characterized by identifying a set of disjoint states, called *accept states*. Every accept state determines the set of methods that can be enabled when an object is in a state. (Many synchronization specification mechanisms [MTY93] include a number of commands for specifying transition among states. We will ignore them here in order to simplify our analysis.) The form for defining an accept state is:

$$A_j = \{m_1, m_2, \ldots, m_n\} \ \texttt{when}\ B_j$$

In the above, A_j denotes an accept state associated with boolean condition B_j. Methods $m_1, m_2, \ldots, m_n$ are enabled when condition B_j is true.

We represent the above command by observing that the accept state denotes a specific event set:

$$A_j = m_1 : (\neg B_j) + m_2 : (\neg B_j) + \cdots + m_n : (\neg B_j)$$

Also, the ordering relationship is defined between events of set A_j and set $M - A_j$:

```
forall var $e_1$ in $A_j$ :
    forall var $e_2$ in $M - A_j$ :
        $(e_2 \oslash e_1)$
```

Let a class defines accept states $A_1, A_2, \ldots,$ and A_k with boolean conditions $B_1, B_2, \ldots,$ and B_k respectively. Let ϕ_j represent the event ordering constraint expression associated with accept state A_j. The event ordering constraint expression associated with the methods of the class is:

$$\phi_1 \circledvee \phi_2 \circledvee \cdots \circledvee \phi_k$$

We therefore see that many of the synchronization primitives used in concurrent object-oriented programming languages can be represented in terms of event sets and relationship between events of the sets. We show representations of other commonly used synchronization primitives in Section 5.6.

4.2.6 Aggregation Anomaly

We now look at the notion of aggregation in object-oriented programming languages. Aggregation is used to define data composition of an object in terms of its component objects. Within the active view of objects, we can look at aggregation as a program composition mechanism as well: it defines the concurrent program associated with a concurrent object as a composition of programs associated with its component objects and methods of the concurrent object. A problem arises when we try to modify the program composition of an component object by extending/modifying the interaction behavior of the object. We call this problem *aggregation anomaly*. We describe the problem by the following examples:

Example 4.2.3. *(Aggregation anomaly).* Assume that a `Buffer` object contains an object of a class `AtomicBuffer`. Also, assume that class `AtomicBuffer` defines two methods: `Read` and `Write`, which are mutually exclusive. Let class `Buffer` impose the following constraint on `Read` and `Write` invocations on the `AtomicBuffer` object component: if certain conditions are true, `Read` and `Write` invocations must occur in a certain order. Since specifications of `Read` and `Write` already include specifications of interaction behavior, class `Buffer` can be defined in two ways: one is by constructing a new class that implements the constraint that `Buffer` imposes, and by including an object of this class as the component object. Here, we are duplicating most of the code from `AtomicBuffer` class. The other is by defining methods in `Buffer` that invoke `Read` and `Write` with suitable synchronization operations. In this case one needs to define dummy methods. ■

Another situation where the aggregation anomaly occurs is the case when interaction is specified among methods of different component objects:

Example 4.2.4. *(Aggregation anomaly).* Assume that an object of class `TwoBuffers` contains objects `buf1` and `buf2` of class `AtomicBuffer`. Interaction between `Read` and `Write` on objects `buf1` and `buf2` is specified by the constraint that `Read` and `Write` on `buf1` have higher priority than the corresponding methods on `buf2`. The constraint cannot be specified easily. One possible approach would be to define methods in `TwoBuffers` that implement the constraint and invoke `Read` and `Write` methods on `buf1` and `buf2` appropriately. Note that this would require that all invocations of `Read` and `Write` on `buf1` and `buf2` be replaced by invocations of the new methods. For objects that include many objects, each containing a number of methods, the process of constructing methods that implement interaction constraints can be quite cumbersome. ■

The aggregation anomaly occurs because interaction among methods of component objects cannot be changed or extended easily. One must either redefine the component objects or define additional methods that implement the extended/modified interaction behavior. The aggregation anomaly is an instance of the program composition anomaly in that we are extending/modifying the concurrent program abstraction associated with a component. However, such a composition requires that either the abstraction be changed or the composition be implemented through indirect means. A resolution of the aggregation anomaly requires that computational and interaction behavior specifications of methods be separated. This permits easy extension and modification of interaction behavior of methods. In addition, languages must support mechanisms for changing interaction behavior of method invocations of component objects by providing mechanism for identifying the method invocations.

4.3 Inter-object Concurrency and Interaction

The notion of independence among objects represents concurrency among the concurrent programs associated with the 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

$$\| \, O_2.m_2(p_1, p_2, \ldots p_n) \texttt{ where } \phi \tag{4.7}$$

specifies the mechanism for invoking method m_2 on an object O_2 in parallel. The event ordering constraint expression ϕ represents interaction between the calling and the called methods. Expression 4.7 derives its form and semantics from the constrained concurrent program composition expressions (see Definition 3.4.1). Assume that a method m_1 includes expression 4.7 in its specification. An execution of the invocation expression during an occurrence of m_1, say $m_1[j]$, maps[1] 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 specified in ϕ. The event ordering constraint expression ϕ is defined over the interaction points of m_1 and m_2. A more general form of the above method invocation expression is:

$$\| \, O_1.m_1 \, \| \, O_2.m_2 \, \| \ldots \| \, O_n.m_n \texttt{ where } \phi \tag{4.8}$$

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

The above invocation mechanism is general in that it subsumes the traditional synchronous and future-based asynchronous method invocation mechanisms. For instance, in the future-based method invocations, interaction occurs between a read (in the calling method) event and a write (in the called method) event on a future object. It can be represented by the following event ordering constraint expression:

$$\phi = \texttt{(}m_2\texttt{.var.write[0]} \otimes m_1\texttt{.var.read[0])}$$

[1]The mapping can be done either by creating a new thread of execution within O_2 or by scheduling an existing thread. It depends on the technique used to implement concurrent threads within objects.

Here, `var` is a future object, and `var.write` and `var.read` respectively denote sets of write and read events over `var`.

The event ordering constraint expressions in method invocation expressions 4.7 and 4.8 can be used to represent relationships among method invocations on objects that methods share through the parameters. The expressions may represent complex protocols among calling and called methods. Also, certain properties of the protocols can be verified by analyzing the expressions formally.

4.4 Inheritance

In sequential object-oriented programming languages, inheritance [WZ88] provides a powerful mechanism for organizing classes in a generalization-specialization hierarchy. In this hierarchy, classes at the top capture more general information than the ones below. Such an organization of classes provides the ability to incrementally extend superclasses by inheriting information such as conceptual behavior, data structures, and/or implementation mechanisms, and by modifying or adding to the inherited behavior. In this section, we examine the notion of inheritance as a mechanism for extending program composition of a concurrent class by adding and/or modifying methods and their interaction behaviors.

4.4.1 Inheritance Anomaly

In many concurrent object-oriented programming languages there is a problem with the inheritance of method implementations. This problem, termed the *inheritance anomaly* [MY93], arises due to the differences in synchronization requirements of a class and its subclasses. We clarify the problem through the following example:

Example 4.4.1. *(Inheritance anomaly).* Let a concurrent class C define two methods m_1 and m_2. Implementations of m_1 and m_2 may contain, in addition to specifications of computations, operations used to define their interaction behavior. Let S be a subclass of C. It inherits definitions of m_1 and m_2. Assume that class S defines a new method m_3. The method m_3 interacts with m_1 and m_2. The interaction behaviors of m_1 and m_2, as specified in C, therefore, have changed in S. Methods m_1 and m_2 may have to be re-implemented in S in order to represent the modified interaction behaviors. Implementations of m_1 and m_2, thus, cannot be inherited in S. ■

The inheritance anomaly is an instance of the program composition anomaly. Here, we can think of the definition of class C as defining a concurrent program abstraction. Inheritance defines a mechanism whereby the concurrent program abstraction can be composed with other abstractions to create more complex concurrent program abstractions. However, the breakdown in inheritance occurs because such compositions require that the original program abstraction itself be modified. This requires that the methods of C be re-implemented in class S.

The reasons for the occurrence of the inheritance anomaly are similar to those of the program composition anomaly: computational behavior of inherited methods remains unchanged in the subclass; only their interaction behavior changes due to changes in the concurrent program composition of a superclass. For instance, computational behaviors of m_1 and m_2 do not change in S; only their interaction behavior changes due to the addition of m_3. *The inheritance anomaly arises because specifications of methods contain specifications of both computational and interaction behaviors*. Any changes in the interaction behavior may, therefore, require changes in the implementation as well.

A resolution for the inheritance anomaly can be derived from the principles used for resolving the program composition anomaly. There are two components: the first is separation of specifications of computational and interaction behaviors of methods. Separation makes it possible to inherit the two behaviors separately, and to modify either to reflect changes in the concurrent program composition of a concurrent class. The second is the ability to make changes in the interaction behaviors of methods. The inheritance anomaly has been studied in great detail and many solutions [KL89, TS89b, BF92, Shi89, RdP91, Neu91, Tho92, Mes93] have been proposed. Most of these solutions are based on the separation of synchronization constraints from the method specifications. Changes in interaction behavior of a method is achieved by changing the relevant synchronization constraints. The *inheritance anomaly* does not arise in our model because the concurrent program composition of concurrent objects is specified in terms of separate computational and interaction behavior specifications. In addition, we provide mechanisms for changing interaction behaviors specifications of methods. We describe them below.

4.4.2 Model of Inheritance

Interaction behaviors of methods add an extra dimension to concurrent class specifications. In this section, we examine what it means to extend this additional component through extension and modification. We also present a model of inheritance which specifies the manner in which interaction behavior of a method can be extended in subclasses.

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

$$\phi = \phi_s \bigcirc\!\!\!\!\wedge \phi_c \tag{4.9}$$

We now look at support for inheritance of interaction specifications Our focus is on minimizing changes in interaction specifications due to changes in the concurrent program composition of a concurrent class when complete reusability cannot be achieved. Complete reusability can be achieved if i) additional semantic dependencies due to new methods can be incrementally added, ii) modifications in semantic dependencies due to 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.

For a class $C = \langle M, \Phi \rangle$, an 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 event ordering constraint expression. Changes in interaction behavior due to changes in the concurrent program 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 the possible interaction behavior changes that can occur.

Let a subclass S extend class $C = \langle M_c, \Phi_c \rangle$ by defining the tuple $\langle \Delta M_c, \Delta\Phi_c \rangle$. Here, ΔM_c and $\Delta\Phi_c$ denote sets of methods and event ordering constraint expressions respectively. Let ΔM_n and ΔM_m be two components of ΔM_c:

$$\Delta M_c = \Delta M_n \cup \Delta M_m$$

Set ΔM_n is a set of methods that are added in class S. Methods of set 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 set $\Delta\Phi_c$, and the manner in which they arise:

$$\Delta\Phi_c = \Delta\Phi_n \cup \Delta\Phi_m$$

Set $\Delta\Phi_n$ contains event ordering constraint 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 event ordering constraint 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. Also, 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.

Example 4.4.2. *(Extension of interaction behavior of methods).* Consider the example of a class `queue` which defines two operations `put` and `get`. A general class may specify that `put` and `get` events are mutually exclusive. Different subclasses can be defined that extend the nondeterministic constraint by specifying additional 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. event ordering constraint expressions here characterize interaction groups containing methods of set ΔM_n.

3. Interactions among the methods of ΔM_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, and how they can be represented by event ordering constraint expressions:

Example 4.4.3. *(Inheritance of event ordering constraint expressions).* Let class `readlastqueue` be a subclass of `queue` (see example 4.2.1). The subclass adds a method `getlast`. Method `getlast` retrieves the last element of the queue. It interacts with method `put`, since it must wait for a `put` event to occur if the queue is empty. Interaction behavior of the methods of `readlastqueue` is extended by defining event ordering constraint expressions that define ordering relationships among `getlast` and `put` events:

```
DelayGetLast = WaitWhile(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 expression:

```
DelayPutWithGetlast = WaitWhile(put, getlast, Full)
```

Other event ordering constraint expressions in `ReadFirstQueue` are inherited from the `queue` superclass. The interaction behavior of the methods of `ReadFirstQueue` is specified by the expression:

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

■

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.

Event ordering constraint 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 i) among the methods of ΔM_c, representing modified interactions among the modified methods, and ii) among the methods of ΔM_c and $M_c - \Delta M_c$, representing additional semantic dependencies among the modified and inherited methods of S.

The set of event ordering constraint expressions, $I(S)$, of S therefore is:

$$I(S) = \{\, \phi_i \mid (\phi_i \in \Delta\Phi_c) \lor ((\phi_i \notin \Delta\Phi_c) \land (\phi_i \in I(Superclass(S))))\}$$

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

$$\phi_s = \left(\bigcirc\!\!\!\!\wedge_{\phi_i \in \Delta\Phi_n} \phi_i\right) \bigcirc\!\!\!\!\wedge \left(\bigcirc\!\!\!\!\wedge_{\phi_i \in \Delta\Phi_m} \phi_i\right) \bigcirc\!\!\!\!\wedge \left(\bigcirc\!\!\!\!\wedge_{\phi_i \in (\Phi_c - \Delta\Phi_m)} \phi_i\right) \tag{4.10}$$

The set of methods, $M(S)$, of S is:

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

The program $P(O_s)$ associated with an object O_s of S is:

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

4.5 Incremental Specification 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 4.7 and 4.8). Our goal in this section is to examine the nature of interaction behavior specifications in the two definitions, and to establish relationships between them. We discuss the implications of these specifications on the reusability of concurrent classes and interaction specifications, and examine their representations by event ordering constraint expressions.

We motivate our discussion through an example. Consider two concurrent programs that interact by sending and receiving messages over a communication channel. There are two semantic constraints: i) every `receive` event must wait for a corresponding `send` event, and ii) there are at most *N* outstanding messages. There are three ways in which the constraints can be represented in an application. The first approach is to define a concurrent class `appChannel` that defines both constraints along with the `send` and `receive` methods. In the second approach, one defines a concurrent class `conChannel` that does not define any of the semantic constraints. The constraints are specified where the `send` and `receive` methods are invoked. In the third approach, the first semantic constraint is defined in a concurrent class, say `channel`, whereas the second constraint is specified during method invocations.

We now examine the three approaches with respect to reusability of concurrent classes and interaction behavior specifications. The first approach advocates constructing concurrent classes that precisely implement requirements of an application. The problem with this approach is that the class definitions are too specialized. They may not be reused in applications that do not impose similar semantic constraints. Applications must re-implement these classes in order to specify diverse semantic constraints. The second approach advocates defining classes that do not specify any semantic constraints. All semantic constraints are represented during method invocations. Class `conChannel` is too general since applications that require asynchronous communication implement the specification of the first constraint repeatedly. The third approach is based on the decomposition of the interaction behavior. Here, the definition of the `channel` class captures only those semantic 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 semantic constraint — every `receive` event must wait for corresponding `send` event — is valid for all invocations of `send` and `receive`. On the other hand, the semantic constraint — `send` events must be delayed if specific `receive` events have not occurred — is valid for specific invocations of `send` and `receive` in an application. Different applications may specify different specialized interaction behaviors for invocations of `send` and `receive`.

Universal interaction behavior represents a single set of semantic constraints that guide execution behaviors of all occurrences of a method. Since a class captures and represents semantic constraints that govern access to its objects, universal interaction behavior of methods is defined where the methods are defined. Examples of such semantic constraints are data consistency, fairness, mutual exclusion, and priority. Note that universal interaction behavior of methods should depend only on intrinsic properties of objects and the methods. This allows one to maintain the notion of independence and encapsulation that classes exhibit — the ability to define classes in isolation from other aspects of applications. Specialized interaction behavior of method invocations, on the other hand, depends on the environment in which the invocations occur. It is, therefore, possible that different occurrences of a method exhibit different specialized interaction behaviors, each extending the universal interaction behavior of the method.

Our programming model supports this 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. *Interaction behavior of a method invocation is $\bigcirc\!\!\!\!\wedge$ composition of the two — universal and specialized — interaction behaviors*. Hence, if event ordering constraint expression ϕ_u represents universal interaction behavior of methods M_1 and M_2, and if interaction expression ϕ_s represents specialized interaction behavior of events $M_1[p]$ and $M_2[q]$, interaction behavior of $M_1[p]$ and $M_2[q]$ is represented by the event ordering constraint expression:

$$\phi_u \bigcirc\!\!\!\!\wedge \phi_s$$

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 invocations impose additional constraints on the possible orderings among the method invocations.

An example of a class that defines universal interaction behavior is `SafeBuf`, whose methods, `put` and `get`, are mutually exclusive. The event ordering constraint expression `MutuallyExclusive(put, get)` (see Section 3.5.1) represents their universal interaction behavior. 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 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 4.5.1. *(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 an application imposes a constraint on `put` and `get` events of `m1` and `m2`: every `put` event in `m1` occurs before the corresponding `get` event in `m2`. The constraint represents specialized interaction behaviors of `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

$$\boxed{\begin{array}{ll}\phi_1 = & \texttt{forall var } k \texttt{ in m1.X.put:} \\ & \qquad \texttt{m1.X.put[}k\texttt{]} \circledless \texttt{m2.X.get[}k\texttt{]}\end{array}}$$

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 the 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`. ■

Interaction behavior of a method invocation is, thus, $\circledwedge$ composition of its universal and specialized interaction behaviors. Such a separation of the interaction behaviors facilitates reusability of both concurrent class and interaction behavior specifications.

4.6 Summary

In this chapter we have presented an active view of concurrent objects where an object is represented as a concurrent program. In this view, a concurrent object is defined by specifying a concurrent program composition of a set of methods and a set of event ordering constraint expressions. The semantics of the composition specifies that invocations of different methods start to execute in parallel with respect to each other. However, in cases when method invocations interact, execution orderings of the invocations must satisfy ordering constraints specified in the event ordering constraint expressions.

We have shown that separation of computational and interaction behavior specifications allows one to extend/modify the concurrent program composition of a concurrent object easily. We considered two cases (the aggregation anomaly and the inheritance anomaly) when changes (extensions or modifications) in the program composition of a concurrent object result a breakdown in the aggregation and inheritance composition mechanisms of the concurrent object-oriented model. We have also shown that the two anomalies are instances of the program composition anomaly and they occur because specifications of methods include specifications of both computational and interaction behaviors.

In addition, we have presented a model of inheritance for concurrent object-oriented programming languages. Within this model, a subclass extends interaction behavior of methods by defining additional semantic relationships between the methods. Since additions of semantic relationships are captured by the $\owedge$ operator in the interaction specification mechanism, interaction behavior of methods in a class is $\owedge$ composition of ordering relationships specified in superclasses and in the class.

Chapter 5

CYES-C++

5.1 Introduction

In this chapter we describe the design of a concurrent object-oriented programming language, called CYES-C++. The conceptual and semantic foundations for CYES-C++ are based on the ideas developed in Chapters 3 and 4. We focus our attention here mainly on the design principles. The details of the language appear in its specification document [Pan].

Design Approach

One approach to language design is to start from the first principle and design a completely new language. This approach, though intellectually fulfilling, requires major effort since it involves complete development of both the specification of the language and the software development tools such as compilers and debuggers. Also, users must learn a new language and a set of tools. We therefore decided to choose an existing object-oriented programming language and extend it. We base our language design on the C++ programming language [ES90]. The reason for choosing C++ from among many object-oriented languages such as SmallTalk is the wide acceptability of C++. In addition, we were also motivated by the availability of many C++ tools. Indeed, we have been able to reuse many existing C++ and C libraries and tools for constructing a prototype implementation for CYES-C++. (See

Chapter 6 for more details on implementation.)

There are three ways in which concurrency can be included in C++. The first two approaches, which we call the *implicit approach*, require no changes in the base language and its compiler. One of them is based on defining a set of libraries [SS87, Gau] that support representations of concurrency and synchronization. In these approaches, a class, say `Task`, implements creations of independent execution threads. A user class can construct independent execution threads by inheriting from the `Task` class. The second approach [Par90] involves using the interface definition of a method of a class as an RPC stub specification. A stub generator can use the definition for constructing an RPC implementation for the method. The generated RPC implementation can then be used for invoking the method on an object that resides on a remote processor. In the third approach, which we call the *explicit approach*, additional constructs are added to the C++ programming languages. Examples of languages based on the explicit approach are: CC++ [CK92], Mentat [Gri93], Charm++ [KK93], Molecule [Geh93], COOL [CGH92], pC++ [GL91], Concurrent C++ [GR88], DC++ [Car93b], μC++ [BS93], PANDA [ABB$^+$93], ESP [SC90], and ACT++ [KL91]. Languages based on the explicit approach require that a preprocessor or a compiler be developed for the languages.

Languages based on the implicit approach have the advantage that no changes need to be made in either the base language or the software tools. The library approach has the added advantage that different libraries can explore different models of concurrency. Library based approaches are especially appealing in C++ since C++ supports many language mechanisms for defining and reusing libraries. Further, since C++ is still evolving, languages based on the explicit approach are based on a language that may change in future. This means that preprocessors or compilers for these languages must be re-implemented. This is especially difficult due to the complexity involved in constructing a parser for C++.

Despite the arguments against explicit approaches, we chose the explicit approach for designing the concurrent programming language. The reasons are based on the fundamental nature of concurrency and its role in the programming process. In the library approach, representations for concurrency and interaction are derived implicitly from class libraries. They can therefore be included in programs only through the sequential composition mechanisms of the C++ programming languages. This involves embedding concurrency and synchronization classes and methods within sequential class definitions and method invocations. As we showed earlier, such composition definitions are difficult to extend and modify. Also, library-based approaches represent concurrency and interactions as specific types. We, on the other hand, view concurrency and interaction as composition mechanisms, mechanisms that specify relationships among method invocations. Such a view cannot be easily added through the library mechanism since the composition mechanisms (mostly aggregation and inheritance) associated with libraries do not directly support representations of such views. Also, library-based approaches support a passive view of concurrent objects. Here objects are merely units of encapsulation. Concurrency is achieved through task objects which do not represent a natural model of active entities of applications.

Design principles

The primary goals of our design are to support true concurrency both within and among concurrent objects, allow modular development of interaction specifications, support reusability of both method and interaction specifications, and integrate inheritance with concurrency. In addition, we want to keep the number of extensions small. Further, we wanted to keep the essential spirit of C++ by supporting similar definition, declaration, and expression specification mechanisms.

We achieve these goals by providing mechanisms for representing abstractions that are orthogonal to the abstraction mechanisms of sequential C++. One such abstraction is the notion of concurrent classes. Concurrent classes support true concurrency within objects. They are generalizations of C++ classes. (In a sequential class, all method invocations are serialized.) Therefore, many of the C++ language constructs (for instance, the definition and declaration mechanisms and the notion of special member functions such as constructors and destructor) apply to concurrent objects as well. The method invocation mechanisms of CYES-C++ also are generalizations of the C++ method invocation mechanism.

One major addition in CYES-C++ is support for interaction specification through event ordering constraint expressions. CYES-C++ supports language mechanisms for defining events, event sets, primitive event ordering constraint expressions interaction composition operators, and event ordering constraint expressions.

The design approach for CYES-C++ is on supporting orthogonal abstraction and composition mechanisms. A direct outcome of the approach is that the abstractions of CYES-C++ can be combined with other abstractions and composition mechanisms (such as inheritance and the template mechanism) of C++ to create powerful concurrent programs abstractions. For instance, the template and inheritance mechanism of C++ can be used to create generic concurrent classes. The generic concurrent classes capture common interaction and computational behaviors. They can be instantiated or extended with different user classes in order to create concurrent user classes with suitable computational and interaction behaviors. Our abstraction mechanisms therefore complement the abstraction mechanisms of C++ to create a powerful concurrent programming language.

This chapter is organized as follows: In Section 5.2 we define the concurrent class mechanism of CYES-C++. Section 5.3 describes the synchronous and asynchronous method invocation mechanisms. We present the interaction specification mechanism in Section 5.4. Section 5.5 contains a number of examples that describe the manner in which inheritance and the template mechanism can be combined with the notion of concurrent classes. Finally, Section 5.7 contains a summary of the chapter.

5.2 Concurrent Objects

In this section we describe the syntactic mechanisms that can used for defining, declaring and accessing concurrent objects.

```
concurrent class queue {
public:
    queue();
    ~queue() { };
    void put(char);
    char get();
    Boolean Full();
    Boolean Empty();
interaction:
    SequentialAdds;
    SequentialRemoves;
    SyncQueueFull;
    SyncQueueEmpty;
private:
             ⋮
};
```

Figure 5.1: Concurrent class specification of concurrent `queue` objects

5.2.1 Definition of Concurrent Objects

Concurrent objects in CYES-C++ are represented by defining a concurrent class. An interface of a concurrent class contains, in addition to `public`, `private`, and `protected` entities of C++ classes, `interaction` section. An interaction section of a concurrent class contains definitions of event sets and event ordering constraint expressions, which can be used to represent interaction among the public methods of the concurrent class.

In figure 5.1, we show the concurrent class specification for a class `queue` (example 4.2.1). There are four constraints on the methods of `queue`, each of which is represented symbolically in the concurrent class definition. Section 5.4.3 contains definitions of these constraints. The semantics associated with a concurrent object is that methods execute in parallel by default. However, their executions must satisfy all ordering constraints specified by the event ordering constraint expressions specified in the `interaction` section.

5.2.2 Declaration of Concurrent Objects

Concurrent objects are declared by the `declarator` mechanism of C++. The declarator mechanism contains a number of naming schemes for sequential objects. Below we show how they apply for declarations of concurrent objects:

- Simple concurrent object declaration: Here, a name is associated with an object by defining it to be of a concurrent class type. For instance, the declaration

  ```
  queue q;
  ```

  associates a name `q` with a concurrent `queue` object.

- Pointers to concurrent objects: Pointer declarations provide an indirect way of naming concurrent objects. The specification

  ```
  queue *qptr;
  ```

  declares `qptr` to be a pointer to a concurrent queue object.

- Array of concurrent objects: Concurrent objects can also be named through the array mechanism of C++. For instance, the declaration

  ```
  queue qarray[100];
  ```

  specifies that `qarray` is an array of 100 concurrent objects. Expression `qarray[i]` names the (i+1)th `queue` object.

- Reference: Another mechanism for naming objects is through the reference operator:

  ```
  queue &qname;
  ```

  In this declaration, `qname` is a reference to a concurrent `queue` object.

Definitions and declarations of concurrent objects are therefore similar to C++ objects. In addition, expressions used for accessing sequential objects can be used for accessing concurrent objects as well. For instance, an execution of the expression

```
qptr->put(val);
```

invokes method `put` synchronously on a concurrent object named through the `qptr` object.

There is one place where names of concurrent objects take a different meaning from those of sequential objects. It occurs during method invocations. In C++, function parameters can be passed by value. Execution of a function therefore involves copying the actual parameters and passing them to the called function. (C++ also supports call by reference through references.) We do not support the call-by-value semantics for concurrent object parameters because the call-by-value semantics involves creating concurrent objects every time concurrent objects are passed as parameters. Creation of concurrent objects is computationally expensive, thereby making method invocations computationally inefficient. Concurrent objects therefore are always passed by reference. This involves copying only the name description of a concurrent object in the invoked method. For instance, a method invocation of the form

```
queue q;
   ⋮
obj.func(q);
```

involves copying the name of the concurrent object `q` and passing it to `func`. This has the implication that all method invocations on the parameter object within `func` are made on the object named by `q`.

Restrictions: The following restrictions are imposed on the definitions and declarations of concurrent objects:

- Concurrent classes cannot contain static members. The reason is that static variables are shared among instances of a class. Since each instance of a concurrent class represents a concurrent program, static variables act as shared variables among the instances. It is difficult to support accesses to such variables on distributed systems.

- Instance variables of concurrent objects cannot be accessed directly. They must always be declared private. They can be accessed and modified through the public methods.

- Currently there is no support for global concurrent objects. However, we do plan to remove this restriction in the future.

- A sequential object cannot include concurrent objects as instance variables.

5.2.3 Special Member Functions of Concurrent Classes

In C++, there are a number of special member functions associated with a class. These functions are either be specified explicitly in the class, or given a default implementation by the C++ compiler. In this section, we discuss these functions within the context of concurrent objects.

Constructor: A constructor is used for initializing instance variables of an object, and for allocating any resources that the object may require. It is invoked when the declaration of the object is within the scope of execution or when the `new` operator is invoked to create the object. Constructors for concurrent objects have identical behavior. For instance, during the execution of the expression

```
qptr = new queue;
```

the constructor of the `queue` class is invoked. Currently there are no special requirements for constructors of concurrent objects. They are therefore mostly used for initializing instance variables of concurrent objects. We do, however, plan to provide mechanisms that can be used for allocating processor and other resources for concurrent objects.

Destructor: A destructor is used for freeing up the resources associated with an object. It is invoked when the declaration of the object goes out of the scope or when the `delete` operator is invoked. Destructors for concurrent objects are invoked in a similar fashion. However, their semantics is different from that of C++ destructors. The difference arises due to the fact that a concurrent object can be accessed through multiple names. In CYES-C++, we distinguish between a primary name and a secondary name of a concurrent object: the primary name of a concurrent object is the one used for creating the object. Secondary names for the concurrent object are created when the object is passed as a parameter during method invocations. In CYES-C++, a concurrent object is deleted only when its primary name go out of scope or is deleted by the `delete` operator. The destructor of a concurrent object applies the destructor on the instance variables of the object, and deletes any processor and synchronization resources allocated to the object.

Copy constructor: A copy constructor is invoked during parameter passing and during the execution of an assignment expression. Since CYES-C++ does not support the call-by-value semantics for parameter passing, copy constructors for concurrent objects are invoked only during the execution of an assignment expression. The following describes the semantics associated with a default copy constructor of a concurrent object:

- Apply copy constructors on instance variables.
- Copy resource specifications.
- Create a separate execution thread for left hand side object.

Note that the synchronization states (such as the states of different event sets) of concurrent objects are not copied.

Initialization and assignment operators: Initialization and assignment operators for concurrent object derive their semantics from the constructors of concurrent objects.

5.3 Method Invocation

In Section 4.3 we showed that inter-object concurrency and interaction can be represented by method invocation mechanisms. In this section we describe the syntactic mechanisms for specifying inter-object concurrency and interaction. Note that the method invocation mechanisms are primary mechanisms for constructing different kinds of concurrent program structures in CYES-C++. There is another view of method invocations, specifically the notion of remote procedure calls in distributed programming languages. CYES-C++ supports this view indirectly in that there are no assumptions in the language regarding the underlying machine architecture. A CYES-C++ program can therefore execute on both shared and distributed machines[1]. On distributed systems, method invocations on a concurrent object that resides on a different processors are equivalent to remote procedure calls. CYES-C++ supports both synchronous and asynchronous method invocations on concurrent objects.

5.3.1 Synchronous Method Invocation

In synchronous method invocation, an invoking method is blocked until the invoked method terminates. The syntax for invoking a method synchronously is identical to the one used for invoking methods in C++. For instance, during an execution of the expression

```
q.put(val);
```

the invoking method is blocked until `put` terminates. Similarly, execution of the expression

```
val = q.get();
```

blocks the invoking method until `get` terminates and returns a value.

5.3.2 Asynchronous Method Invocation

A method `func` can be invoked asynchronously on a concurrent object `obj` by an expression of the form:

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

[1]An implementation of the language currently runs on a network of IBM RS6000 workstations.

The term `evoce` is an event ordering constraint expression. It represents interaction among the interaction points of the calling method and an occurrence of `func`. A simple instance of the asynchronous method invocation is:

```
par q.put(val)
```

Here method `put` occurs in parallel with the invoking method. CYES-C++ also supports the ability to invoke many methods in parallel through the `parfor` operator:

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

The above operator is similar to `parfor` of CC++ [CK92]. However, invocations of methods are asynchronous in CYES-C++. The above `parfor` expression terminates once all methods have been invoked. This is unlike the `parfor` operator of CC++ where a `parfor` expression terminates only after all invoked methods have terminated. In certain applications the CC++ `parfor` operator is more suitable since they support a default synchronization point for the invoked methods. We are looking at ways of introducing the operator in CYES-C++.

5.4 Interaction Specification

Interaction in CYES-C++ is represented by event ordering constraint expressions. In this section, we describe the syntactic mechanisms used for representing event sets, events, and event ordering constraint expressions. Their detailed semantics can be found in Chapters 3 and 4.

5.4.1 Event Sets

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

Primitive Event Sets

For every method `M` of a concurrent object, the following primitive event sets are supported:

- The term `M` denotes the set of all invocations of method `M`.
- The term `M:waiting` denotes the set of all invocations of `M` currently waiting.
- The term `M:running` denotes the set of all invocations of `M` currently executing.
- The term `M:terminated` denotes the set of all terminated invocations of `M`. .

Nonprimitive Event Sets

Nonprimitive sets are constructed from other event sets through the mechanisms describes below:

Parameters of Methods: Event sets can be constructed on the basis of values of parameters of method invocations. For instance, the expression `add(2)` denotes an event set containing all invocations of method `add` with the parameter value 2. Such event sets are useful in representing interactions among method invocations that can be distinguished by values of their parameters. Currently we support only integer parameters.

Conditional Event Sets: Conditional event sets are used to capture states of concurrent objects and to associate them with the events of the sets. They are represented by a term of the form `M:B`, where `M` is an event set and `B` is a boolean condition. The term denotes the set of all events of `M` for which `B` is true. For instance, the term `get:empty()` denotes all invocations of `get` for which the condition `empty()` is true.

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

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

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

Event Set Expressions: Event sets can be combined with other event sets with the following set operators:

1. Union (+)

2. Difference (−)

Hence, an expression of the form

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

extends the event set `fullqueue` to include the events of set `putlast:full()`. The named event set and the set operators are useful when extending and modifying the interaction behavior of methods of superclasses in a subclass. Named event sets are also used for constructing generic concurrent classes (see Section 5.6).

Interaction Points: Interaction points of methods are denoted by the `NamedSelector` mechanism. It is used to select event sets from a set of events. Hence, in the method invocation expression

```
par obj1.m1(p1), obj2.m2(p2)
        where evoce
```

the term `obj1.m1` denotes a set of events. The events in this set are the method invocations on its parameter`p1`, and on local and global objects. The term `obj1.m1:p1.op1` denotes all `op1` method invocations on object `p1` in an invocation of `m1` on `obj1`.

5.4.2 Events

Events are represented by selecting specific events from an event set. We support two mechanisms for denoting events:

1. Occurrence Number: Events can be selected through their occurrence numbers. The occurrence number of an event in an event set denotes its invocation order. (Note that an event can have different occurrence numbers in different event sets.) The term `S[exp]` is used to denote an event in the event set `S`. The integer expression `exp` determines its occurrence number in `S`. For instance, term `put:full()[2]` denotes the third occurrence of an event in set `put:full()`.

2. Event variable: Operators `forall` and `exists` use event variables to iterate over an event set.

5.4.3 Event Ordering Constraint Expressions

Interaction among methods of concurrent objects is represented by event ordering constraint expressions. In this section, we briefly describe the syntactic mechanisms used for specifying event ordering constraint expressions. The detailed semantics can be found in Chapters 3 and 4.

Interactions can be defined by instantiating a named event ordering constraint expression with suitable parameters or by explicitly specifying an event ordering constraint expression. A named event ordering constraint expression is defined in the following manner:

```
evocename(p1, p2, ..., pn) {
    evocexp
}
```

The above definition associates the event ordering constraint expression `evocexp` with `evocename`. Terms `p1`, `p2`, ... and `pn` are parameters. The expression `evocexp` may contain references to the parameters. Named event ordering constraint expressions allow one to define expressions that can be reused by instantiating them with different parameters. Also, named event ordering constraint expressions can be used for changing interaction behaviors of methods of superclasses in subclasses.

An event ordering constraint expression is constructed from a set of *primitive ordering constraint expressions* and a set of *interaction composition operators*. The syntactic representations of the primitive expression and the operators in CYES-C++ are shown in table 5.1.

Example 5.4.1. *(Interaction specification).* We present an example that illustrates the manner in which the concepts of event sets, events, and event ordering constraint expressions can be used to specify interaction. In this example, we derive the event ordering constraint

Primitive expression	`B => (e1 < e2)`
And constraint operator	`evoce1 && evoce2`
Or constraint operator	`evoce1 || evoce2`
forall constraint operator	`forall var v in S { evoce }`
	`forall occ i in S { evoce }`
exists constraint operator	`exists var v in S { evoce }`
	`exists occ i in S { evoce }`

Table 5.1: Syntactical representation of event ordering constraint expressions

expressions that were symbolically specified in the interaction section of class `queue` (figure 5.1). Note that example 4.2.1 also contains specifications of the interaction among the methods of `queue`. However, we use a different approach here: the interactions are specified in terms of named event sets. This approach has the advantage that specifications of the interactions can be extended easily by modifying the definitions of the named event sets in a subclass.

We first define two named event ordering constraint expressions. Terms `S`, `S1`, and `S2` in the expressions below denote events sets.

```
Serialize(S) {
   forall occ i in S {
      (S[i] < S[i+1])
} }
```

```
WaitWhile(S1, S2) {
   forall var a in S1 {
      forall var b in S2 {
         (a < b)
} } }
```

Below We define four named event sets that capture different aspects of a `queue` object and associate them with specific events:

```
QueueEmpty = get:empty()
Queuefull = put:full()
AddToQueue = put
RemoveFromQueue = get
```

Set `QueueEmpty` contains all invocations of `get` for which a `queue` object is in empty state. Similarly, set `QueueFull` contains all invocations of `put` for which a `queue` object is in full state. Set `AddToQueue` contains all events that add information to the `queue`. Set `RemoveFromQueue` contains all events that remove information from the `queue`. We now instantiate the named event ordering constraint expressions with suitable named event sets:

```
SyncQueueEmpty     = WaitWhile(AddToQueue, QueueEmpty)
SyncQueueFull      = WaitWhile(RemoveFromQueue, QueueFull)
SequentialAdds     = Serialize(AddToQueue)
SequentialRemoves  = Serialize(RemoveFromQueue)
```

■

5.5 Inheritance

In Section 4.4 we presented a model of inheritance in which inheritance is a composition mechanism for extending the concurrent program composition of a concurrent class through additions and modifications of methods and their interaction behaviors. In this model, interaction behavior of methods in a class is $\textcircled{\wedge}$-composition of event ordering constraint expressions specified in the class and the superclasses. The inheritance model of CYES-C++ is derived from this model. In this section we present a number of examples that illustrate the ways in which concurrent classes can be extended.

Extension of Interaction Behavior: The first example shows the manner in which interaction behaviors of methods of a class can be extended when the class is extended by adding a method.

Example 5.5.1. *(Method addition).* Let `readlastqueue` be a subclass of class `queue`:

```
concurrent class readlastqueue:  public queue {
public:
   char getlast();
interaction:
      ⋮
}
```

Method `getlast` retrieves the last element of a `queue` object. It interacts with method `put` of class `queue`: invocations of `getlast` must wait for invocations of `put` if the `queue` is empty. Similarly invocations of `put` must wait for invocations of `getlast` if the `queue` is full.

The interactions between the added method and the inherited methods can be specified easily by adding invocations of `getlast` in the named event sets of `queue`:

```
RemoveFromQueue    = RemoveFromQueue + getlast
QueueEmpty         = QueueEmpty + getlast:empty
```

Both computational and interaction behavior specifications of the methods of class `queue` are inherited in class `readlastqueue`. In addition, the event ordering constraint expression that were specified in class `queue` over the named event sets apply to the invocations of `getlast` as well:

- The expression `SyncQueueEmpty` specifies that events of both `get` and `getlast` are delayed with respect to events of `AddToQueue` if the queue is empty.

- The expression `SyncQueueFull` specifies that the events of set `QueueFull` are delayed with respects to the events of `RemoveFromQueue` (containing events of both `get` and `getlast`).

- Since the events of `RemoveFromQueue` are serialized (by the event ordering constraint expression `SequentialRemoves`), all invocations of `getlast` are serialized as well. ■

State Partitioning: In the next example, we show the definition of a concurrent subclass that partitions a specific state of its superclass.

Example 5.5.2. *(Concurrent object state partitioning).* Let `queueone` be a subclass of `queue`:

```
concurrent class queueone:  public queue {
public:
   void gettwo(char val[]);
interaction:
   SyncQueueOne
      ⋮
}
```

Method `gettwo` accesses two elements of the `queue` object atomically. Invocations of `gettwo` are delayed with respect to `put` if the buffer is empty or has one element. Note that a `queue` object can be in one of the three states: full, empty, or partially filled. The addition of method `gettwo` partitions the partially filled state into two: `queue` with one item, and `queue` with more than one item. State partitions can be represented through definitions of new event sets. Let method `one()` return true if a `queueone` object contains one item. We first define the following event sets:

```
QueueOneObject = gettwo:one()
```

The event ordering constraint expression

```
SyncQueueOne = WaitWhile(AddToQueue, QueueOneObject)
```

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

```
EmptyQueue = EmptyQueue + gettwo:empty()
RemoveFromQueue = RemoveFromQueue + gettwo
```

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

Multiple Inheritance: We now present an example where a concurrent class is constructed by composing two concurrent classes. The goal here is to show that interaction behaviors defined in different concurrent classes can be combined and extended easily through the notion of event sets.

Example 5.5.3. *(Composition of two concurrent classes).* Below we show the interface of a concurrent class `LockObj` :

```
concurrent class LockObj {
public:
   lock();
   unlock();
   int locked();
interaction:
   LockEventSet = lock:locked()
   UnlockEventSet = unlock
   LockObjExp(LockEventSet, UnlockEventSet)
}
```

Method `lock` is used to lock an instance of `LockObj`. Method invocations on a locked instance of `LockObj` are delayed until the `unlock` method is invoked on the instance. Expression `LockObjExp` represents interaction among the events of sets `LockEventSet` and `UnlockEventSet`. (For brevity, we have left out the definition of event ordering constraint expression `LockObjExp`.)

We construct a class `LockedQueue` that is composed from concurrent classes `LockObj` and `queue`:

```
concurrent class LockedQueue:  public LockObj, queue {
interaction:
   LockEventSet = LockEventSet+put:locked()+get:locked();
}
```

Invocations of `put` and `get` on an instance of `LockedQueue` are delayed if the instance is locked. This additional interaction behavior is specified by including invocations of `put` and `get` in set `LockEventSet`. ■

Models of concurrency: We now show that inheritance can also be used to represent different models of concurrency. The example below constructs an interleaving model by serializing all invocations of methods.

Example 5.5.4. *(Interleaving concurrency model).* In CYES-C++, methods of concurrent objects are concurrent by default. However, certain applications may require that executions of method invocations be interleaved. In order to represent such concurrent classes, the following concurrent class can be defined:

```
concurrent class interleaving {
public:
         ⋮
private:
      ⋮
interaction:
   AllInvocations = {}
   Interleave(AllInvocations) =
      MutuallyExclusive(AllInvocations, AllInvocations);
}
```

Expression `MutuallyExclusive(S1, S2)` is defined in Section 3.5.1. A concurrent class, say `userclass`, can thus be defined as an extension of the `interleaving` class:

```
concurrent class userclass:  public interleaving {
public:
      ⋮
private:
      ⋮
interaction:
   AllInvocations = AllInvocations + S
}
```

For methods M1, M2, $\cdots$, and Mn of `userclass`, event set `S` is defined:

```
S = M1 +M2 + ⋯ + Mn
```

Executions of methods of `userclass` satisfy all ordering constraints specified on the events of set `AllInvocations`: the method invocations are mutually exclusive. ■

5.6 Generic Concurrent Classes

The C++ programming language provides the template mechanism for specifying generic classes. Templates allow one to capture essential elements of objects or functions. In this section, we describe the manner in which the template mechanism can be extended to define generic concurrent classes.

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

Example 5.6.1. *(Generic sync class).* CC++ [CK92] supports the notion of `sync` synchronization variables. A `sync` variable is a write-once variable. All reads to the variable are delayed until the first write has taken place. We define a generic `sync` class in the following manner:

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

The expression `Interaction(WriteSet, ReadSet)` is defined as:

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

Methods `read` and `write` are defined:

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

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

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

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

Note that the generic `sync` class specifies a default implementation for the `read` and `write` methods. The user may override the default implementations through inheritance:

```
template<class T>
concurrent class userclass:  public sync <T> {
public:
   T & read();
   write(T &);
}
```

In addition, sets `ReadSet` and `WriteSet` can be changed to include any method invocations. The `sync` class therefore not only captures the abstraction associated with write-once variables but also the abstraction of interaction relationships between the `read` and `write` methods. ■

Example 5.6.2. *(Generic mailbox class).* Another example of a generic concurrent class is a first-in-first-out mailbox class. This class captures the abstraction associated with storing and retrieving information from a first-in-first-out mailbox. It includes interactions between sends and receives over the mailbox. The following describes the essential elements of a generic mailbox class:

```
template<class T>
concurrent class mailbox {
public:
   virtual T &receive();
   virtual send(T &);
private:
   T data[MAILBOXSIZE];
   int size;
interaction:
   RemoveSet = {receive};
   AddSet = {send};
   MailboxInt(RemoveSet, AddSet)
}
```

For brevity, we omit the definitions of `send` and `receive` here. One possible definition of event ordering constraint expression `MailboxInt(RemoveSet, AddSet)` is shown below:

```
MailboxInt(RemoveSet, AddSet) {
   forall var i in AddSet {
         (AddSet[i] < ReadSet[i])
   } &&
   forall var i in RemoveSet {
         (RemoveSet[i] < AddSet[i+MAILBOXSIZE])
} }
```

The `mailbox` generic concurrent class can now be instantiated with a user class to create user-specific mailboxes. Note that the event ordering constraint expression `MailboxInt` is dependent on the size of the `mailbox`. This limits the possible ways in which the `mailbox` generic concurrent class can be instantiated. However, it is possible to redefine the `mailbox` generic class such that a set of boolean conditions are used to specify the interaction relationships between events of `AddSet` and `RemoveSet`. Event ordering constraint expression here will be similar to the ones defined in the `queue` concurrent class (example 5.4.1). ■

The template and concurrent class mechanism can therefore be used to define generic concurrent classes that capture essential concurrency, interaction, and computational attributes of concurrent classes. These generic classes can then be composed with other classes to construct concurrent classes.

5.7 Summary

We presented the design of a concurrent object-oriented programming language CYES-C++. The language includes concurrency in the C++ programming language with the concept of a concurrent class type and synchronous and asynchronous method invocation mechanisms. It contains mechanisms for representing interactions among method invocations through the notion of event sets and event ordering constraint expressions. It supports inheritance of both method and interaction specifications. More importantly, it supports creation of abstractions associated with concurrency and interaction. These abstractions can be composed with other abstraction mechanisms (such as template and inheritance) of C++ to construct reusable concurrent classes.

Chapter 6

Implementation of CYES-C++

6.1 Introduction

In this chapter we briefly describe the design and implementation of a translator for the CYES-C++ programming language. The goals are to demonstrate the feasibility of the implementation of a concurrent programming language based on “separation of concerns”, and to establish an experimental platform for the performance analysis of the execution behaviors of CYES-C++ programs.

6.1.1 Problem

The problem of implementing CYES-C++ can be partitioned into two subproblems: the implementation of different aspects of a concurrent object-oriented programming language, and the implementation of those aspects that are specific to the C-YES model.

Implementation of concurrent object-oriented programming languages aspects

An implementation of a concurrent object-oriented programming language includes implementation of concurrent objects, concurrent method invocations, object distribution, object communication, and load balancing. One problem is the management of communication and method invocations among concurrent objects that reside on different processors. In most implementations [TMY93, CKS93, Kal90, GWS93, Nak91, KC93, BPG$^+$93] of concurrent object-oriented programming languages, a globally unique representation of a concurrent object is used to determine the processor on which the object is located. Differences among the approaches occur in the implementations of the global representation of concurrent objects. In the CYES-C++ implementation, a concurrent object is represented by a pair — name and value — of C++ objects. The name object implements the mechanisms needed for accessing a remote value object. The value object, on the other hand, stores the state of the concurrent object.

The other implementation problems are load balancing and object distribution, which involve distribution of objects across different processors in a way that minimizes the cost of communication among the objects while keeping the load on the processors balanced. The two goals are orthogonal. There may not be an optimal solution for the most general problem. Some programming languages [CGH92, CK92] therefore provide language constructs for specifying how objects should be placed on processors. Currently, the CYES-C++ implementation employs a simple algorithm for distributing objects: an object is placed on a processor with the least load. The algorithm does not take communication cost into account. In certain applications, this algorithm may be extremely inefficient. We plan to extend the object distribution algorithm in order to incorporate additional information such as interaction relationships among objects, memory hierarchy, machine load factor, and machine configuration.

Implementation of C-YES model aspects

The two unique elements — separation of concerns and event ordering constraint expression — of the C-YES model are implemented in the following manner:

Separation of concerns: In the C-YES model, specifications of computational and interaction behaviors of programs are separated. Separation of the two behaviors creates complications in that the CYES-C++ translator must find suitable places in source programs where synchronization code for components can be inserted. The synchronization code should be placed in such a way that its execution implements the interaction behaviors of the components.

Interactions are specified at two places in CYES-C++ programs: one in concurrent class definitions (see Section 5.2.1), and the other in method invocation expressions (see Section 5.3). Implementation of the first involves establishing and preserving execution orderings among method invocations. Event ordering constraint expressions specified in a concurrent class can be easily implemented by generating two synchronization procedures for each method of the class. One, called the prefix of the method, is executed before the execution of an invocation of the method, while the other, called the postfix of the method, is executed after the execution of a method invocation. The execution of the prefix ensures that all ordering constraints associated with a method invocation are satisfied. The execution of the postfix, on the other hand, wakes up all method invocations that are waiting on a method invocation. The execution behavior of an invocation of a method `M` is therefore represented in the following manner:

```
M_prefix();
M();
M_postfix();
```

In this program `M_prefix` and `M_postfix` are the prefix and postfix of `M` respectively.

Event ordering constraint expressions defined in method invocation expressions are used to specify interaction behaviors of interaction points of calling and called methods. Implementation of such event ordering constraint expressions is much more difficult. The reason is that it is difficult to find a suitable place where the synchronization code for the interaction points of the calling and called methods can be inserted. There are two possible implementations:

1. The method specifications can be modified so that synchronization codes are inserted before and after the textual occurrences of interaction points. However, this involves modifications of the specifications of the calling and called method.

2. The generated synchronization code can be associated dynamically with an interaction point (method invocations on an object) in the object upon which the method is invoked. However, this requires the ability to access the context of the invoked method from within an object. This is made more difficult by the fact that the invoking and invoked methods may reside on different processors.

Currently, we have not implemented event ordering constraint expressions specified in method invocation expressions. While this does not restrict the set of application programs that can be written, it does have implications on the extensibility and reusability of programs. (See Section 4.5 for more detail.)

Event ordering constraint expressions: Interactions in CYES-C++ are defined in terms of the primitive event ordering constraint expression and the interaction composition operators. The language includes mechanisms for defining event sets. Note that event sets may be dynamic in nature in that their content may change over time. Also, they may contain infinite events. Event ordering constraint expressions containing such event sets may assert infinite possible ordering relationships. They may also define relationships among events that may occur in the future. Such event ordering constraint expression cannot be evaluated by synchronization code that statically defines and preserves all relationships among method invocations. Our approach therefore is to generate synchronization code that incrementally evaluates event ordering constraint expression. In this approach, execution of each method invocation creates and preserves all possible relationships of which it may be a part.

6.1.2 Design goals and approach

The primary concerns of the design are: portability, support for true concurrency within concurrent objects, and reusability. We describe each in detail below:

Portability: One of the primary design goals was the portability of the translator across both shared and distributed memory machines. We attain this goal by designing the runtime system of CYES-C++ as a machine-independent abstraction of a parallel machine. Here a parallel machine is represented as a collection of execution spaces and mapping functions.

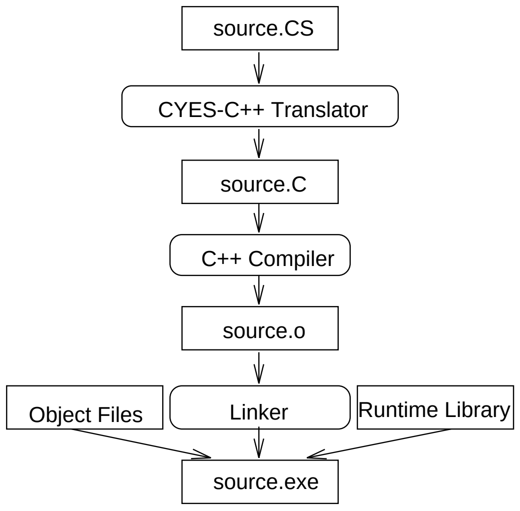


Figure 6.1: Compilation and linking steps of a CYES-C++ program

An execution space supports the ability to execute multi-threaded programs. The mapping functions implement communication among programs that reside in different execution spaces. A CYES-C++ program is transformed into a concurrent program whose components execute in different execution spaces and communicate with each other through the mapping functions.

In figure 6.1, we show the different steps in the compilation of a CYES-C++ program. The translator parses a CYES-C++ source program, and generates a set of C++ programs. The generated programs contain transformed source programs along with calls to the runtime library. The programs are then compiled with a C++ compiler in order to generate object files. The object files are then linked with the runtime library creating an executable.

The translator can be ported easily by re-implementing the execution spaces and the mapping functions. No changes need to be made to either the front end of the compiler or the nature of the generated code.

Intra-object concurrency

There are two reasons for supporting true concurrency within an object: the first arises from the semantics associated with the composition of a concurrent object. Here, method invocations are concurrent by default. The second is that true concurrency within concurrent objects can be used on represent massively parallel programs. We are interested in examining the execution behavior of such objects as well as exploring techniques for implementing them efficiently.

Our current implementation supports true concurrency by associating a thread of execution with every method invocation. One of the direct implications of this approach is that it significantly simplifies the code that is generated for the management and scheduling of method invocations. However, the approach is expensive both from computational resource usage and computational efficiency point of view. We are examining ways in which event ordering constraint expressions can be analyzed to determine if a thread should be created for a method invocation. For instance, an event ordering constraint expression that serializes events of a set can be analyzed to determine that only one thread needs to be created for all events of the set.

Development with libraries

We used the implementation of the translator as an exercise in testing the reusability of existing libraries. The intention from the beginning was not to write any code that existed in a usable form elsewhere. We evaluated many libraries in order to find those that are extensible and efficient, and conform to a standard so that the translator can be easily ported to different machines. We came across many problems: most libraries were inadequately tested and had many bugs. Also, many C++ compilers had problems compiling certain constructs. Further, some libraries could only be compiled with specific compilers on specific machines. This exercise provided insights into the seamless integration of systems from predefined components. It also underlined difficulties due to incompatibilities in software development tools such as compiler and loaders, special requirements of the libraries, learning curves associated with the libraries, and difficulties in debugging.

We settled on using three libraries – i) a C++ parser library, ii) a standard C++ component library called STL (Standard Template Library) [SL94], and iii) a distributed thread library called Nexus [FGT94]. STL provides many template-based container data structures such as lists, vectors, sets, multisets, and maps. Nexus was designed and implemented at Argonne National Laboratory for implementation of back-ends for concurrent programming languages. It has been used for implementing the CC++ and Fortran-M [CF95] programming languages. It provides support for creating threads on a processor, for specifying synchronization among the threads of a node, and for communication among threads through the active-message paradigm. In the active message paradigm, it is possible to directly invoke a function on a remote node.

Status

The current CYES-C++ implementation runs on a network of IBM RS/6000 workstations. It supports creation and distribution of concurrent objects both on local and distributed nodes. In addition, both synchronous and asynchronous method invocations on local and remote objects are supported. Event ordering constraint expressions containing `forall`, &&, and primitive event ordering constraint expressions are supported. We chose these operators for implementation because these operators are most commonly used in applications.

This chapter is organized as follows: In Section 6.2 we describe the design of the runtime system. Section 6.3 describes the manner in which CYES-C++ programs are transformed. Finally we describe an implementation of event ordering constraint expressions in Section 6.4.

6.2 Runtime System

The runtime system supports creation of concurrent objects, concurrent method invocations, and evaluation of event ordering constraint expressions. It is implemented by a run-

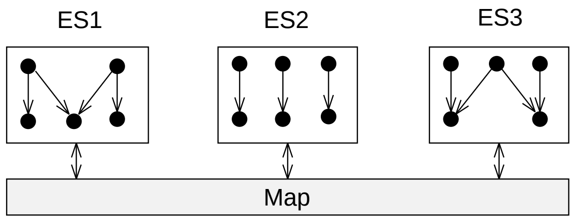


Figure 6.2: The runtime model of a parallel machine

time library. The primary role of the runtime system is to implement an abstraction of a parallel machine so that a CYES-C++ program can be mapped to the machine and executed. We first describe the parallel machine abstraction, and the manner in which CYES-C++ programs are executed. We then briefly describe the abstractions that the runtime system implements.

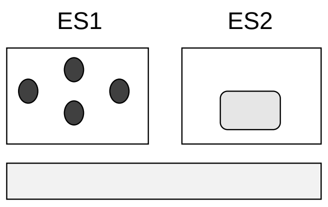
6.2.1 Model of Parallel Machines

In the runtime system, a parallel machine is modeled as a set of *execution spaces* and a *map* as shown in figure 6.2.

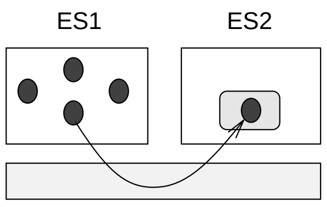
Execution Space

An execution space implements the elements necessary for executing multi-threaded distributed programs. It supports the ability to i) create and destroy objects, ii) construct threads of executions on a processor, and iii) specify synchronization among threads. It is used for implementing both intra-object and inter-object concurrency. For instance, intra-object concurrency is supported by constructing one execution thread for every method invocation. Further, inter-object concurrency is implemented by constructing separate execution threads for invoking and invoked methods.

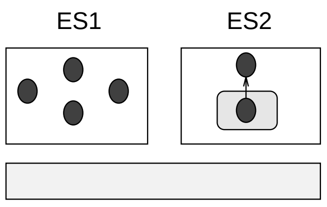
Implementation note: The concept of execution space is implemented by the node and context mechanisms of Nexus. In Nexus, a node is an abstraction of a physical processor. Within a node, multiple contexts can be created. A context is characterized by an address



(a) Create a named location in target execution space



(b) Map object into the named location



(c) Unmap object from the named location

Figure 6.3: Steps in mapping of an object from one execution space to another

space and an executable. In CYES-C++ runtime system, an execution space is represented by a node containing a single context. The executable associated with the context is the executable created after compiling and linking a CYES-C++ program. ■

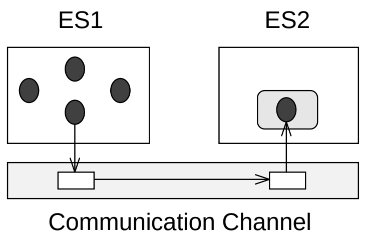
Map

Address spaces of execution spaces are disjoint. It is therefore not possible to directly access an object residing in a different execution space. We implement an abstraction, called *map*, for accessing remote objects. The following steps are used for accessing a remote object (Figure 6.3):

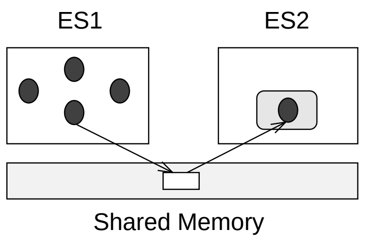
- Create a named address in the target execution space (figure 6.3(a)).
- Copy the object into the named location (figure 6.3(b)).
- Unmap the object from the named location(figure 6.3(c)).

A map is implemented by a set of mapping and unmapping functions. The steps in figures 6.3(a) and 6.3(b) are implemented by the mapping functions whereas the step in figure 6.3(c) is implemented by the unmapping functions.

A map provides an abstraction of the memory model of the underlying machine. Different maps can be implemented, each representing a specific memory model of the parallel machine. For instance, in distributed systems (figure 6.4(a)) where execution spaces



(a) Object mapping on a distributed machine



(b) Object mapping on a shared memory machine

Figure 6.4: Abstraction of memory models of parallel machines

may reside on processors with distributed memory, mapping an object involves i) copying the object into a message buffer, ii) sending the message over a communication channel to the remote processor, and iii) copying the message into a named location in the execution space. In shared memory systems (figure 6.4(b)), on the other hand, the mapping may involve copying the object into a named location in the shared memory and then copying a pointer to the named location into the remote execution space. If the object already resides in the shared memory, the first step can be eliminated. Similarly, for hierarchical systems (systems containing distributed clusters where each cluster is a collection of processors with shared memory), a map will depend on the characteristics of the source and target execution spaces to determine if a mapping requires distributed mapping, shared memory mapping, or a combination of the two. A map may also use machine specific characteristics of the execution spaces to efficiently implement mappings among the execution spaces.

This model of parallel machines is the basis for the portability of the CYES-C++ runtime system. The runtime system can be ported to different architectures by porting the implementation of execution spaces, and mapping and unmapping functions. Currently, we have implemented the mapping and unmapping functions for distributed systems.

6.2.2 Model of Execution of a CYES-C++ Program

We now describe the manner in which executions of CYES-C++ programs take place.

In order to create execution spaces on different processors, the user provides the names of the processors in a resource description file. The resource description file resides in the same directory as the program executable. An execution of the program starts by first creating an execution space on the processor on which the program execution is started. We call this execution space the *primary execution space*. The primary execution space is responsible for creating execution spaces, called *secondary execution spaces*, on other processors, initiating program executions, and terminating the CYES-C++ runtime system. The execution behavior of a program is partitioned into three distinct phases: i) initialization phase, ii) program execution phase, and iii) termination phase. We describe each in detail below:

Initialization Phase

During the initialization phase, the runtime system initializes the system data structures and starts different daemons. The primary execution space initiates this phase in the following manner:

1. Initialize local data structures.
2. Read the resource description file, and create secondary execution spaces on the specified hosts.
3. Create local *system state* and *system load* daemons. These daemons in turn create corresponding daemons in the secondary execution spaces.

The system state daemons maintain the global state of the runtime system. They keep track of information about execution spaces, concurrent objects, communication data structures, and states of different threads. State daemons interact with each other in order to apprise each other of changes in the state of the runtime system.

The system load daemons store load information of the execution spaces. Currently, a load daemon keeps track of the number of objects and threads that exist in its execution space. The load computation algorithm computes the load of an execution space in terms of numbers of objects and threads currently active in the execution space.

The load information is used for distributing newly created concurrent objects: a concurrent object is placed in an execution space with least load. Although there are provisions in the system for overriding this object distribution algorithm, they are mostly ad hoc (such as locating the aggregates of a concurrent object in the same execution space as the object). Also, the object distribution algorithm is clearly not optimal. We plan to change this algorithm in order to incorporate additional semantic information such as interaction relationships, memory hierarchy, load factor, and the underlying machine configuration in a future implementation.

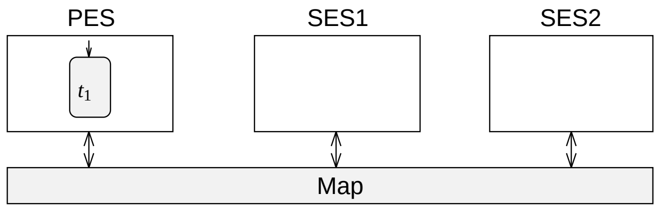
Program Execution Phase

During this phase, the execution of the user program takes place. It is started by transferring the control of the main execution thread of the primary execution space to the `main` procedure of the user program. The user program performs computations by creating concurrent objects in different execution spaces and by invoking methods on these objects. We describe the typical execution behavior of a user program by the following example:

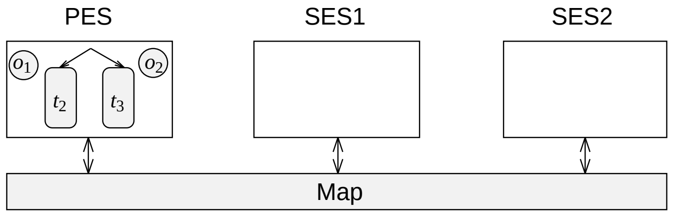
Example 6.2.1. *(CYES-C++ program execution).* We show an example execution of a program. We assume that there are three execution spaces, one primary execution space `PES` and two secondary execution spaces `SES1` and `SES2`. Figure 6.5 shows configurations of the execution spaces at different points in the program execution. We describe them below:

1. In figure 6.5(a) we show the state of the program, just before the program enters the program execution phase. The main execution threads of the secondary execution spaces `SES1` and `SES2` are blocked. Thread t_1 in `PES` is the main execution thread, and is about to start to execute the `main` subroutine of the user program.

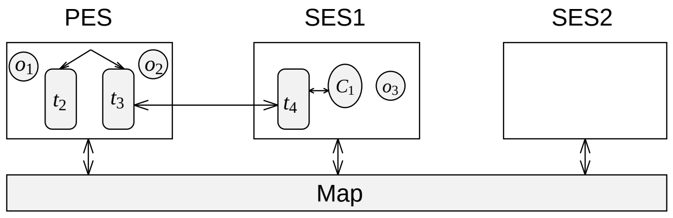
2. In figure 6.5(b) we show the state after thread t_1 has created local threads, t_2 and t_3, and sequential objects, o_1 and o_2. Note that creation of sequential objects, and



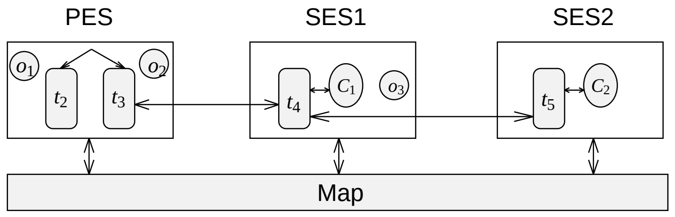
(a) System state before the execution of a user program



(b) System state after main thread creates local objects and threads



(c) System state after main thread creates a concurrent object and invokes method on the object



(d) System state after a method creates a concurrent object and invokes method on the object

Figure 6.5: Execution behavior of a CYES-C++ program

invocations of methods on sequential objects occur in the execution space in which they are invoked.

3. In figure 6.5(c) we show the state of the user program after thread t_2 has created a concurrent object, C_1, and has invoked a method on C_1. The concurrent object is created by first selecting an execution space (`SES1` in this case), followed by placing the object in the execution space. The method invocation on C_1 is executed by a thread, t_4, in `SES1`. Sequential object, o_3, in `SES1` is created by t_4. Note that a method invocation on a concurrent object occurs in the execution space in which the concurrent object resides.

4. Finally, figure 6.5(d) shows the state of the user program after thread t_4 has created a concurrent object, C_2, in `SES2`. Also thread t_5 executes the method invoked by thread t_4 on object C_2. ■

To summarize, creation of sequential objects and invocations of sequential methods take place in the execution space in which they are initiated. A concurrent object, on the other hand, may be created in an execution space different from the one in which the create operation was initiated. Also, method invocations on a concurrent object take place in the execution space in which the concurrent object resides.

System Termination Phase

The system termination phase involves gracefully terminating the runtime system. It starts once the main subroutine of the user program terminates. It is initiated in the primary execution space. The following steps are taken during the system termination phase:

- Terminate all load server daemons.
- Terminate global state daemons of the secondary execution spaces.
- Delete the secondary execution spaces.

- Release local resources. This is done by deleting the global state daemon of the primary execution space.
- Exit by deleting the primary execution space.

6.2.3 Runtime Abstractions

The runtime library implements abstractions for representing concurrency, synchronization, and communication.

Concurrency

The runtime library defines two classes for creating concurrent threads of execution: Class `LocalComputation` supports creation of threads in the local execution space, whereas class `RemoteComputation` supports creation of threads in remote execution spaces.

Implementation note: `LocalComputation` is implemented on top of Nexus's thread management utilities. `RemoteComputation`, on the other hand, is built on top of Nexus's communication facilities. ■

Synchronization

The runtime library defines two classes for specifying synchronization among threads in an execution space: Class `SynchVar` is a simple lock primitive. Class `CondSynchVar` implements a conditional synchronization primitive.

Implementation note: `SynchVar` and `CondSynchVar` respectively are class wrappers of Nexus's `nexus_mutex` and `nexus_cond` synchronization primitives. ■

Communication

The runtime library implements many abstractions for communication among threads that reside in different execution spaces. Two examples are `CommnicationBuffer` and `Mailbox`. CommunicationBuffer is a simple buffer mechanism. It is used for storing information that will be sent from one execution space to another. The `Mailbox` class implements named mailboxes in different execution spaces. Threads of different execution spaces can communicate with each other by appending to and retrieving information from named mailboxes.

Implementation note: The `CommunicationBuffer` abstraction is built on top of the communication mechanism of Nexus, whereas the `Mailbox` abstraction is built on top of the `CommunicationBuffer` abstraction. ■

Mapping and Unmapping Functions

Mapping and unmapping functions also implement communication between entities of different execution spaces. They manage connections between execution spaces through named mailboxes. Mapping and unmapping functions have the following form:

```
Map(SpaceMap &, ObjectType &);
Unmap(SpaceMap &, ObjectType &);
```

Class `SpaceMap` implements the necessary abstractions for mapping objects and functions from one execution space to another. The runtime library provides mapping and unmapping functions for primitive types such as integer, character, floating point, and string. These functions can be combined with other mapping functions in order to construct mapping and unmapping functions for user classes. Although the CYES-C++ translator can generate default mapping and unmapping functions for an object from its class definition, the programmer may need to override the default functions in cases when the object may have more than one possible mappings. One example is the mapping of a pointer object. Mapping a pointer object may involve either mapping the value of the pointer (the address of the object it points to) or the set of objects that it points to. While the former can be generated automatically, it may not be possible to automatically generate the latter. This is because it is not possible to determine the number of objects to which a pointer may address. In these cases the programmer must provide the mapping and unmapping functions.

We show both the default and one possible user-defined mapping and unmapping functions for a user defined class called `userclass`:

```
class userclass {
public:
            ⋮
private:
    int sz;
    exclas *ptr;
}
```

In this class specification, `size` contains the number of `exclas` objects to which the variable `ptr` points. In figure 6.6, we show both the default and user defined mapping and unmapping functions. The mapping and unmapping functions in figures 6.6(b) and 6.6(d) use the value of `size` to map and unmap objects.

In addition to the resolution of the semantic ambiguities, the programmer may want to map only specific aspects of a class. Also, she may optimize the mapping and unmapping functions on the basis of semantic information unavailable to the translator.

Mapping Methods for Concurrent Classes: The runtime library also supports mappings of methods of concurrent classes. For instance, an invocation of the form `obj->M(par1, ···, parN)` requires that method `M` be executed in the execution space in which `obj` resides. The runtime library provides a number of mapping functions for mapping methods from one execution space into another. For instance, the above method invocation is mapped in the following manner:

```
Map(smap, MethodIdentifier("M"), METHOD);
Map(smap, par1);
        ⋮
Map(smap, parN);
applyFunction(smap);
```

```
void Map(SpaceMap &sm,
         userclass &val)
{
 Map(sm, val.sz);
 Map(sm, val.ptr);
}
```

(a) Default mapping function

```
void Map(SpaceMap &sm,
         userclass &val)
{
 Map(sm, val.sz);
 for (i=0;i<val.sz;i++)
    Map(sm, val.ptr[i]);
}
```

(b) User defined mapping function

```
void Unmap(SpaceMap &sm,
           userclass &val)
{
 Unmap(sm, val.sz);
 Unmap(sm, val.ptr);
}
```

(c) Default unmapping function

```
void Unmap(SpaceMap &sm,
          userclass &val)
{
 Unmap(mp, val.sz);
 val.ptr = new exclas[val.sz];
 for (i=0;i<val.sz;i++)
     Unmap(mp, val.ptr[i]);
}
```

(d) User defined unmapping function

Figure 6.6: Mapping and unmapping functions for a user defined class

In the above, `smap` is a map associated with concurrent object `obj`. We first map the identity of method `M` (a unique integer). Mapping also includes mapping `M`'s arguments, and then applying the method on the remote node. Unmapping a method involves identifying the method, unmapping its parameters, and then executing the method with suitable parameters. Mapping and unmapping functions for methods are generated by the translator.

Implementation note: The mapping and unmapping functions are implemented in terms of named mailboxes. ■

6.3 Transformation of CYES-C++ Programs

We now describe the manner in which CYES-C++ programs are transformed. The transformation process can conceptually be divided into two parts: one involving the transformation of concurrent object-oriented programming language constructs such a concurrent class definitions and method invocations, and the other involving the transformation of constructs that are specific to the C-YES model. We describe the first part in this section. The implementation of the later is described in Section 6.4.

A CYES-C++ program contains class definitions, method definitions, object declarations, and method invocation expressions. These expressions form a program name space. In this name space, a program entity can access another entity within its scope. An example is an object's ability to invoke a method on another object that is defined in the same scope. The target space in which programs are executed, on the other hand, are represented by a set of disjoint execution spaces: an entity cannot directly access another entity that resides in a different execution space. The primary goal of the transformation process is to transform a CYES-C++ program in such way that an entity's ability to access another is preserved in the transformed program.

We describe the technique used by the CYES-C++ translator for the transformation of different CYES-C++ expressions below:

6.3.1 Transformation of Concurrent Class Specifications

The CYES-C++ translator constructs a representation for concurrent objects from their class definitions. The representation is an extension of the (l-value, r-value) [ASU86] representation of objects in a shared memory space. The r-value of an object denotes the value associated with the object. The l-value of the object is the memory location at which its r-value is stored. Note that in the shared memory space an object can be represented solely by its l-value: its l-value can be used to uniquely determines its r-value. In a partitioned memory space, however, the l-value is not sufficient since there are many possible r-values associated with an l-value (one for each execution space). A representation of an object must contain, in addition to its address, information about the execution space.

In a transformed program of CYES-C++ program, a concurrent object is represented by two C++ objects: a name object, and a value object. The name object contains the location (execution space and address) information of the value object. The value object (r-value) implements the instance variables and methods of the concurrent object. Any state change in the concurrent object is reflected by corresponding changes in its value object. The name object implements mechanisms for invoking methods on value objects. Note that name and value objects for a concurrent object may reside in different execution spaces. We show an example of the transformation of a concurrent class in terms of a name class and a value class below:

```
class queue_name {
public:
    queue_name();
    ~queue_name();
    void put(char val);
    char get();
    Boolean Full();
    Boolean Empty();
private:
    ExecutionSpace *space;
    Address *location;
        ⋮
};
```

(a) Name class of `queue` concurrent class

```
class queue_value {
public:
    queue_value();
    ~queue_value();
    void put(char val);
    char get();
    int full();
    int empty();
private:
    int size;
    char buffer[BUFSIZE];
private:
    void put_prefix(Event *);
    void put_postfix(Event *);
    void get_prefix(Event *);
    void get_postfix(Event *);
    void queueServer();
};
```

(b) Value class of `queue` concurrent class

Figure 6.7: Transformation of `queue` concurrent class

Example 6.3.1. *(Transformation of* `queue`*).* Figure 6.7 shows the (name, value) representation of concurrent class `queue`. Class `queue_name` stores (ExecutionSpace, Address) information of a `queue_value` object. Also, the public interface of `queue_name` is identical to that of `queue`.

Class `queue_value` contains methods and instance variables of `queue` object. In addition, it contains a number of additional methods such as `put_prefix` which are generated by the translator in order to evaluate event ordering constraint expressions. Execution behaviors of these methods are described in Section 6.4. ■

Methods of Name Objects: The sole purpose of the methods of name objects is to map method invocations and their arguments into the execution space of the value object, and execute the method invocations in the execution space of the value object. We show this through the following example:

Example 6.3.2. *(Methods of name objects).* we show an implementation of `put` method of `queue_name` below:

```
void queue_name::put(int val)
{
    InvokeFunction(_put_);
    Map(*map, val);
    applyFunction(*map);
}
```

The implementation of method `put` first maps a representation of method `put` (through `InvokeFunction` routine), maps its argument, and applies the function in the execution space of the `queue_value` object. Any values returned by the remote method execution are also returned by the method of the name object. ■

6.3.2 Transformation of Object Declarations and Method Invocations

We now look at how declarations of concurrent objects, and method invocation expressions are transformed.

Transformation of Object Declarations

Declarations of concurrent objects are transformed into declarations of name objects. For instance, in figure 6.8 we show declarations of `queue` concurrent objects (figure 6.8(a)) and their corresponding transformations (figure 6.8(b)). All references to `queue` concurrent class have been transformed into references to `queue_name`. We now show the

```
queue *qobj;
queue qarray[100];
        ⋮
qobj = new queue;
        ⋮
delete qobj;
```

(a) Declaration of concurrent objects

```
queue_name *qobj;
queue_name qarray[100];
        ⋮
qobj = new queue_name;
        ⋮
delete qobj;
```

(b) Transformation of concurrent objects

Figure 6.8: Transformation of concurrent object declarations

manner in which the above transformations of object declarations can be used for creating and deleting concurrent objects.

Object Creation: In a CYES-C++ program, a concurrent object is created when a variable declaration is within the scope of execution or is created explicitly by the new operator. For instance, declaration of `qarray` in figure 6.8(a) is used to creates 100 `queue` objects. In the transformed program, a name object and a value object are created for every concurrent object of a CYES-C++ program. It is achieved through the invocation of the constructor function of the name object. Note that since we have transformed a concurrent object declaration into a corresponding name object declaration, a concurrent object constructor invocation is transformed into a name object constructor invocation. For instance, in figure 6.8(b) the constructor of the `queue_name` class is invoked when `qobj` object is created by the execution of the `new` operator. The following steps take place during the execution of the name object constructor:

- Determine an execution space for a value object.
- Create a value object in the chosen execution space.
- Record the location and address of the value object in the name object.
- Initialize local data structures.

```
qobj->put(val);

        ⋮
val = qobj->get();
        ⋮
par qobj->put(val);
```

(a) Method invocation expressions

```
qobj->MakeMethodSynchronous();
qobj->put(val);
qobj->WaitForSynchronousMethod();
        ⋮
val = qobj->get();
        ⋮
qobj->put(val);
```

(b) Transformed method invocation expressions

Figure 6.9: Transformation of method invocation expressions

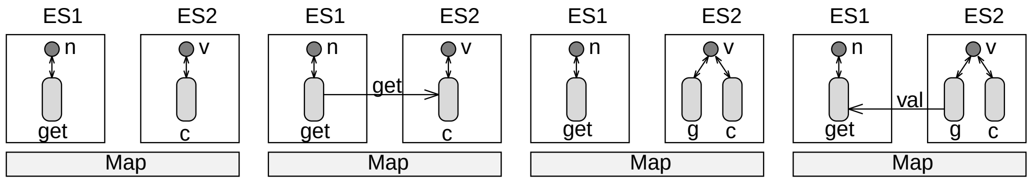
Object Deletion

Deletion of a concurrent object requires that the corresponding name and value objects be deleted. A concurrent object is deleted when its declaration goes out of scope or is deleted explicitly by the `delete` operator. Hence, in the transformed program, this occurs when the corresponding name object goes out of scope or is deleted. This in turn invokes the destructor associated with the name object. The destructor executes the following steps:

- Map the destructor to the value object. The value object executes the destructor and deletes itself.
- Release local data structures and resources.

Method Invocation Expressions: Method invocation expressions on concurrent objects are transformed into method invocations on name objects. For instance, in figure 6.9(a) we show method invocations on a `queue` object. Figure 6.9(b) shows the transformed method invocation expressions. The first `put` invocation is a synchronous method invocation. The calling thread therefore synchronizes by calling routine `WaitForSynchronousMethod`.

A method invocation on a concurrent object is therefore implemented by an invocation of the method on its name object. The execution of the method invocation on the named object maps the invoked method along with its parameter into the execution space



(a) Invocation of method `get` on a `queue_name` object

(b) `get` of `queue_name` maps the method invocation in execution space of value object

(c) Controller creates thread for `get` invocation

(d) Thread executes and returns value to `queue_name` object

Figure 6.10: Execution behavior of an object controller

of the value object. The mapped method is executed in the execution space. The name object therefore provides an interface between the invoking method and the value object.

Name classes therefore implement the mechanisms needed for creating, deleting, and invoking methods on remote value objects.

6.3.3 Implementation of Concurrent Objects

We now describe the manner in which concurrency and interaction within a concurrent object is implemented.

The semantics associated with a concurrent object specifies that method invocations on the concurrent object are concurrent by default. However, their executions must satisfy all ordering constraints. An implementation of the concurrent object must preserve this semantics. There are two aspects of the implementation: one involves creation and management of concurrent threads for method invocations, and the other involves implementation of event ordering constraint expressions. We describe the first here. The latter is described in Section 6.4.

There are two views of a concurrent object: one is the passive view which captures the state of the concurrent object. In the transformed program, the passive view of a concurrent object is represented by its name and value objects. The other is the active view which manages the execution environment of a concurrent object. The CYES-C++ translator creates a thread, called *controller*, for implementing the active view of the object. The controller is started during the creation of the value object. It first initializes the data structures and resources associated with the value object. It constructs an event for every method invocation, unmaps the arguments of the invocation, constructs a thread, and associates the event with the thread. The thread executes independently and performs the computations associated with the method.

In figure 6.10(a) we show the role that a controller plays in the execution of a method:

- Method `get` is invoked on a `queue_name` object (`n`) (figure 6.10(a)).
- The execution of `get` of `queue_name` maps the method into the execution space of the `queue_value` object (figure 6.10(b)).
- The controller (`c`) receives the method request, unmaps it along with its arguments, and constructs a thread of execution (figure 6.10(c)).
- The thread executes independently, and returns any results directly to the invoking thread (figure 6.10(d)).

6.3.4 Execution Behaviors of Method Invocations

We now describe the behavior of the thread associated with each method invocation. A method is executed only if all ordering constraints associated with the method are satisfied. The behavior of an invocation of method `M` on a value object `obj` is thus defined:

```
obj->M_prefix();
obj->M();
obj->M_postfix();
```

The execution of routine `M_prefix` ensures that all execution ordering relationships are created and maintained. The execution of method `M_postfix`, on the other hand, ensures that all events that are waiting for an event to finish are signaled when the event terminates. Both `M_prefix` and `M_postfix` are generated from event ordering constraint expressions.

6.4 Implementation of Event Ordering Constraint Expressions

Event ordering constraint expressions define execution ordering relationships among events. An implementation therefore must ensure that events occur in the order that satisfies the constraints specified in interaction specifications. There are two aspects of the implementation of the event ordering constraint expressions. The first is the technique used by the translator for generating code. The second is the nature of the generated code, and the manner in which it implements the semantics of the event ordering constraint expressions. We first described the code generation process.

6.4.1 Code Generation

The translator generates code by first transforming event ordering constraint expressions into a normalized form. It then uses the normalized expressions to generate the prefix and postfix routines for methods.

Normalization

The translator normalizes event ordering constraint expressions so that each normalized expression contains at most two nested levels of the `forall` operator and the primitive event ordering constraint expression. We show two examples of normalized expressions in figure 6.11.

The normalization process is based on the application of the following two rules:

```
forall var e1 in S1:B1 {
    forall var e2 in S2:B2 {
        (e1 < e2)
}}
```

(a) Normalized event variable expression

```
forall occ i in S1 {
    (S1[i] < S2[i])
}
```

(b) Normalized occurrence variable expression

Figure 6.11: Examples of normalized event ordering constraint expressions

```
1.  forall var e1 in (S1+S2) {E} =
        ( forall var e1 in S1 {E}) &&
        ( forall var e1 in S2 {E})
2.  forall var e in S {(E1 && E2)} =
        ( forall var e in S {E1}) &&
        ( forall var e in S {E2})
```

The first rule is based on the definitions of `forall` and event sets. It is used to remove set operators (such as $+$ and $-$) from event ordering constraint expressions. The second rule shows the distributive property of `forall` over &&. It is used to remove the && operator from inside the `forall` expressions.

The resulting normalized expression defines ordering relationship among events of two event sets.

Code Generation

The second step is to use the properties of the event sets and the ordering relationships to generate prefix and postfix behavior of methods. The ordering relationships are preserved by methods when they execute their corresponding prefix and postfix. We now describe the nature of the generated code and how its execution implements the semantics of event ordering constraint expressions.

6.4.2 Implementation of Normalized Expressions

A normalized expression contains two entities: event sets and ordering relationships among the events of the sets. A correct implementation of the expression must ensure that all ordering relationships specified among the events are preserved.

An ordering relationship between events `e1` and `e2`

```
(e1 < e2)
```

can be implemented operationally by delaying the occurrence of `e2` until `e1` has terminated. Hence the prefix and postfix behaviors of `e1` and `e2` can be defined as shown in figure 6.12.

prefix of e1:

postfix of e1:
```
    Signal lockX;
```

(a) prefix and postfix behaviors of e1

prefix of e2:
```
    Wait on lockX;
```

postfix of e1:

(b) prefix and postfix behaviors of e2

Figure 6.12: Prefix and postfix execution behaviors of two ordered events

In the above, the two events share the primitive `lockX`. This suggests that the above implementation can be extended for a normalized expression in that the generated code defines a lock between each pair of events of event sets, and generates appropriate waits and signals on the locks for the events. However, the characteristics of event sets render this solution impractical. The reason is that event sets may contain an infinite number of events. Also, they may be dynamic in nature and their contents may change over time. Event ordering constraint expressions containing such event sets may not only assert infinite possible relationships but may also define relationships among events that do not exist or may never occur.

Our approach therefore is to generate code that incrementally evaluates event ordering constraint expressions. It is based on the following relationship between the `forall` and && operators:

```
forall var e in S {E(e)} =
    E(e1) && ( forall var e in S-{e1} {E(e)})
```

The above expression suggests that an event ordering constraint expression `E(e)` that holds true for all events `e` of a set `S` is equivalent to the expression in which `E` holds for a single event, `e1`, *and* for events of set `S-{e1}`. Hence, the `forall` expression can be implemented by ensuring that every event occurrence in `S` preserves all relationships specified in `E`. This suggests the following behavior for an event of `S`:

```
1.  Create relationship by evaluating E
2.  Wait for preceding events
3.  execute method
4.  wake up all waiting events
```

Items 1 and 2 form the prefix, and item 4 forms the postfix of an event of `S`. Both prefix and postfix are dependent on the event ordering constraint expressions and are generated from the specifications of the expressions. We now describe the actual generated code.

Generation of Prefix and Postfix of a Method

CYES-C++ supports two kinds of `forall` operators: one containing event variables and the other containing occurrence number variables (see table 5.1). We describe the generated code for the first `forall` operator.

Event Ordering Constraint Expressions Containing Event Variables: The most general event ordering constraint expression containing event variables has the following form:

```
forall var e1 in S1:B1 {
    forall var e2 in S2:B2 {
        (e1 < e2)
} }
```

The translator generates prefix and postfix for events of event sets `S1:B1` and `S2:B2`. They are shown in figure 6.13.

```
if (B1) {
  S1:B1 = S1:B1+e1;
  HappensBefore(e1, S2:B2);
}
```

(a) Prefix for events of set `S1:B1`

```
While (TRUE) {
  if (B2) {
    S2:B2 = S2:B2+e2;
    if (B1) {
      Wait on S1:B1 events
  }
}
```

(b) Prefix for events of set `S2:B2`

```
Signal S2:B2 events
S1:B1 = S1:B1-e1;
```

(c) Postfix for events of set `S1:B1`

```
S2:B2 = S2:B2-e2
```

(d) Postfix for events of set `S2:B2`

Figure 6.13: Prefix and postfix execution behaviors of methods

The prefix behavior of an event of `S1:B1` (represented by `e1`) is to record (through `HappensBefore`) that the event occurs before events of set `S2:B2`. This captures the relationship with both current and future events of `S2:B2`. Its postfix behavior is to signal the events of set `S2:B2`. The effect is that all waiting events of `S2:B2` are awakened.

The prefix behavior of an event of `S2:B2` (represented by `e2`) is to check condition `B1` in order to ensure that the event set `S1:B1` exists and that it must wait for events of `S1:B1`. It waits by delaying on a lock. When it is awakened by an event of `S1:B1`, it must recheck if the ordering relationships still exist. If they do, it waits again otherwise it exits the while loop. Its postfix behavior involves removing it from the event set `S2:B2`. Note that the ordering relationships among the events of `S1:B1` and `S2:B2` are preserved by each event. It does so by i) checking for ordering relationships with existing events, ii) setting locks if relationships exist, and by iii) waiting or signaling on the locks.

Event Ordering Constraint Expressions Containing Occurrence Variables: We now describe the postfix and prefix behaviors of events constrained by an event ordering constraint expression of the following form:

```
forall occ i in S1 {
        (S1[exp1] < S2[exp2])
}
```

In the above, `exp1` and `exp2` are integer expressions used for determining occurrence numbers of events. They are defined in terms of the occurrence variable `i`. The above expression specifies relationships between two specific events of `S1` and `S2`. The algorithm for prefix and postfix are shown in figure 6.14. The generated prefix behavior of each

```
occNum = occurrence number;
while (TRUE) {
  eval exp1(i);
  eval exp2(i);
  i = i+1;
  HappensBefore(exp1, exp2);
  if (i > occNum)
    break;
}
```

(a) Prefix for events of S1

```
occNum = occurrence number;
while (TRUE) {
  eval exp1(i);
  eval exp2(i);
  i = i+1;
  HappensBefore(exp1, exp2);
  if (i > occNum)
    break;
}
If (Related)
  Wait for S2 events
```

(b) Prefix for events of S2

```
Signal related events
S1 = S1 - e;
```

(c) Postfix for events of S1

```
S2 = S2 - e
```

(d) Postfix for events of S2

Figure 6.14: Prefix and postfix execution behaviors of methods

event evaluates `exp1` and `exp2`. This is done to determine the occurrences numbers of events that are related. There is a need to put constraints on allowable occurrence number expressions. This is because we need some mechanism to determine when an event should stop evaluating occurrence number expressions. The problem occurs because the occurrence variable `i` may range over the set of natural numbers. We require that occurrence number expressions be monotonically increasing. This property can be used to determine if an event should stop evaluating `exp1` and `exp2`: the event can stop when the occurrence variable `i` is greater than the occurrence number of the event. The reason is that by evaluating `exp1` and `exp2` until its occurrence number, the event has ensured that all relationships containing this event have been examined.

Each event records the occurrence number of events in a table. An event of `S2` is delayed if there is an ordering relationship for the event in the table. Similarly an event of `S1` uses the table to signal events of `S2`.

6.5 Summary

We have presented the design of a translator for the CYES-C++ programming language. The problem of implementing the language can be divided into two subproblems: one common to concurrent object-oriented programming languages and the other specific to the C-YES model. The first subproblem includes support for concurrent objects, concurrent method invocations, and object distribution. The CYES-C++ translator transforms program entities (such as class definitions, object declarations, and method invocation expressions) of a CYES-C++ program in such a way that objects can be created on different nodes in the generated program. Also, methods can be invoked on objects that reside on different nodes. Creation and deletion of objects as well as communication among the objects is supported through a runtime library.

The second subproblem involves implementation of event ordering constraint expressions. Event ordering constraint expressions are implemented by generating a prefix and a postfix for each method of a concurrent class from interaction specifications. The role of the prefix of a method is to ensure that all ordering constraints associated with invocations of the method are satisfied. Every method invocation therefore determines ordering relationships with existing events, and waits if the method's execution may violate the relationships. The postfix of a method is used by an event to signal all events waiting for the termination of the event. An evaluation of the event ordering constraint expression is therefore carried out incrementally and in parallel by different method invocations.

The translator supports both intra-object and inter-object concurrency. Also, it can be easily ported to different architectures by re-implementing a small set of classes.

Chapter 7

Experimental Results

7.1 Introduction

In this chapter, we describe the experiments we performed in order to analyze the different aspects of the CYES-C++ implementation. The primary goals of the experiments are the following:

- Validate that CYES-C++ provides compact and usable abstractions for representing concurrent programs.
- Show that concurrent programming languages based on "separation of concerns" can be implemented efficiently.
- Explore ways in which the implementation can be improved.

The focus of the experiments therefore is on i) measurements of the overheads of the implementation, ii) evaluation of performances of application programs, and iii) comparisons of the execution behaviors of application programs specified in CYES-C++ with the execution behaviors of corresponding programs specified in a language which uses low level primitives for specifying concurrency and interaction.

7.1.1 Experiment Execution Environment

The experiments were carried out on a network of RS6000 workstations, connected by an FDDI network. The Pablo instrumentation library [RAN$^+$94] was used for collecting data, and the xgraph tool was used for plotting the measurements.

7.2 Applications

We now describe the various applications and the results of the experiments.

7.2.1 System Overhead Measurements

We first present the measurements of the various overheads associated with the CYES-C++ implementation. We have divided this section into two parts. In the first part, we describe the various resource overheads associated with the current implementation. In the second part, we present the execution times associated with different aspects of the implementation.

Resource Requirements

The CYES-C++ implementation uses the thread and synchronization primitive resources of the Nexus thread library for implementing concurrency and interaction among method invocations. It is resource intensive in terms of the utilization of both thread and synchronization resources: We describe the usages of these resources below:

Thread Usage: Table 7.1(a) shows the thread requirements of the various aspects of the implementation. In the current implementation, two threads are created for every method invocation: one for storing the parameters of a method invocation in a named location, and another for executing the invocation. One of the threads can be eliminated by changing the implementation of the value object and the object controller such that a single thread is used for both storing the parameters and method executions. The thread overhead can be reduced further by analyzing event ordering constraint expressions in order to determine if

Implemented entity	Number of threads
Runtime system	2
Object	1
Method invocation	2

(a) Thread overhead

Implemented entity	Number of locks
System	3
Mailbox	1
Value object	6
Name object	2
Event set	1
Event	1

(b) Synchronization primitive overhead

Table 7.1: Thread and synchronization primitive overheads

a new thread should be created for a method invocation. For instance, we can deduce from the following expression

```
forall occ i in p {
    p[i] < q[i+N] }
```

that the thread for an event `p[i]` can be used for event `q[i+N]` as well.

Synchronization Resources: In table 7.1(b) we show the use of the synchronization primitives in various aspects of the implementation. Currently, all synchronization primitive resources are allocated statically. The synchronization primitive overhead can be reduced by modifying the implementation in the following manner:

- Dynamic allocation of the synchronization primitives for events: The synchronization primitives for a method invocation could allocated only if the method is included in an event ordering constraint expression.

- Allocation of the synchronization primitives from a pool: The runtime system could maintain a pool of synchronization primitives. Creation of a synchronization primitive would involve picking a primitive from the pool. Similarly, its deletion would involve returning it to the pool.

System overheads (in seconds)							
Processors	Termination	Object creation			Object deletion		
		(10)	(100)	(200)	(10)	(100)	(200)
2	.010	.124	.464	.975	.031	.021	.432
3	.012	.158	.552	1.127	.030	.074	.519
4	.013	.187	.593	1.181	.035	.076	.163
5	.016	.214	.630	1.254	.015	.078	.166
6	.016	.244	.680	1.351	.020	.263	.510
7	.017	.265	.705	1.434	.021	.268	.518
8	.019	.295	.758	1.543	.022	.267	.170
9	.020	.322	.814	1.663	.023	.261	.550
10	.023	.336	.875	1.782	.015	.083	.180
11	.026	.338	.963	1.910	.016	.264	.562
12	.028	.339	1.044	2.022	.015	.278	.487
13	.047	.342	1.121	2.164	.016	.250	.534
14	.032	.344	1.199	2.286	.016	.261	.558

Table 7.2: Execution times of operations such as system termination, object creation and deletion

7.2.2 Execution Time Measurements

We measured the execution time associated with the following operations: i) system termination, ii) creation of concurrent objects, and iii) deletion of concurrent objects.

Experiment

The measurements are carried out by defining a concurrent program that contains a concurrent class, called `queue`. The interface of the `queue` object is shown in figure 5.1. The concurrent program starts the CYES-C++ runtime system, creates and deletes a number of `queue` objects, and then shuts down the system. We recorded the execution times for each of these operations on 2 to 14 processors. Table 7.2 shows the costs associated with the different operations. We now analyze the execution times of the operations.

Object Creation: Figure 7.1 contains plots of execution times associated with creation of objects (as shown in table 7.2). We make the following observations:

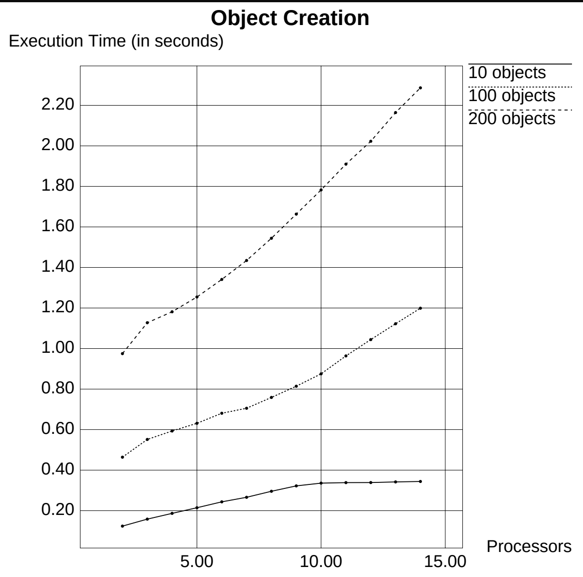


Figure 7.1: Execution times associated with creation of concurrent objects

- The cost of creating concurrent objects increases as the number of processors is increased. Ideally, this cost should decrease because objects are distributed over larger number of processors, thereby decreasing the load factors for the processors. It should therefore be faster to create new objects. However, we believe that the cost increase is due to increase in the number of messages that are exchanged among the execution spaces. Each object creation causes a change in the load factor of an execution space. This change triggers the local load daemon to inform other load daemons about the change in its load. As the number of processors increases, the number of messages that the load daemon must send increases as well. The performance can be improved by modifying the load daemon such that it does not send messages every time there is a change in its load. This phenomenon underlines the conflict between the effort to keep the loads of different execution spaces balanced and the performance effects of doing so.

- As the number of objects is increased, the execution cost increases as well. The average cost for creating an object on a 7 processor system configuration is 0.0265 sec-

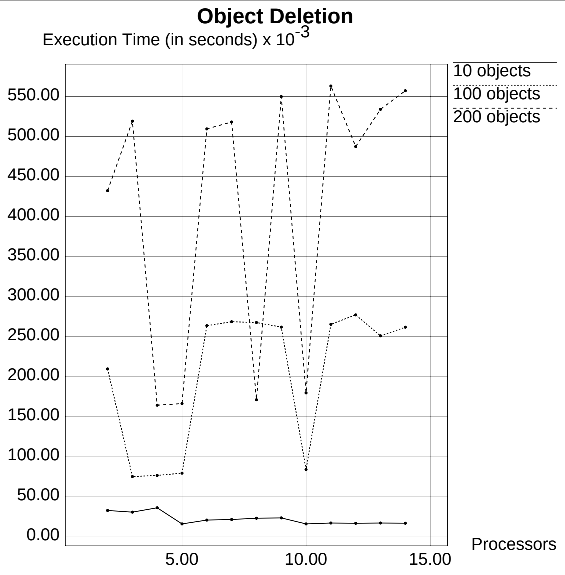


Figure 7.2: Execution times associated with deletion of concurrent objects

onds when 10 objects are created, 0.0705 when 100 objects are created, and 0.0717 when 200 objects are created. The differences in the costs, we believe, are again due to increase in the number of messages exchanged among the load daemons.

Object Deletion: Figure 7.2 contains the plots of the execution times of the deletion of objects. The cost associated with the deletion of object does not show any specific pattern. This, we believe, is because of the fact that the deletion of concurrent objects is implemented asynchronously. The figure only shows the costs of dispatching delete messages to the controllers. We believe that a synchronous version (invoking method waits for an acknowledgement from the controller) of the delete operation will have execution characteristics similar to that of the create operation.

System Termination: The system termination cost is shown in figure 7.3. The system termination involves i) termination of all daemons at remote nodes ii) shutting down the remote execution spaces, and iii) terminating local daemons. The system termination cost increases as the number of processors increases.

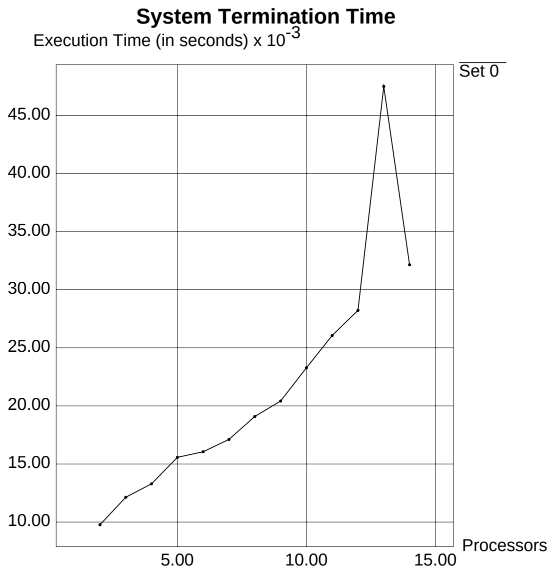


Figure 7.3: Execution times associated with termination of runtime system

7.2.3 Queue

In this application, a concurrent `queue` object (see figure 5.1 for its definition) is shared between two component programs. The component programs are represented as instances of a concurrent class called `comp`. The interface of class `comp` is shown below:

```
concurrent class comp {
public:
    comp();
    ~comp();
    void compute(queue & q);
};
```

The two components repeatedly add and remove information from the `queue` object. There are two goals of this application: i) test the implementation of the event ordering constraint expressions, and ii) evaluate the costs of method invocations and synchronization. A method invocation on a concurrent object is implemented by the following set of steps:

1. Map method and arguments into the remote execution space (map),

Method invocation costs (in seconds)					
Method	Map	Unmap	Prefix	Computation	Postfix
get	0.000215	0.000282	0.000479	0.000017	0.000331
put	0.000278	0.000181	0.000472	0.000017	0.000704

Table 7.3: Method invocation costs

2. unmap the method and arguments (unmap)

3. construct a thread for the method,

4. execute the prefix associated with the method (prefix),

5. execute the method (computation), and

6. execute the postfix (postfix).

In table 7.3, we show the costs associated with the different steps of methods `put` and `get` of the `queue` object. The table does not show the communication and thread creation costs. The method invocation cost can be reduced by the following optimizations:

- Generate prefix and postfix for methods only if they are included in event ordering constraint expressions.

- Make prefix and postfix methods inline functions.

- The current implementation always maps arguments of methods even if the name and value objects reside in the same execution space. This involves packing and unpacking data in communication buffers unnecessarily. By modifying the mapping and unmapping functions to recognize this special case, the extra cost can be avoided.

7.2.4 The Gaussian Algorithm

We implemented the *forward elimination step* of the Gaussian elimination algorithm. The goal of this experiment was to test the implementation of event ordering constraint expressions containing event sets with parameter variables.

```
concurrent class Matrix {
public:
    Matrix(int size);
    ~Matrix();
    float Read(int i, int j, int k);
    void Write(int i, int j, int k, float val);
private:
    int size;
    float mat[100][100];
interaction:
    Write(i, j, k)[0] < Read(i+1, j, k)[0]
    Write(i, j, k)[0] < Write(i+1, j, k)[0]
};
```

Figure 7.4: Interface of a concurrent class `Matrix`

The CYES-C++ program for the forward elimination step is derived completely from the one presented in Section 3.5.4. The program contains two concurrent classes: `pivot`, and `matrix`. The `pivot` concurrent class encapsulates the program associated with a pivot step. Its interface is shown below:

```
concurrent class pivot {
public:
    pivot();
    ~ pivot();
    void Pivot(int i, matrix A, int matsize);
};
```

Method `Pivot` implements the ith pivot step over concurrent object A. The concurrent class `Matrix` encapsulates a $n \times n$ matrix. Its interface is shown in figure 7.4.

Class `Matrix` supports methods `Read` and `Write` for accessing the elements of the matrix in parallel. For instance, method `Read(i, j, k)` is used to read the element `A[j][k]` of the matrix. Note that both `Read` and `Write` methods define an additional parameter, `i`. This parameter is needed because we have not yet implemented event ordering constraint expressions that are defined in terms of interaction points of methods. (In the case of the Gaussian elimination algorithm, interaction points of a pivot denote the pivot's reads and writes to the matrix object.) Our approach is to represent the interaction among the interaction points of the pivots in the matrix object itself. This is done by associating an additional parameter with the `Read` and `Write` methods. The additional parameter, `i`, identifies the pivot that invokes `Read` or `Write` methods on a `Matrix` object. For instance, the event ordering constraint expression

```
Write(i, j, k)[0] < Read(i+1, j, k)[0]
```

specifies the following interaction relationship between the ith and (i+1)st pivot: the first event in the event set `Write(i, j, k)` (denoting the ith pivot's writes to the element `A[j][k]`) occurs before the first event in the event set `Read(i+1, j, k)` (denoting (i+1)st pivot's read of the same element). Such expressions allow us to represent interaction relationships on the basis of the values of the parameters.

7.2.5 The Barnes-Hut Algorithm

The Barnes-Hut algorithm [BP86] is used for computing the positions of N particles in space. The positions of the particles change due to the gravitational forces they exert on each other. For instance, the acceleration a_i on a particle p_i due to a particle p_j is:

$$a_i = \frac{G \cdot m_j}{r_{ij}^2}$$

In this equation, G is the universal gravitational constant, m_j is the mass of p_j, and r_{ij} is the distance between p_i and p_j.

A simple approach to computing the positions of the particles is to calculate the acceleration between every pair of particles. An algorithm using this approach is shown below:

```
foreach time step $\Delta t$:
    foreach particle $p_i$
        foreach particle $p_j$
            $a_i = a_i + \frac{G \cdot m_j}{r_{ij}^2}$
        $v_i(t + \Delta t) = v_i(t) + \Delta t \cdot a_i$
        $p_i(t + \Delta t) = p_i(t) + \Delta t \cdot v_i(t)$
```

The computational complexity for each time step is $O(n^2)$, where n is the number of particles. The Barnes-Hut algorithm reduces the computational complexity of the above algorithm through the concept of *far* relationship between particles: a particle is far from another particle if its distance is beyond a certain constant. If all particles in a cluster are far from another particle, the cluster can be treated as a single particle when calculating the gravitational interaction among the particles of the cluster and a particle outside the cluster. Therfore, the algorithm for evaluating the acceleration between a particle and a cluster of particles is:

```
Acc(particle, ClusterOfParticles)
    if far(particle, ClusterOfParticles)
        Acc = PairwiseAcc(particle, ClusterOfParticles);
    else
        foreach cluster c in ClusterOfParticles
            Acc = Acc + Acc(particle, c);
```

In this algorithm, the space containing the particle is organized in terms of clusters. In the Barnes-Hut algorithm, the clusters are represented by quad trees. The details regarding the quad tree and its representations can be found in [CT92]. The simulation algorithm is shown below:

```
foreach time step $\Delta t$
    Create QuadTree from SetOfParticles;
    foreach particle $p_i$ in SetOfParticles
        $a_i$ = Acc($p_i$, QuadTree);
        move($p_i$, $a_i$, $t$);
```

A CYES-C++ Implementation of Barnes-Hut Algorithm

We define a CYES-C++ program for the Barnes-Hut algorithm by dividing the number of particles equally among a set of concurrent objects. The concurrent objects encapsulate the notion of spaces, quad trees, and particles. They are represented by a concurrent class, called `BarnesHutSolver`. This class supports methods for adding particles, constructing a quad tree, and computing the accelerations of particles. The interface of class `BarnesHutSolver` is shown below:

```
concurrent class BarnesHutSolver {
public:
    BarnesHutSolver(int myIndex);
    ~BarnesHutSolver();
    void CreateParticles(SetOfParticles &particles);
    void SetSolvers(SetOfSolvers &otherSolvers);
    void Compute(int iterations);
    SetOfParticles &GetParticles();
    Vector &Acc(Particle &part);
private:
    SetOfSolvers solvers;
    Space *spc;
    int numIt;
    int myId;
};
```

Class `SetOfSolvers` is used to store a set of `BarnesHutSolver` objects. Method `CreateParticles` adds particles to the space encapsulated by the `BarnesHutSolver` class. Each `BarnesHutSolver` object keeps track of other `BarnesHutSolver` objects. The main routine of the algorithm uses method `SetSolvers` for storing information about the different `BarnesHutSolver` objects with each `BarnesHutSolver` object. Method `Acc` computes the gravitational interaction between a particle and the particles of a `BarnesHutSolver` object. Method `Compute` is used to compute the position of the particles contained in the space encapsulated by the object. It is defined in the following manner:

```
create quad tree from particles
for each particle $p_i$
    for each solver in solvers
        $a_i$ = $a_i$ + solver.Acc($p_i$)
    $a_i$ = $a_i$ + space->Acc($p_i$);
    $p_i$.move($a_i$, time);
```

The parallel algorithm for the Barnes-Hut algorithm is therefore defined below:

```
create solvers = set of $m$ BarnesHutSolver objects;
create $n$ particles;
foreach solver in solvers
    solver.SetSolvers(solvers);
    solver.CreateParticles(set of $\frac{m}{n}$ particles);
for each iteration i
    foreach solver in solvers
        solver.Compute(i)
```

All interactions among the `BarnesHutSolver` objects take place through method invocations on the objects.

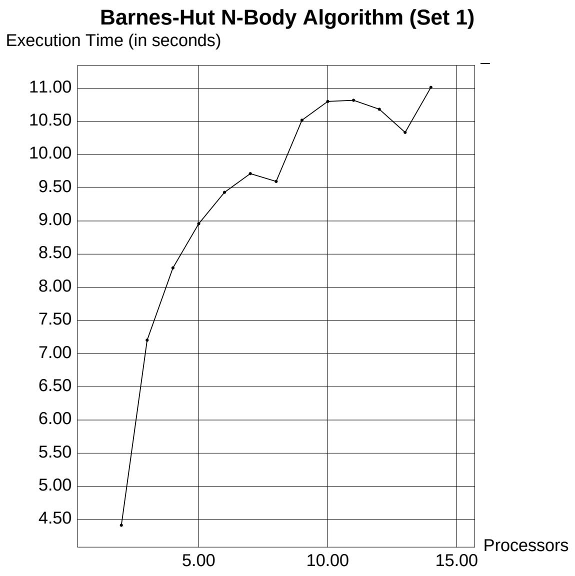
Experiment

We ran the Barnes-Hut algorithm for 100 particles over 10 iterations. The program was run on 2 to 14 processors.

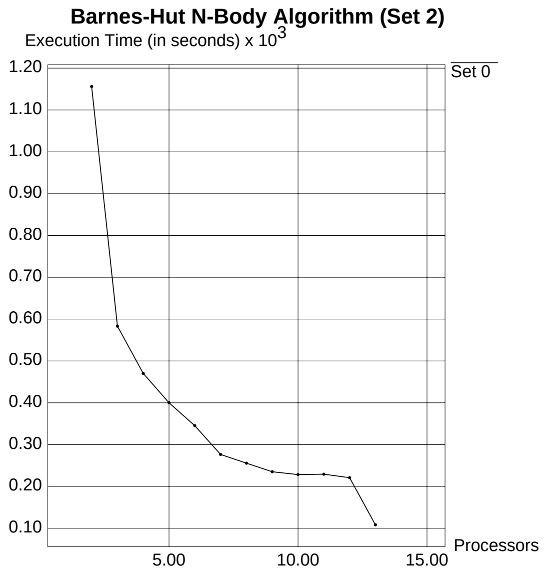
Measurements

In figure 7.5, we show the execution times for the Barnes-Hut algorithm on different numbers of processors. We ran two sets of experiments:

Set 1: We first ran a normal version of the Barnes-Hut Algorithm. Figure 7.5(a)) shows the execution times of the program for different number of processors. We observe that



(a) Set 1



(b) Set 2

Figure 7.5: Execution times for the Barnes-Hut algorithm

the execution time increases as the number of processors increases. This behavior occurs because, as the number of processors is increased, the number of remote method calls increases while the actual computation for each method invocation remains small. For N processors and P particles, the total number of remote method invocations is $P \times (N-1)$. The communication cost therefore increases linearly with the number of processors. The increase in the communication cost is not offset by a decrease in the cost of computation since there is little computation to execute.

Set 2: We ran another set of experiments in which we modified the particle acceleration routine by adding computation of 0.037464 seconds (the additional computation is a nested loop). This effectively increases the computation time associated with the acceleration routine. We made this modification in order to determine if the execution time of the algorithm improves as the number of processors is increased. Figure 7.5(b) shows the execution times for this set. Note that the execution time decreases as the number of processors is increased. This shows that additional performance can be achieved through addition of processors only if granularity of the computation is large with respect to the communication cost.

7.2.6 Parallel Evaluation of value of π

The primary goal of this experiment is to compare the performance behavior of the CYES-C++ implementation with the performance behavior of other concurrent programming languages. For this experiment, we chose to compare CYES-C++ against a low level concurrent programming language that uses C++ for specifying computation and the Nexus thread library for thread creation, communication, and synchronization. We will call this language C++Nexus. We chose C++Nexus because CYES-C++ is implemented on top of Nexus and C++. A comparison of the execution behaviors of CYES-C++ and C++Nexus will allow us to determine if the implementation of CYES-C++ is inordinately expensive in comparison to a low overhead concurrent programming language.

Algorithm

We use a parallel algorithm [GLS94] for the evaluation of the value of *pi* as a basis for comparing the two languages.

The value of pi can be evaluated by the integration of the function$\frac{1}{1+x^2}$:

$$\frac{\pi}{4} = \int_0^1 \frac{1}{1+x^2}$$

Function $f(x) = \frac{1}{1+x^2}$ can be integrated numerically in the following manner:

1. Divide the interval between 0 and 1 into n subintervals. Larger values of n give better approximations of the value of π. We chose n to be 100.

2. Evaluate the area enclosed by function $f(x)$ in each subinterval. The area for an interval between points x_1 and x_2 can be evaluated approximately by computing the area of the rectangle with length $(x_2 - x_1)$ and height $\frac{1}{1+\left(\frac{x_2-x_1}{2}\right)^2}$.

3. Sum the areas of the subintervals to determine the value of$\frac{\pi}{4}$.

In a parallel version of the above algorithm, computations of areas of the subintervals can take place in parallel. We describe the parallel versions of the algorithm below:

CYES-C++

In the CYES-C++ program, a concurrent class `picomp` supports methods for computing the area of a set of subintervals. The interface of `picomp` is shown in figure 7.6.

In this definition, variable `myId` stores an integer identifier for a `picomp` object. Variable `numProcs` specifies the number of objects used for evaluating the value of π and the variable `parts` stores the total number of subintervals. These variables are used to determine the subintervals that a `picomp` object evaluates. Method `compute` is used to evaluate the areas of the subintervals.

The concurrent program evaluates the value of π by first creating `n` `picomp` objects where `n` is the number of processors. It then invokes methods `compute` on the ob-

```
concurrent class picomp {
public:
    picomp(int id, int totalProcs, int n);
    ~picomp();
    double compute();
private:
    int myId;
    int numProcs;
    int parts;
};
```

Figure 7.6: Concurrent class specification of a component of a parallel algorithm for evaluation of value of π

jects. The objects execute the methods in parallel, and return the partial result to the main program. The main program adds the partial results to compute the value of π.

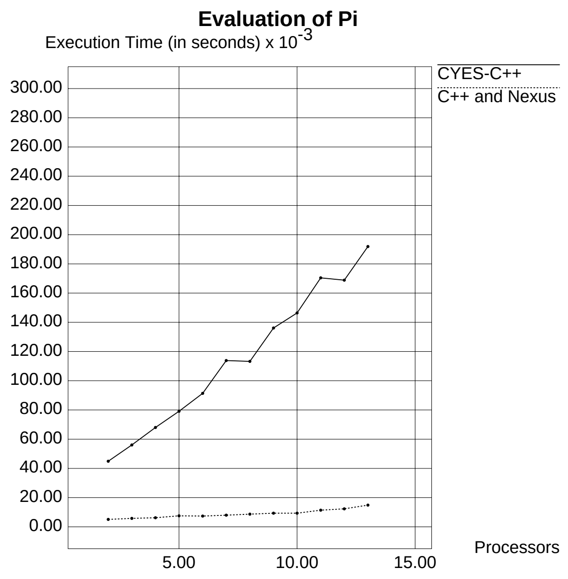
C++Nexus Program

In the C++ version of the program, the thread facility of the Nexus library is used to create `n` parallel threads on `n` processors. Each thread computes the areas associated with its subintervals, and returns them to the main thread. The main thread adds the results to compute the value of π.

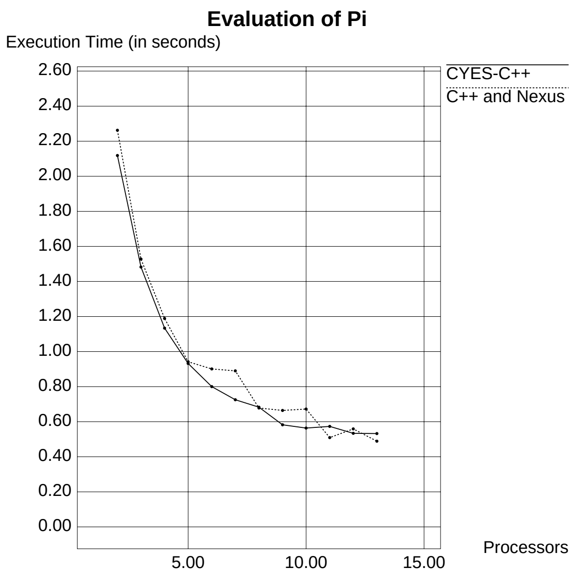
Performance Results

Both programs were run on 2 to 13 processors. We ran two set of experiments:

Set 1: We first ran a normal version of the programs.In figure 7.7(a), we show the execution times associated with the two programs on different number of processors. The execution time of the CYES-C++ program increases as the number of processors is increased. The reason is that the number of remote method calls increases as the number of processors is increased. However, since the actual computation for each invocation is quite small, The increase in the communication cost is not offset by a decrease in the cost of computation. In figure 7.7(a), we also show the execution time for the C++Nexus program. The execu-



(a) Performance behavior of the normal algorithm



(b) Performance behavior of the algorithm with additional computation of 4.2 seconds

Figure 7.7: Execution times for parallel π algorithms

tion time of the C++Nexus program also increases as the number of processors is increased. However, the increase is not as pronounced as it is in the case of the CYES-C++ program. The differences arise due to costs associated with method invocations, load balancing, object management, and extra thread creations in the CYES-C++ implementation.

Set 2: We ran another set of experiments in which we added extra computation (a loop of 4.2 seconds) to the algorithm. Figure 7.7(b) shows the execution times of the two programs on different processors. The performances of both CYES-C++ and C++Nexus programs improve as the number of processors is increased. We note that the execution behaviors of both programs are similar. Indeed, the performance of the CYES-C++ program is slightly better than that of the C++Nexus program. We surmise that this could be due to the fact that the system was more heavily loaded when the C++Nexus program was run. However, these measurements do show that the CYES-C++ implementation is not inordinately expensive compared to the C++Nexus implementation.

7.3 Summary

In this chapter we describe the experiments we carried out in order to measure different aspects of the CYES-C++ implementation. We have presented experiments designed to measure the overheads associated with different aspects of the CYES-C++ implementations. We have analyzed a number of techniques that can be used to reduce the overheads. For instance, in the current implementation, two threads are created for each method invocation. One of the threads can be eliminated by changing the implementation of the value object and the controller. Thread overhead can be reduced further by analyzing the event ordering constraint expressions to determine if a thread should be constructed for a method invocation. We have also presented a number of examples that showed the feasibility of the CYES-C++ language for concurrent programming.

The current implementation of CYES-C++ runs on a network of RS6000 workstations. In this system configuration, the cost of communication among the processors plays an important role in the design of concurrent programs. We showed that parallel speed-up can be achieved through additions of processors if there is sufficient computation in an application.

The experimental analysis of the CYES-C++ implementation has demonstrated that the programming paradigm is usable, and that implementation of a programming language based on “separation of concerns” is feasible.

Chapter 8

Summary and Conclusion

In this dissertation we developed a compositional approach to concurrent programming. The conceptual foundations for the approach are based on the postulate that computation and interaction are two orthogonal elements of concurrent programming, and they should be specified separately in a concurrent program composition mechanism. In Chapter 2, we showed that concurrent programs are difficult to extend and modify in concurrent programming approaches that do not separate specifications of computations and interactions. More importantly, there is a problem associated with the derivation of concurrent programs in terms of existing program abstractions. Program abstractions for representation of concurrency cannot be composed easily from existing program abstractions. Such compositions may often require changing the abstraction itself. Also, since programming languages use composition mechanisms for defining abstractions in terms of other abstractions, the inability to construct new program abstractions from existing program abstractions causes a breakdown in many of these composition mechanisms. We presented two examples of such breakdowns, one in inheritance and the other in aggregation, for concurrent object-oriented programming languages.

In Chapter 3 we developed a model of computation in which concurrent programs are composed from separate specifications of computations and interactions. The model includes a concurrent program composition mechanism, a representation of component programs, and an interaction specification mechanism. We applied the C-YES model to define an object-oriented concurrent programming language, CYES-C++. The benefits of the compositional approach become evident during the design of the programming language and application program development. Some of them are listed below:

- Concurrent programs can be easily extended by adding new component programs. Only the interaction behavior of the components may need to be changed. Note that in certain cases the redefinition of interaction may only involve adding new event ordering constraint expressions or modifying only a small subset of the event ordering constraint expressions. For instance, in example 5.5.1 the interaction behavior of the methods of the concurrent class `queue` is modified by extending only the event sets of the superclass. No changes were made in the event ordering constraint expressions of the `queue` class.

- A concurrent program can be easily modified by modifying interaction behavior specifications of the components. This supports a concurrent program design methodology where concurrent programs can be constructed quickly from existing core components and interaction behavior specifications.

- Specifications of component programs are encapsulated. It is therefore possible to change the implementation of a component without making any changes in other components, or their interaction behavior specification as long as the nature of computation, interaction behavior, and the interaction points do not change. For instance, in figure 3.22 we show an alternate implementation of the pivot program. The definitions of the `ForwardElimination` concurrent program and the `PivotInt` event ordering constraint expression do not change, since there are no changes in either the interaction points or the orderings among interacting events.

- Specifications of both computation and interaction can be reused.

- Another important aspect of "separation of concern" is that specifications of both interaction and computations are abstractions, which can be composed with other programming language abstractions in order to define further concurrent program abstractions. One example is the notion of generic concurrent classes developed in Chapter 5. A generic concurrent class is a composition of computations, interactions, and the template mechanism of the C++ programming language. It is used to capture common computational and interaction behaviors. It can be instantiated in various ways to associate different computational and interaction behaviors with methods of user defined classes. This supports reusability of both computational and interaction behaviors. Also, it raises the level of abstraction at which concurrent programs can be defined.

We extended the support for compositionality to the specifications of interactions as well. The interaction specification mechanism is declarative. It supports a modular approach to interaction specification. Global and complex interactions are specified by decomposing them into a set of local and simpler interactions. The local interactions can then be represented by event ordering constraint expressions, and combined with the suitable interaction composition operators to capture the global interactions. This approach allows one to change interaction behavior of programs by changing only the relevant event ordering constraint expression. Also, representations of local interactions can be reused in specifications of other interactions. We showed a number of examples in chapters 3 and 4 that highlight this property.

The interaction specification mechanism captures fundamental abstractions of interaction. It represents interactions by ordering relations among interacting events of programs. It is not based on the semantic properties of a specific synchronization primitive. Also, it does not depend on the semantic properties of the events. This can therefore be used to specify any interaction behavior for any invocation of any action. We showed the generality of the event ordering constraint expressions in two ways: one by representing many commonly used synchronization primitives as event ordering constraint expressions, and the other by modeling the composition of interaction specifications in a class as an $\textcircled{\wedge}$ composition of superclass and class interaction specifications. This model is general in that it represents models of inheritance of many concurrent object-oriented programming languages.

Most programming language abstractions that are general in their scopes are harder to implement and, often, their implementations are inefficient. The reason is that a compiler or a translator must generate code that implements the most general case. This is evident in our current implementation of CYES-C++ as well. The current implementation is inefficient from both the resource and the computational efficiency point of view. However, it is possible to apply advanced compiler techniques to construct an efficient implementation for CYES-C++. It can be done by identifying common patterns of interaction relationships and by optimizing their implementations. Similarly, the need for extra resources can be avoided by dynamically allocating and reusing existing resources. We believe that a finely tuned implementation of CYES-C++ will be as efficient as a comparable concurrent programming language. Indeed, it may be more efficient in certain cases because event ordering constraint expression capture more semantic information, which can be used to optimize the execution behavior of a program.

8.1 Future Work

The research work in this dissertation forms a beginning point for extensive research in the areas of parallel software development, programming languages design, analysis, and implementation, and formal verification. We briefly describe open questions in each of these areas:

Parallel software development

There are a number of areas in parallel software development that require further work:

Extension of the composition mechanism: Much of our effort has gone into deriving the concurrent program composition mechanism from specifications of computation and interactions. However, there are many other factors, — such as object distribution, program scheduling, and machine configuration — that affect the manner in which concurrent programs are designed and represented. Some of these elements represent aspects of the underlying machine architecture, and are crucial for the optimal execution behavior of concurrent programs. Other elements represent semantic relationships among components of a program.

The questions that we have not addressed in this dissertation relate to the role these elements play in the composition of a concurrent program: how do the elements affect the composition of a concurrent program? Are some of the factors orthogonal? How can they be represented? What is the nature of the concurrent program composition mechanism that includes these factors? Can the notion of "separation of concerns" be extended to these factors as well? How do these factors relate to our concerns regarding extensibility and modifiability?

Methodology: In this dissertation, we developed only the rudimentary aspects of a methodology for the design and implementation of concurrent programs for which computations and interactions are separated. Further work needs to be done in this regard. It includes development of mechanisms for identifications of components, their computational behaviors, semantic relationships among the components, separation of the two elements, and their representations.

Concurrent program and data structure libraries: The CYES-C++ concurrent programming supports mechanisms for defining generic and abstract concurrent classes. Definitions of such classes allow one to capture representations of common computational and interaction behaviors. Further work needs to be done in the representation of commonly used abstractions. Examples of these abstractions are i) concurrent container data structures such as queues, lists, trees, sets, and maps, ii) models of concurrency, and iii) and commonly used interaction patterns. The work not only involves representations of these abstractions but also the manner in which they should be organized in a concurrent class library.

Programming language abstractions

CYES-C++ supports two separate mechanisms for representing concurrent program structures: the concurrent class abstraction mechanism and the method invocation mechanism. During the development of applications, we observed that concurrent programs for certain applications contain specific structures. These structures determine how components of the concurrent programs are created and how they interact with each other. The structures either represent aspects of the machine architecture or capture certain application-specific semantic relationships. We noted that representations of such structures in terms of the concurrent program composition mechanisms of CYES-C++ were quite cumbersome. Also, their representations in terms of CYES-C++ abstractions lose certain semantic information that can be used for load balancing and object distribution. Further work is needed to determine how concurrent program abstraction mechanisms of CYES-C++ can be augmented in order to allow representations of the concurrent program structures easily.

In addition, certain features of CYES-C++ need to be reexamined, redefined, and/or extended: i) addition of a synchronous `parfor` method invocation mechanism, ii) support for global concurrent objects, iii) definition of scope rules for event ordering constraint expressions and their definitions, and iv) incorporation of execution space placement information with concurrent objects.

Implementation

Much work on the implementation of CYES-C++ remains to be done. Some of them are enumerated below:

- Implementation of the $\circledvee$ constraint operator.
- Implementation of event ordering constraint expressions that contain interaction points of methods.
- Implementation of inheritance.

- Efficient implementation of concurrent objects while supporting the notion of true concurrency.
- Compiler-based analysis of event ordering constraint expressions.
- Support for object co-location and object migration.
- Optimization of method invocations.
- Porting of the runtime system across different architectures.
- Investigation of the load balancing and object distribution mechanisms and their integration in the runtime system.
- Support for debugging and instrumentation.

Theory

In this dissertation, we developed only the preliminary aspects of semantics of the composition mechanism and the event ordering constraint expressions. We have not explored the possibilities of formally verifying certain properties of concurrent programs from their specifications. It is our belief that the C-YES model-based approach to concurrent programming is especially suitable for formal reasoning about concurrent systems. The reason is that the primitives and the operators of event ordering constraint expressions are well defined. It is therefore possible to verify certain safety and progress properties of the system from interaction behavior specifications in a rigorous manner. In addition, since specifications of computational and interaction behaviors of programs are completely separated, many properties of the programs can be verified — in isolation from component programs — solely from the event ordering constraint expressions.

The future work here includes development of the logic and proof structures for determining *safety* and *progress* properties of concurrent programs, as well as mechanisms for combining properties that are derived from computational and interaction behavior specifications. Further, relationships between the event ordering constraint expressions and temporal logic need to be explored.

Applications

In our research, we have experimented mostly with small examples. More complex and large examples from different domains such as computational sciences, simulation, VLSI, and visualization need to be implemented in order to fully evaluate the advantages and limitations of the C-YES model.

Bibliography

[ABB$^+$93] H. Assenmacher, T. Breitback, P. Buhler, V. Hubsch, and R. Schwarz. PANDA — Supporting Distributed Programming in C++. In *ECOOP*, pages 361–383, 1993.

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *Computer*, 19(8):26–34, 1986.

[AG94] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., second edition, 1994.

[Agh86] Gul A Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, Massachusetts, 1986.

[Ame87] 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.

[And79] S. Andler. Predicate Path Expression. In *Proc. Sixth ACM Symposium on Principles of Programming Languages*, pages 226–236, 1979.

[And81] G. R. Andrews. Synchronizing Resources. *ACM Transaction on Programming Languages and Systems*, 3(4):405–430, 1981.

[And91] Gregory R. Andrews. *Concurrent Programming*. The Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.

[AO93] G.R. Andrews and R.A. Olsson. *The SR Programming Language*. Benjamin/Cummings, 1993.

[AOC$^+$88] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An Overview of the SR Language. *ACM Transaction on Programming Languages and Systems*, 10(1):51–86, 1988.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communication of the ACM,* 21(8):613–641, 1978.
The language FP and its semantics are discussed in detail.

[BAS85] J.C. Browne, M. Azam, and S. Sobek. A Unified Approach to Parallel Programming. In *IEEE Software*, page 11, July 1985.

[Ber91] Brian N. Bershad. *The PRESTO User's Manual*. Department of Computer Science, University of Washington, Seattle, Washington 98195, Oct 1991.

[BF92] J. P. Bahsoun and L. Feraud. A Model for Designing Reusable Parallel Software Components. In *Parallel Architecture and Languages Europe, LNCS 605,* pages 245–260. Springer Verlag, 1992.

[Blo79] Toby Bloom. Evaluating Synchronization Schemes. In *Proc. 7th Symposium on Operating Systems Principles*, pages 24–32. ACM, 1979.

[BN84] Andrew Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[BP86] J. Barnes and P.Hut. A Hierarchical $O(N \log N)$ Force Calculation Algorithm. *Nature*, 324, 1986.

[BPG$^+$93] F. Bodin, Beckman P, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel System. In *Supercomputing Conference*, November 1993.

[Bri72a] Per Brinch Hansen. A Comparison of Two Synchronizing Concepts. In *Acta Informatica*, volume 1, pages 190–199, 1972.

[Bri72b] Per Brinch Hansen. Structured Multiprogramming. In *Communication of the ACM*, volume 15, pages 574–578, 1972.

[Bri73] Per Brinch Hansen. Concurrent Programming Concepts. volume 5, pages 223–245, 1973.

[Bro85] J.C. Browne. Formulation and Programming of Parallel Computers: a Unified Approach. In *Proceedings of International Conference on Parallel Processing*, pages 624–631, 1985.

[BS93] 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.

[Car93a] Denis Caromel. Toward a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9), September 1993.

[Car93b] Harold Carr. *Distributed C++*. PhD thesis, The University of Utah, November 1993.

[CF95] K. M. Chandy and I. Foster. A Deterministic Notation for Cooperating Processes. *IEEEPDS*, 1995. to appear.

[CG86] K.L. Clark and S. Gregory. Parallel Programming in Logic. *ACM transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[CGH92] 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.

[CH74] 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.

[CK92] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional Parallel Programming. Technical Report Caltech-CS-TR-92-13, Cal Tech, 1992.

[CKS93] Harold Carr, Robert R. Kessler, and Mark Swanson. Compiling Distributed C++. In *IEEE Symposium on Parallel and Distributed Processing*, 1993.

[CL89] 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.

[CM84] K.M. Chandy and J. Misra. The Drinking Philosopher Problem. *ACM TOPLAS*, 8(3):632–646, 1984.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Con87] John S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, 1987.

[CR88] E. Cooper and R. Rashid. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988.

[CT92] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming*. Jones and Bartlett, 1992.

[Dij65] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communication of the ACM*, 8(9):569, 1965.

[Dij68a] E. W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[Dij68b] E. W. Dijkstra. The Structure of the THE Multiprogramming System. *Communication of the ACM*, 11(5):341–346, 1968.

[DKM$^+$89] 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.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[FGT94] Ian Foster, John Garnett, and Steven Tuecke. *Nexus User's Guide*. Argonne National Labs, August 1994.

[FOT92] I. Foster, R. Olson, and S. Tuecke. Productive Parallel Programming: The PCN Approach. *Scientific Programming*, 1(1):51–66, 1992.

[Fro92] Svend Frolund. Inheritance of Synchronization Constraints in Concurrent Object–Oriented Programming Languages. In *ECOOP '92, LNCS 615*, pages 185–196. Springer Verlag, 1992.

[Gai88] Haim Gaifman. Modeling Concurrency by Partial Orders and Nonlinear Transition Systems. In *Linear Time, Branching Time, and Partial Order of Logics and Models for Concurrency*, number 354 in LNCS, pages 467–488, 1988.

[Gau] Philippe Gautron. Porting and Extending the C++ Task System with the Support of Lightweight Processes. In *Proceedings of the Usenix C++ Technical Conference*, pages 135–146.

[GC86] J. E. Grass and R. H. Campbell. Mediators: A Synchronization Mechanism. In *Sixth International Conference on Distributed Computing Systems*, pages 468–477, 1986.

[Geh84] Narain H. Gehani. *Ada: Concurrent Programming*. Prentice Hall, Englewood Cliffs, N.J., 1984.

[Geh93] 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.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transaction on Programming Languages and Systems*, 7(1):80–112, 1985.

[GL91] D. Gannon and J.K. Lee. Object-Oriented Parallelism: pC++ Ideas and Experimentations. In *1991 Japan Society for Parallel Processing*, pages 13–23, 1991.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing interface*. The MIT Press, Cambridge, Massachusetts, 1994.

[GR86] Narain H. Gehani and W. D. Roome. Concurrent C. *Software – Practice and Experience*, 16(9):821–844, 1986.

[GR88] Narain H. Gehani and W. D. Roome. Concurrent C++: Concurrent Programming with Class(es). *Software — Practice and Experience*, 18(12):1157–1177, 1988.

[Gri93] Andrew S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, 26(6):39–51, 1993.

[GWS93] Andrew S. Grimshaw, Jon B. Weissman, and W. Timothy Strayer. Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing. Technical Report CS-93-40, University of Virginia, July 1993.

[HCM94] Matthew Haines, David Cronk, and Piyush Mehrotra. *The Chant User's Guide*. ICASE, NASA Langley Research Center, August 1994.

[Hen88] Mathew Hennessy. *Algebraic Theory of Processes*. The MIT press, Cambridge, MA, 1988.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa72] C. A. R. Hoare. Towards a Theory of Parallel Programming. In C. A. R. Hoare and R. H. Perrot, editors, *Operating System Techniques*. Academic Press, NY, 1972.

[Hoa74] C. A. R. Hoare. Monitor: An Operating System Structuring Concept. *Communication of the ACM*, 17(10):549–557, 1974.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, 1978.

[Hud89] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Survey*, 21(3):359–411, 1989.

[HWA$^+$90] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughs, T. Johnsson, D. Kieburtz, R. Nikhil, S.P. Jones, M. Reeve, D. Wise, and J. Young. Report on the Programming Language Haskell. Technical Report RR 777, Yale University, April 1990.

[Kal90] L. V. Kale. The Chare Kernel Parallel Programming Language and System. In *International Conference on Parallel Processing*, pages II–17–II–24, August 1990.

[KC93] Vijay Karamcheti and Andrew Chien. Concert — Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Proceedings of Supercomputing*, 1993.

[KK93] 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.

[KL89] Dennis Kafura and Keung Lee. Inheritance in Actor based Concurrent Object-Oriented Languages. In *Proceedings ECOOP'89*, pages 131–145. Cambridge University Press, 1989.

[KL91] D. G. Kafura and G. Lavender. Recent Progress in Combining Actor-Based Concurrency with Object-Oriented Programming. In *ACM OOPS Messenger, Proceedings OOPSLA/ECOOP 90 workshop on Object-Based Concurrent Systems*, volume 2, pages 55–58, April 1991.

[KZ93] D. Kimeleman and D. Zernik. On-the-fly Topological Sorting for Interactive Debugging and Live Visualization of Parallel Programs. In *Third ACM ONR Workshop on Parallel and Distributed Debugging*, pages 12–20, May 1993.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[LSC81] P.E. Lauer, M.W. Shields, and J.Y. Cotronis. Formal Behavioral Specification of Concurrent Systems without Globality Assumptions. In *Lecture Notes in Computer Science*, volume 107, pages 115–151. Springer Verlag, 1981.

[Mat93] Satoshi Matsuoka. *Language Features for Re-use and Extensibility in Concurrent Object-Oriented Programming*. PhD thesis, The University of Tokyo, Japan, June 1993.

[Mes93] José Meseguer. Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming. In *Proc. 7th ECOOP'93*. Springer Verlag, 1993.

[Mil80] Robin Milner. A Calculus for Communicating Systems. *LNCS 94*, 1980.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[MTY93] 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.

[MWBD91] 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.

[MY93] 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.

[Nak91] Kiroaki Nakamura. Implementation of an Object-Oriented Concurrent Programming Language on a Multiprocessing System. In Y. Ohno, editor, *Distributed Environments: Software Paradigms and Workstations*, pages 39–50. Springer Verlag, 1991.

[NB92] P. Newton and J.C. Browne. The Code 2.0 Graphical Programming Environment. In *Proceedings of the ACM International Conference on Supercomputing*, July 1992.

[Neu91] Christian Neusius. Synchronizing Actions. In *ECOOP '91*, pages 118–132. Springer Verlag, 1991.

[Ost86] Anita Osterhaug. *Guide to Parallel Programming*. Sequent Computer Systems, Inc., 1986.

[Pan] Raju Pandey. CYES-C++ Reference Manual. Under preparation.

[Par72] David L Parnas. On the Criteria to be Used in Decomposing Systems into Modules. In *Communication of the ACM*, pages 1053–1058. ACM, 1972.

[Par90] G. D. Parrington. Reliable Distributed Programming in C++: the Arjuna Approach. In *Second Usenix C++ Conference*, pages 37–50, San Francisco, April 1990.

[Per87] Brinch Hansen Per. Joyce - A Programming Language for Distributed Systems. *Software – Practice and Experience*, 17(1):29–50, 1987.

[Pra86] Vaughan Pratt. Modeling Concurrency with Partial Order. *International Journal of Parallel Programming*, pages 33–71, 1986.

[RAN$^+$94] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B.W. Schwartz, and L.F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1994.

[RdP91] S. Crespi Reghizzi and G. Galli de Paratesi. Definition of Reusable Concurrent Software Components. In *ECOOP '91*, pages 148–165. Springer–Verlag, 1991.

[Rep88] J. H. Reppy. Synchronous Operations as First-Class Values. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, SIGPLAN Notices, Val 23, No. 27*, page 250259. ACM Press, 1988.

[RK83] Krithivasan Ramaritham and Robert M Keller. Specification of Synchronization Processes. In *IEEE Transactions on Software Engineering*, volume SE-9, pages 722–733, 1983.

[Rob92] J.A. Robinson. Logic and Logic Programming. *Communications of the ACM*, 35(3):41–65, 1992.

[RT78] D. M. Ritchie and K. Thompson. The Unix Time–Sharing Systems. *The Bell System Technical Journal*, 57(6):1947–1970, 1978.

[SC90] K. Stuart Smith and Arunodaya Chatterjee. A C++ Environment for Distributed Application Execution. Technical Report ACT-ESP-275-90, Microelectronics and Computer Technology Corporation, Sept. 1990.

[Sch86] David A. Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.

[Set94] Ravi Sethi. *Principles of Programming Languages*. Addison Wesley, 1994.

[SG91] Hayssam Saleh and Philippe Gautron. A Concurrency Control Mechanism for C++ Objects. In *Object-based Concurrent Computing Workshop Proceedings, LNCS 612* ,, pages 195–210. Springer Verlag, July 1991.

[Sha86] E.Y. Shapiro. Concurrent Prolog: A Progress Report. *IEEE Computer*, 1986.

[Sha89] Ehud Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[Shi89] Etsuya Shibayama. Reuse of Concurrent Object Descriptions. In *Concurrency: Theory, Language, and Architecture, LNCS 491*, pages 110–135. Springer–Verlag, 1989.

[SL94] A. Stepanov and M. Lee. *The Standard Template Manual*. Hewlett-Packard Laboratories, September 1994.

[SS87] B. Stroustrup and J.E. Shopiro. A Set of C++ Classes for Co-routine Style Programming. In *Proceedings and Additional Papers, C++ Workshop*, 1987.

[Ste73] G. W. Stewart. *Introduction to Matrix Computations*. Academic Press, New York, 1973.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Second Edition edition, 1991.

[Sun90] V.S. Sundaram. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[Tho92] Laurent Thomas. Extensibility and Reuse of Object-Oriented Synchronization Components. In *Parallel Architecture and Languages Europe, LNCS 605*, pages 261–275. Springer Verlag, 1992.

[TMY93] K. Taura, S. Matsuoka, and A. Yonezawa. An Efficient Implementation Scheme for Concurrent Object-oriented Languages on Stock Multicomputers. In *Proceedings of the Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*. ACM, 1993.

[TS89a] Chris Tomlinson and Mark Scheevek. Concurrent Object Oriented Programming Languages. In Won Kim and F. H. Lochovsky, editors, *Object Oriented Concepts, Databases, and Applications*, pages 79–124. ACM Press, 1989.

[TS89b] 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.

[Tur85] David A. Turner. Miranda: A Non-strict Function Language with Polymorphic Types. Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture. Springer-Verlag, 1985.

[Weg87] Peter Wegner. Dimensions of Object–Based Language Design. In *OOPSLA'87*, page 168. ACM Press, 1987.

[WZ88] 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.

[YBS87] 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.

[YT87] 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.

Vita

Raju Pandey was born in Patna, India on February 5, 1962, the son of Jagdish Pandey and Damyanti Pandey. In 1979, he joined Indian Institute of Technology at Kharagpur, India where he received the degree of Bachelor of Technology in 1984. His major area of study was Computer Science and Engineering. Subsequently, he entered the Graduate School of University of Massachusetts at Amherst where he received the degree of Master of Science in Computer Science in 1986. After graduation, he worked as a Software Engineer for Schlumberger Corporation for three and one-half years. In August 1989, he entered the Graduate School of the University of Texas at Austin.

Permanent Address: c/o Jagdish Pandey
East Lohanipur, P.O. Kadam Kuan
Patna, PIN 800-003
India.

This dissertation was typeset with LaTeX 2_ε by the author.