Specification of Multi-Object Coordination Schemes Using Coordinating Environments

by

Manibrata Mukherji

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE AND APPLICATIONS

APPROVED:

[illegible]

Dennis G. Kafura, Chairman

[illegible]	[illegible]
James D. Arthur	Adrienne G. Bloss
[illegible]	[illegible]
John A. N. Lee	Calvin J. Ribbens

July 1995

Blacksburg, Virginia

c.2

LD
5655
V856
1995
M854
c.2

SPECIFICATION OF MULTI-OBJECT COORDINATION SCHEMES USING COORDINATING ENVIRONMENTS

by

Manibrata Mukherji

Dennis Kafura, Chairman

Department of Computer Science

(ABSTRACT)

This dissertation proposes a coordination model for concurrent object-oriented programming languages (COOPLs). The model, termed *Coordinating Environments* (CEs), prescribes coordination among concurrently executing objects that compute as a group to achieve a common task or goal. The model represents coordination constraints and coordinating actions in a structured manner by grouping them into syntactic entities called Coordinating Behaviors (CBs). A group coordinator, termed a Coordinating Environment object (CE object), reduces the intrusive effects of coordination by transparently observing message-acceptance and method-termination events in components and triggering one or more coordinating actions on them. The conflict between the issues of *information hiding* (for better encapsulation) and *information externalization* (to enable coordination) is partially resolved by requiring components to provide state-interrogation methods. This allows a CE object to obtain and use local state information of components for the purpose of coordination.

A method for developing reusable coordination specifications in C++ is described. The method consists of two major steps: defining an abstract Coordinating Environment class (CE class) to capture the coordination problem in an abstract manner and then defining a concrete CE class (a subclass of the abstract CE class) to map the coordination effect embodied in the abstract CE class to a specific coordination problem. The method makes extensive use of the inheritance, polymorphism, and dynamic binding mechanisms of C++. Seven coordination problems, ranging from the coordination of a panel of buttons to the coordination of a multi-car elevator system, are specified to illustrate the method. A detailed design of the major components of the CEs model is also described.

The issues involved in using formal abstractions for coordinating process-agents specified in the Calculus of Communicating Systems (CCS) are also investigated. Using CCS directly to specify coordination has two weaknesses. First, coordination is modeled at a very low level in CCS by making agents engage in explicit communications. Such low-level specifications are poor candidates for specifying designs of software components that must satisfy software engineering criteria such as *separation of concerns* and *reusability.* Second, when the computation steps of the composition of agents are determined using the Expansion Law of CCS, many terms are generated that represent incorrect coordination sequences among the agents. Thus, the need for a calculus that addresses both the issue of concurrency and communication and the issue of effectively managing the communication among concurrent agents (that is, coordination) is identified, and the *Calculus of Coordinating Environments* (CCE) is proposed as a first step towards satisfying that need.

Acknowledgments

This dissertation, like any other, is the culmination of a long and difficult journey. During that trying time, I have been extremely lucky to have had the support, encouragement, and good wishes of an array of people who, knowingly or unknowingly, have contributed towards making my efforts worthwhile.

Among the faculty members at Virginia Tech, I thank my advisor, Dr. Dennis Kafura, for his cooperation, his patience, and his willingness to help. Knowing him and working with him over the last five years has been an enlightening experience for me. I would also like to thank my committee members for their constructive criticism of my work. They have helped me to be on track and not to lose sight of the goal. I thank Dr. James Arthur for teaching an excellent course in compiler construction and for the moments we spent learning and discussing Linda. I thank Dr. Adrienne Bloss for teaching a course in denotational semantics which inspired me to pursue and appreciate theoretical computer science. I thank Dr. Mary Beth Rosson for her encouragement during the Doctoral Symposium at OOPSLA'94. I also thank Dr. Verna Schuetz for her encouragement and her faith in me during a very critical period of my graduate career. I also thank the Department of Computer Science at the Oakland University for allowing me to use its resources.

Among my friends at Virginia Tech, I thank Jagdish and Sriram for studying with me for the qualifier examinations. I have learnt a lot from them during those intense months of preparation. I thank Ben for being a friend and for being able to share with him the feelings of a Ph.D. student searching for a dissertation topic. I thank Rajiv for being a very good friend, a formidable opponent in table tennis, and for the good times we have shared. I also thank Greg, Keung, Siva, and Anup for being good friends. And a special thanks to Bhupat and Mridula: knowing the two of you has been a real pleasure.

Among my family members, I thank my parents for having faith in me, for their encouragement, and for bearing the stress and the hardships of sending me abroad to pursue higher studies. I would especially like to thank my father for his constant encouragement and his guidance to remain on track in life. I thank Suntu for being a wonderful sister and for all the good times we have shared. I thank my mother-in-law and my father-in-law for their faith in me and for sharing, approving, and encouraging me to pursue my academic aspirations. I thank Dada and his family for the good times we have shared. I thank my grandmother for her love and care: you will always have a special place in my heart. I thank Sarojda for inspiring me and for all the good times we have shared. And a special thanks to Didi and Prabirda: your love and your kindness have been crucial in shaping every moment of the last eight years. Finally, I thank Mr. Sarkar for his pledge of support when it was needed the most.

In spite of the good-wishes and the support of all the above people and of the several whom I could not acknowledge in this limited space, I am very lucky to have the whole-hearted support of a very special person, my wife, who has shared every moment of about half of my life with me: the joys and the sorrows, the successes and the failures, the ups and the downs, and the pleasures and the pains. I am fortunate to have such a person beside me and I thank her for sharing with me every moment of an intense and eventful graduate career, for her prayers, for cheerfully bearing many hardships, for pursuing a successful career of her own, and for giving me the opportunity to finish this dissertation. I dedicate this dissertation to her.

Contents

List of Figures

Chapter 1

Object-Group Coordination Models

1.1 Introduction

Multi-object groups are an important abstraction mechanism for structuring object-based, client-server software systems. Such groups encapsulate functionality and simplify communication patterns between a group and its clients. The objects participating in a group (the **components** [1] of the group) either cooperate or compete with each other to provide some service to its clients. Commonly used examples of object groups are the components of a vending machine, the components of an elevator system, and a set of radio buttons. Clients of such groups request service by explicitly communicating with one or more components of the group.

Although composed of multiple objects, a group projects itself as a coherent, logical entity to its clients. To achieve this **group coherence**, the components must be **coordinated**. Coordination, in this context, refers to the actions involved in **operation scheduling** and **state propagation**. Operation scheduling refers to the need, when scheduling an operation in an object, to take into account the local states of other components in the group in addition to the object's local state. **Local state**, in this context, refers to the values of the local instance variables in an object. Operation scheduling insures that an individual object does not take an action that is inconsistent with the constraints of the group of which it is a member. State propagation refers to the act of propagating the ef-

[1] Henceforth, whenever a term or a concept is introduced for the first time, it will be printed in italics. Bold letters will be used to emphasize important terms or concepts.

fects of local actions in an object to other objects in the group through a timely exchange of data. This insures that a change in the condition of the object is made known outside of the object in which the change takes place. Operation scheduling and state propagation will be illustrated by means of a simple example, that of a vending machine, in subsequent sections of this chapter. The vending machine is assumed to have three component objects, a coin acceptor (with methods *Insert, RefundAll, RefundExcess*, and *InsertedAmount*) and two slots (each with methods *RequestItem*, *DispenseItem*, *HowManyItems*, and *Price*). The coin acceptor accumulates and returns coins. The slots store and dispense candies. It is the responsibility of the coordination mechanism to impose a coordinated behavior on the components so that the ensemble behaves like a vending machine. The coordination mechanism must insure that operations are scheduled so that: (i) no slots open when no coins have been inserted and, (ii) after an item is extracted, the excess amount is refunded and no slots are allowed to be opened until more coins are inserted. The coordination mechanism must also ensure the propagation of local state changes so that the deposited amount is transmitted from the coin acceptor to a slot and the deposited amount is set to zero after a candy is extracted. This example will be reconsidered in Chapter Three to show how these coordination requirements are realized using the proposed mechanism.

The problem of specifying and maintaining group coherence is particularly difficult when a group consists of **concurrently executing objects** which interact with **unsynchronized clients**. Such objects, equipped with their own threads of control, are always prepared to interact with clients and with other objects in the group. An unsynchronized client is one that uses the services of a group without regard for the internal consistency of the group or the possible effects of its actions on other clients. Thus, unlike **synchronized clients**, who dispatch request messages both in a predefined manner and only when a service is possible, unsynchronized clients are characterized by spontaneous and uncontrolled dispatch of request messages. For example, an unsynchronized client of a vending machine may try to extract more than one item after inserting coins only once.

The first goal of this work is to provide language support for coordinating object-groups by realizing the **Coordinating Environments model** (CEs model) of coordination. The model uses a small number of coordinating mechanisms and can be integrated transparently with an imperative, concurrent object-oriented programming (COOP) environment.

The two prime properties of the model are **separate encapsulation of coordination** and **non-intrusion**. The first property means that coordination is specified and managed independent of each coordinated object. A coordinated object engages only in those communications which are necessary for providing its service to the client; a separate agent manages all communications necessary for achieving group coherence. The second property means that component objects maintain their independence and encapsulation in the presence of the coordination. It will be argued that the CEs model achieves these two properties more completely than other coordination mechanisms.

The second goal of this work is to develop formal abstractions for coordinating autonomous agents. More specifically, the feasibility of coordinating process agents defined in the Calculus of Communicating Systems (CCS) [1] is studied by introducing a special formal agent called a **Coordinating Environment agent** (CE agent) and studying its formal properties. The resulting calculus, termed the **Calculus of Coordinating Environments** (CCE), is considered as the first step towards a more expressive calculus which will provide an integrated approach for specifying concurrency, communication, and coordination.

In this chapter, section 2 discusses related work and tries to further motivate the first goal discussed above. The motivation behind the second goal is discussed in Chapter Four.

1.2 Motivation and Related Work

The architecture of a coordination model is one of **centralized**, **decentralized**, or **hybrid**. The major difference between the first two categories is that *centralized* models coordinate through a central agent which bears the complete responsibility of coordinating components whereas in *decentralized* models that responsibility is distributed among the components themselves. In *hybrid* models the responsibility is shared between the components and a central agent. Within each category, some models **describe** coordination for the purpose of design and verification [2, 3, 4, 5] while others **prescribe** coordination using language primitives that enforce coordination at runtime [6, 7, 8, 9, 10, 11, 12].

In the following sections, selected models in each category of architecture are discussed followed by a critical analysis of the properties of coordinators in that category.

1.2.1 Centralized Coordination Models

The salient feature of centralized coordination models is that components are coordinated using a central *group coordinator* agent. Group coordinators are used in *ActorSpaces* [6], *Composite Multimedia Objects* [8], *Interface Groups* [9], *Arapis' Roles* [4], and *Subsystems* [5]. Of these, the three latter proposals are discussed in the following sections.

1.2.1.1 Interface Groups

The *interface group* abstraction is meant for distributed computing based on the client-server model of computation. The abstraction is superimposed on the ANSA [13] model of object-based computation. The prime components of the ANSA model are *objects* that encapsulate state and provide services, *interfaces* which are associated with objects, each interface representing a single service, and *operations* which are members of an interface and realize the service provided by the interface.

An *interface group* hides a collection of interfaces by providing a single interface, the *group interface*, which represents all the interfaces in the group. Invoking an operation in the group interface causes the same invocation to be broadcast to all the members of the interface group. The results yielded by the members are collated into a single result by the interface group and returned to the client. The motivations behind forming such a group are the following:

- to enable multi-endpoint communications between clients and servers to be replaced by single-endpoint communications; the client can invoke a single operation in the group interface and thereby invoke a large number of operations in several similar interfaces,
- to hide the number of identical servers in a group from the clients,
- to hide the distribution of the servers in a network from the clients,
- to exploit parallelism by dividing tasks into parallel activities,
- to support reliable servers by replicating the service provided by a group, and

- to provide continued service availability by recovering from partial failures.

The following points must be noted about the *interface groups* abstraction.

- The prime motivation is to hide multi-endpoint communication and not to separate communication between objects on the basis of computation and coordination.
- A group consists of homogeneous interfaces only. Thus, the model is weak with respect to addressing the needs of groups which consists of objects with different interfaces.
- The parallelism that is exploited in *interface groups* is based on data decomposition; each object in a group executes the same operation(s) on different input data supplied by a client. Another possible type of parallelism that can be exploited in object groups relies on a functional decomposition of the computation in a group; each object executes a different operation in parallel but the group as a whole achieves some common goal. *Interface groups* are weak in exploiting this form of parallelism.

1.2.1.2 Arapis' Roles

In [4] object groups are defined using the notion of *roles*. A *role* is a composition of objects which realize a well-defined unit of functionality and has the following features:

- it acts like a central coordinator which intercepts all the messages meant for its components,
- it provides its own interface using which external objects access the components, and
- it disallows any direct communication among components — all communication among components is routed through the *role object*.

The order of processing the messages is specified as a set of temporal constraints in a *role*. The constraints are separated into public and private parts. Public constraints are defined using the public interface of the *role object*. Private constraints are defined using the interfaces of the component objects of the *role*.

The following points must be noted about *roles*.

- The scheme is primarily descriptive and meant to verify the correctness of programs using a temporal logic framework.
- In its current form, *roles* use propositional temporal logic to specify the constraints which prevents it from specifying constraints based on the arguments passed in messages.
- *Roles* do not endeavor to separate the communications among components on the basis of computation and coordination — both kinds of communications are used uniformly in specifying the temporal constraints.

1.2.1.3 Subsystems

A *subsystem* [5] is defined as groups of classes and/or other *subsystems* which collaborate among themselves. Some advantages of *subsystems* are that they provide a clearly delimited unit of functionality and they simplify patterns of communication between objects. These advantages render programs easier to understand and modify.

The simplification of the patterns of communication is achieved in a *subsystem* by defining a group coordinator that intercepts the request messages and distributes them to the components. The main drawback of the scheme is that the group coordinator, before delegating a request to a component, must make sure that the component will be able to process it. For example, if a data-buffer object is a component of a group and the group coordinator delegates a *GetItem* message to the buffer, then the group coordinator must be certain that there are items in the buffer. Thus, the group coordinator becomes involved in decisions which should be made locally in components, thereby diminishing the *separation of concerns* that could be achieved between the group coordinator and the components.

1.2.1.4 Centralized Coordination Models: A Discussion

In centralized coordination models a coordinator object assumes the responsibilities of operation scheduling and state propagation thus decoupling components and making them more reusable. But in doing so, components of a group are hidden from the clients. Thus, clients, instead of interacting with individual components, interact with a central coordinator object. Hiding smaller parts of a part-whole hierarchy is a useful mechanism for controlling complexity in procedural hierarchies but such a scheme conflicts with the basic motivation behind the concurrent object-oriented programming (COOP) paradigm: computation through point-to-point, direct communication among software counterparts of real-world entities. For example, to depress a button in a panel of buttons, a client depresses the button directly; to extract a candy from a vending machine, a client deposits coins directly to the coin acceptor and extracts a candy directly from a slot; to open or close the doors of an elevator car, a client interacts directly with the car. Thus, in many situations, projecting a single, monolithic interface of the group coordinator which hides the interfaces of the components of the group conflicts with the expectations of clients and introduces an artificial element in the interactions between clients and servers.

Another problem with the centralized paradigm is the following. The interface of a centralized coordinator duplicates (and, in some cases, alters) the interfaces of the coordinated components (since the coordinator must intercept every message sent to a group). The resulting interface is large and lacks structure (the coordinator does not serve a single purpose but serves the purpose of all the components combined). Also, the interface may contain several methods which are unrelated to the coordination responsibility of the coordinator. To impart some structure to the interface of a central coordinator, some proposals restrict a client's view of the interface. But such view control is difficult to achieve especially when the same coordinator must project different views to different sets of clients. For example, in an elevator system, a group coordinator for an elevator car must project a view to the clients who are inside the car that is different from the view that must be projected to the clients who are on the floors.

Other than hiding the components, two problems of the centralized paradigm manifest themselves in the vending machine example (Figure 1.2) as follows. First, since the coordi-

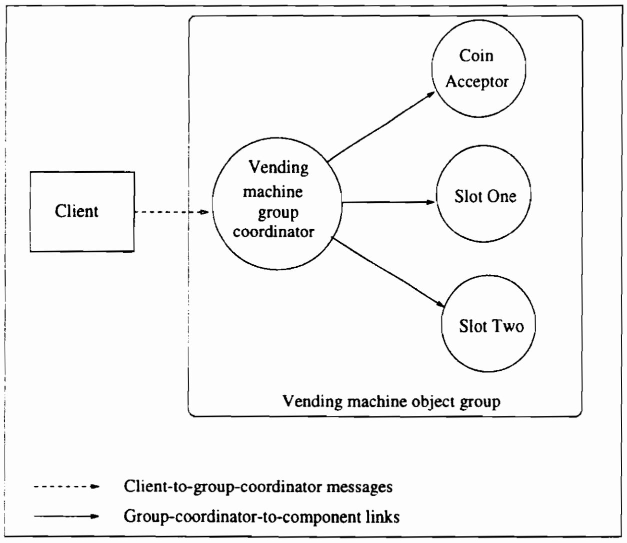
Coin Acceptor
Vending machine group coordinator
Slot One
Slot Two
Client
Vending machine object group
Client-to-group-coordinator messages
Group-coordinator-to-component links

Figure 1.1: Coordinating the components of a vending machine using a group coordinator.

nator intercepts all messages intended for components, it handles messages unrelated to the coordination of the group. For example, request messages to a slot inquiring the price of an item play no role in the coordination of the group. But, the coordinator must handle that message because the group's client should not bear the unreasonable burden of determining which messages to send to the coordinator and which directly to the object. Even if this burden was borne by the client, the coordinator could not operate in an assured manner due to the possibility of a mistake on the client's part. Second, the group coordinator changes the interface projected by slot objects due to programming language considerations. For example, consider the *Price* method of a slot object. The designer of the slot object might publish a description of this method as simply "Price()". However, because there are two slots in the vending machine, the group coordinator's interface would present this interface as either "Price(int SlotId)", where an additional disambiguating parameter is added, or as "PriceSlot1()" where the name of the method is changed. This implies that the same object in different groups may have different interfaces as seen by a client. Such changes are undesirable because it diminishes the value of the concept of "the interface of an object" as this interface must always be understood in the context of a particular group.

1.2.2 Decentralized Coordination Models

The salient feature of decentralized coordination models is that components communicate explicitly with each other to maintain group coherence. Thus, any object-oriented programming language may be used to realize this form of coordination by defining objects that are aware of each other and which include the necessary communications for operation scheduling and state propagation. Among the proposals that use the decentralized paradigm [11, 7, 2, 3, 12], *Contracts* [2, 3] is the most sophisticated. It endeavors to formalize the communication requirements of components and is discussed next.

1.2.2.1 Contracts

A *contract* **specifies behavioral compositions** — "groups of interdependent objects cooperating to accomplish tasks". A contract defines a behavioral composition by specifying the following:

- The participants of the behavioral composition.
- The following contractual obligations of the participants.
  - Type obligations — each participant must support certain variables and a certain external interface.
  - Causal obligations — each participant must perform an ordered sequence of actions and make certain conditions true in response to its input messages.
- Invariants that participants cooperate to maintain.
- Preconditions on participants to establish the contract and the operations which instantiate the contract.

Two other features of a contract are *contract refinement* and *contract inclusion.* The former allows the specialization of contractual obligations and invariants of a predefined contract and the latter allows a contract to be composed of simpler contracts.

A contract is defined independently of classes that implement a program in an object-oriented language. Class implementations are mapped to participant specifications using *conformance declarations.* A conformance declaration is a specification of how a class supports the role of a participant. It describes the variables and methods the class provides to the role.

1.2.2.2 Decentralized Coordination Models: A Discussion

In object-based systems, explicit communication for the purpose of coordination leads to the following problems. First, coordination is an issue at the time of designing the participating objects, thereby complicating their design. Second, since coordination aspects must be considered at design time and included in the implementation of an object, the reusability of an object in different object groups is limited by the foresight of the designer in considering different group scenarios. Third, coordination partners must be recorded explicitly in the state of an object. This binds an object to a specific group situation, further reducing its reuse potential.

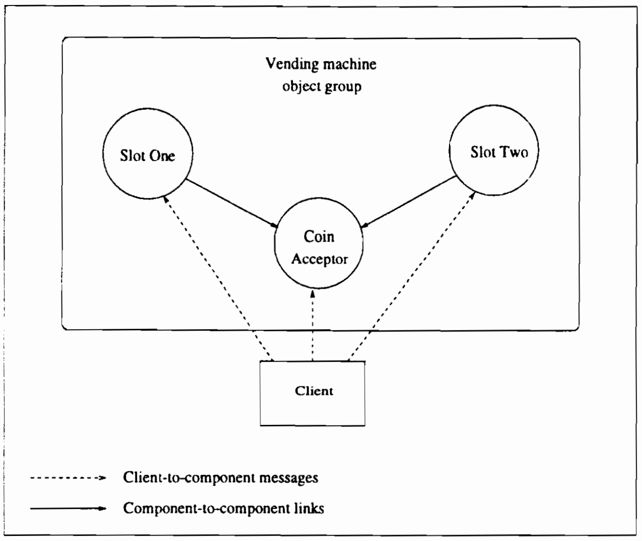
Vending machine object group
Slot One
Slot Two
Coin Acceptor
Client
Client-to-component messages
Component-to-component links

Figure 1.2: Coordinating the components of a vending machine through explicit communication.

The problems with explicit communication manifest themselves in the vending machine example as follows (Figure 1.1). When a client wants to extract a candy from a slot, the following must be done by the slot: a critical section must be entered, the inserted amount must be obtained from the coin acceptor, if the inserted amount is sufficient, a candy must be dispensed and the coin acceptor must be instructed to refund the excess amount inserted, and the critical section must be exited. The reason for using a critical section is to ensure that while a slot is deciding whether to dispense an item, no other slots is allowed access to the deposited amount (thereby guarding against unsynchronized clients who may try to open more than one slot). As a result, a slot must store a pointer to the coin acceptor component, it must be aware of the interface of the coin acceptor, it must invoke its methods, and it must explicitly manage the entry to and exit from a critical section. Thus, the possibility of implementing a slot as a general purpose, reusable object which is unaware of any other objects and which only participates in dispensing candies is thwarted by entangling coordination decisions with the functionality of a slot.

1.2.3 Hybrid Coordination Models

The salient feature of a hybrid coordination model is that coordination is achieved through the cooperation of both the components in a group and a *transparent* group coordinator. Clients are not aware of the presence of the group coordinator and do not communicate with it (thereby making it transparent). Also, components share part of the operation scheduling responsibility of the group coordinator but do not engage in explicit communication with each other for coordinating purposes. Thus, group coordinators in these models achieve a better *separation of concerns* by concentrating solely on coordination issues. Two models in the hybrid category, which are similar to the model developed in this dissertation, are *Synchronizers* [10], and *Abstract Communication Types* (ACT) [41].

1.2.3.1 Synchronizers

A *synchronizer* attempts to integrate a coordination model with the Actor model [15] of computation. In the Actor model, agents called *actors*, which possess their own threads of control, compute by sending request messages to other actors. A mail queue in each actor buffers incoming request messages. The address of the mail queue serves as the unique identity of an actor. Program segments called *behaviors* in an actor process request messages. An actor has only one behavior associated with it at any point in time called its *current behavior*. The current behavior processes only one request message and specifies a *replacement behavior* for processing the next request message. The current behavior defines a subset of an actor's methods as its **current interface**. A request message is processed only if the corresponding method appears in the current interface of the object (referred to as **message acceptance**). Otherwise, the request message remains buffered to be processed by a subsequent behavior. Special procedures are introduced which determine the replacement behavior (or **replacement interface**) of an object based on the current values of some of its instance variables.

A synchronizer extends the message scheduling scheme for single actors to multiple actors. A synchronizer has references to all the actors it coordinates. Each component actor is assumed to have its own local constraints which determine its local interfaces. On **observing** the acceptance of a message by an actor, the synchronizer tries to satisfy the group constraints. Constraint satisfaction in synchronizers is realized through pattern-matching using a *declarative* notation. If no patterns match, the actor executes the message. If any pattern matches then either local state variables in the synchronizer, if any, are updated before the message executes, and/or the message is delayed, or the synchronizer deletes itself.

1.2.3.2 Abstract Communication Types

An *abstract communication type* (ACT) is a first-class object which abstracts interactions among objects. The ACT model is based on the *composition filters* object model. An object in the composition filters model contains an interface part and an implementation part. The salient feature of the interface part is that it handles both incoming messages (that is, messages sent to the object) and outgoing messages (that is, messages sent by the object) using *input filters* and *output filters*, respectively. A *filter describes* conditions which must be met for incoming messages to be accepted and/or delegated to other external objects. A filter may also operate on a message thereby replacing parts of the message. As noted in [41], without filters, the object model is very similar to conventional object models.

An ACT object extends the message filtering behavior of a single object to multiple objects. An object participating in a group forwards an accepted message to an ACT object by using a special filter called *Meta*. The ACT object may extract data from the message arguments or may modify the contents of the message before sending it back to the component. An ACT object can also take actions on component objects, use filters to restrict or modify its own message processing behavior, and use other ACT objects to realize certain aspects of its coordinating behavior.

1.2.3.3 Hybrid Coordination Models: A Discussion

Compared to decentralized and centralized coordination models, the hybrid coordination models achieve a higher degree of reuse of components and achieve a higher degree of separation of concerns. As a result, models in this category are promising candidates for coordinating object groups.

Although successful in handling group coordination issues, both the synchronizers and the ACT models leave scope for improvement. A few noteworthy issues which, if addressed, will yield a better coordination model are as follows. First, constraint satisfaction in the synchronizers model is realized through pattern-matching in an unstructured collection of rules specified using a declarative notation. Similarly, the filters in the ACT model are ordered collections of constraint descriptions which are tried sequentially until a match occurs. In both models, the lack of structure may interfere with ease of understanding when the complexity of constraints increases. Second, synchronizers observe only the acceptance of request messages by components but do not observe their termination. Similarly, in the ACT model, the termination of a method is not treated as a significant event which must be observed by an ACT object. Observing termination is important because, in an asynchronous environment, message acceptance does not guarantee immediate execution of the requested method. In order to coordinate correctly, a coordinating agent must ensure that the updating of instance variables by the requested method is over. Third, synchronizers maintain instance variables that logically belong to components thereby violating their encapsulation. For example, in the vending machine synchronizer, the amount inserted by the customer and the price of each item dispensed is stored in the synchronizer itself although those items should be maintained by the coin acceptor and the slots, respectively. Fourth, in the ACT model, objects participating in a group explicitly store pointers to ACT objects, they are aware of the types of ACT objects, and they specify filters which explicitly forward received messages to ACT objects. Such explicit inclusion of the details of a coordinator object interferes with the reusability of a component by tying it down to a specific group.

The CEs coordination model is proposed in this dissertation which is a new hybrid coordination model, as shown in the classification of coordination models in Figure 1.3.

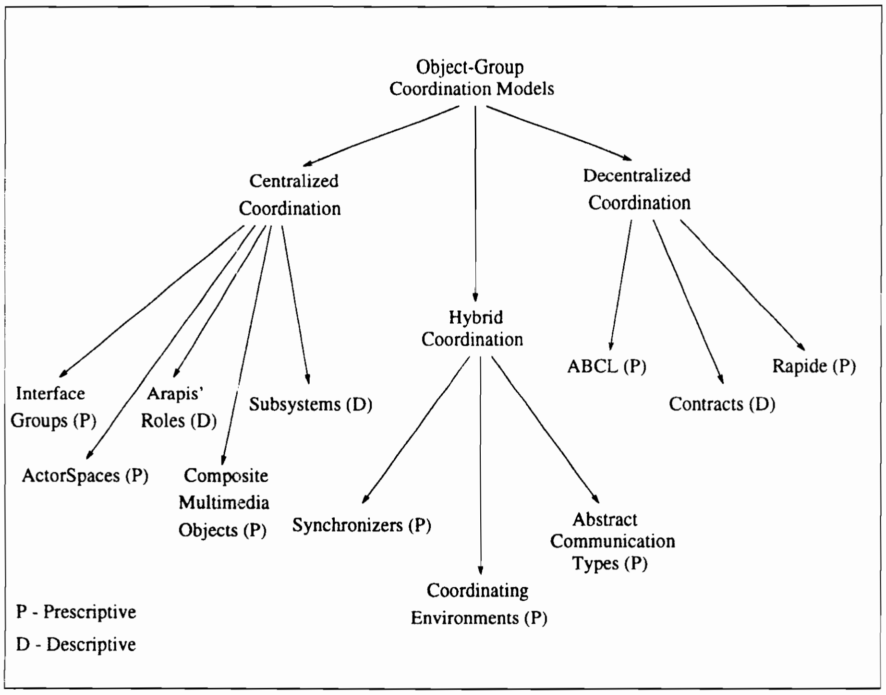

Figure 1.3: A classification of object-group coordination models.

While the CEs model has several similarities with both the synchronizer and the ACT models, there are several major differences. A detailed comparison of the features of the latter models with those of the CEs model is presented in Chapter 6.

Chapter 2

The Coordinating Environments Coordination Model

2.1 The Object Model

The object-group model proposed in this dissertation is based on an object model that uses a three-step method invocation procedure when a **client object** invokes a public method of a **server object**. The first step in invoking a method is to construct a special object called a **request message** which stores a reference of the method being invoked (referred to as the **requested method**) and a list of the actual argument values. In the second step, the request message is sent to the server object. In the third step, the server object, after some local decision making, either accepts the request message for execution or rejects it. If the message is rejected, it is returned to the client object. If the server accepts the request message, the client has the option of either executing the requested method using its own thread of control or requesting the server to execute it using a new thread of control. In either case, any values which must be returned to the client are returned using a special object called a **Cbox** which a client supplies to a server in a request message. The server fills a Cbox with the return values and the client extracts them explicitly. A client blocks on a Cbox until return values are supplied.

There are three advantages of using a three-step method-invocation procedure. First, a server object may apply a selection criterion to each request message it receives and make a decision as to whether to process it or reject it based on its local state. The local state of an object is the collection of the values stored in all its instance variables at any point in time. Note that although the values of some instance variables may not play a role in the decision-making process, the local state consists of all instance variables of an object. For certain local states, an object may not be able to execute one or more of its public operations. The message-selection ability prevents the invocation of an unavailable method by a client. The second advantage is that clients and servers may execute in parallel thereby increasing the potential for exploiting the inherent concurrency in an application. The third advantage is that well-defined entry and exit points are introduced in the message-processing cycle of a server object which may be utilized to realize an "observer" object which coordinates the activities of a collection of such server objects. The latter feature is the basis for proposing the object-group model in section 2.2.

The usual, synchronous method invocation procedure used in extant object-oriented programming languages (OOPLs) is also available in the object model and is used to realize message sending, as discussed in the following section. Synchronous invocation is also used by an object to execute local methods.

The object model replaces the master-slave relationship among clients and servers by a more equitable relationship. As a result, objects in this model are called **autonomous** objects since they can exercise a better control over their service-providing ability. Autonomous objects are very similar to, but not exactly the same as, *actors* proposed by the *Actor model of computation* [15]. The behavior of autonomous objects is discussed in detail in the next section.

2.1.1 Autonomous Objects

The request-message-processing cycle of an autonomous object is shown in Figure 2.1. An object has a **public interface** (PI) which is visible to a *client.* The PI is determined by the name, return type, and types of the arguments of the public *methods* of an autonomous object.

When a client wants some service from a server object, it **synchronously** (that is, not using a request message) invokes a special method of the server (denoted by Arc 1 in Figure 2.1), which will be referred to as the **Request Handler** (RH), and supplies the request message as an argument to it. The RH applies a selection criterion to the request message before executing it.

The RH uses the **current public interface** (CPI) to determine whether to execute

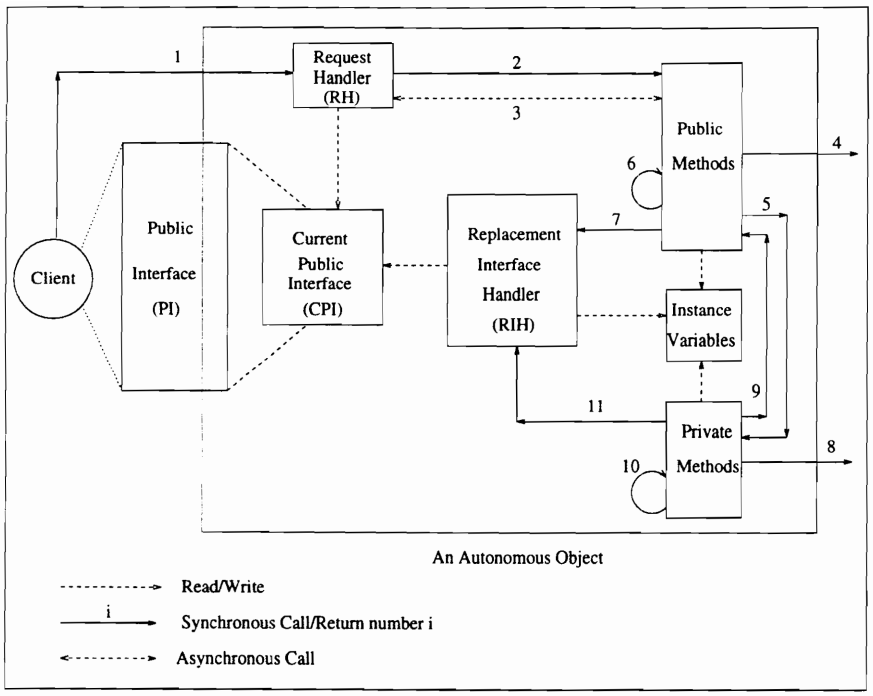

Figure 2.1: The request-message-processing cycle of an autonomous object.

a request message. The CPI is a subset of the PI which stores only representations of the method names which the object may execute in its current local state and contains no information regarding the return types and the types of arguments accepted by the methods. The very first CPI is installed from the constructor of an object. Once installed, a CPI remains valid until a method is scheduled for execution. Once a method is scheduled, the CPI is invalidated. Before terminating, the executing method is expected to install a new CPI. If it does not, the last CPI is marked as valid after the method terminates. Due to the sequential installation of CPIs in an autonomous object, request messages are processed in a **strictly sequential order**.

As the first step of processing a request message, the RH checks whether the name of the method in the request message appears in the CPI. If it does, the RH *accepts* the message for execution (referred to as **message acceptance**). If it does not, then the message cannot be processed in the current local state of the server and the RH terminates with a return value denoting message rejection. On rejecting a request message, a client returns to its own context along with the request message.

An alternative to returning the client would have been to buffer the request message in the server object and to process it later when the CPI of the server permits. Such a scheme would implicitly guarantee the eventual processing of a request message by a server object and is suitable for application domains in which client objects are sufficiently synchronized with application objects so as to make this guarantee reasonable and the message-processing decisions of server objects are based solely on their local states. But, in application domains (e.g., transaction processing) in which a client may be unsynchronized and server objects are part of larger collections of objects, buffering of request messages introduces unnecessary complications in the message processing activities of a server object. As an example application of the latter domain, consider a vending machine which is composed of two types of server objects: a coin acceptor and a collection of slots. A client may try to open multiple slots before inserting any coins in which case the slot objects must ignore such requests. In this case, the constraint that is violated is that coins must be inserted before extracting an item and the decision of dispensing an item by a slot is affected by the presence of the coin acceptor. If a slot buffers requests, then, when coins are inserted, it must determine which message has been sent after coin insertion and delete any earlier messages. The latter requires the existence of some message-arrival-disambiguation procedure. Also, the deletion of request messages raises the issue of how to avoid a possible deadlock situation which arises when a client blocks expecting a response from a server object and the server does not send one. Hence, to avoid introducing unnecessary complications and to enable the composition of server objects into object groups, server objects do not buffer request messages in this object model.

The non-buffering property of server objects gives rise to two modes of sending request messages. In the first, referred to as the **only-once** mode, a request message may be sent only once and control returns to the client if the message is not accepted. The client, in this case, must decide what to do about the failed request. In the second, referred to as the **until-accepted** mode, a request message is sent repeatedly until it is accepted by a server object. In this case, the client blocks until the message is accepted. Although easier to handle, the latter case may give rise to a deadlock situation if the server object never accepts the request message.

After a message is accepted by the RH, depending on the type of the request message, the requested method is either invoked using the thread of control of the client (Arc 2) or a new thread of control is spawned to execute the method (Arc 3). The latter invocation will be referred to as an **asynchronous call**. If an asynchronous call is not used, the client's thread of control returns only after the requested method terminates. If an asynchronous call is used, the client's thread returns after spawning the new thread of control.

The two modes of sending request messages along with the two ways of executing a requested method (either using the thread of control of a client or with a new thread of control) gives rise to the four following ways in which method executions may be achieved in the object model: **only-once-same-thread**, **only-once-new-thread**, **until-accepted-same-thread**, and **until-accepted-new-thread**.

In order to prevent simultaneous execution of methods in a server object, the RH processes request messages in a critical section. The CPI is accessed from the RH after entering the critical section. If the client's thread is used to execute the requested method then the critical section is exited only after the method terminates thereby preventing the processing of any new request. If an asynchronous call is made to execute the requested method, then, although the client's thread is not blocked, the critical section is unlocked only after the requested method terminates. Thus, even if multiple clients invoke the RH simultaneously, it processes requests sequentially.

The events which may take place while a method is executing are captured in Figure 2.1 by the arcs labeled 4 through 11. Note that the sequential numbering of the actions is purely for reference and does not imply any ordering among them.

While executing, a public method may send request messages to objects (Arc 4) which are either already known to the object in which it executes, referred to as its **acquaintances**, or are created by it dynamically. Note that a request message may **not** be sent to the same object in which the method is executing because an object processes request messages sequentially. Also, a calling sequence that results in a cycle among objects must be avoided for the same reason. A public method may synchronously invoke either other local public methods (Arc 6) or local private methods (Arc 5).

Private methods behave similarly to public methods in that they can send request messages (Arc 8) and synchronously invoke other local private methods (Arc 10) or local public (Arc 9) methods. In both private and public methods, the local state of an object may be read or written to.

A salient feature of the object model is the ability of some or all of both public and private methods to install a new CPI on an object. To do so, both public and private methods use a special private method of an autonomous object, referred to as a **replacement interface handler** (RIH). A method, public or private, may synchronously invoke the RIH *at most once* (Arc 7 or Arc 11) only after the method is certain that it will not change the instance variables of the object anymore. The RIH reads the values of (probably, a subset) of the instance variables and installs a new CPI based on those values. Note that in a sequence of synchronous method invocations, the RIH may be invoked multiple times but the last invocation of the RIH, before the requested method terminates, installs the final CPI for the object.

2.2 An Object-Group Coordination Model

When a collection of autonomous objects compete or cooperate to achieve a common task or goal, they are collectively referred to as an **object group** and each object in the group is referred to as a component. A client requests service from an object group by explicitly communicating with one or more of its components. Processing of client requests depends on the **CPI of the group** which is the union of the CPIs of some or all the components. A **group coordinator** is an entity which coordinates a group of autonomous, independently conceived components by taking actions necessary to ensure that there is no conflict between the CPIs of the components and the CPI of the group. Such an entity allows a better *separation of concerns* by enabling components to concentrate solely on their primary role of providing service.

The **Coordinating Environments** model (CEs model) of coordination for object groups enables the design of group coordinators, referred to as **CE objects**, that coordinate in a **non-intrusive** manner. Non-intrusion means that a group coordinator **transparently observes** key events in components and takes **coordinating actions** which **transparently alter** the behavior of components. Event observation is considered transparent if the designer of an observed component does not have to provide additional functionality in the component for event observation and if the coordinator does not have to know the structure of data contained in that component's messages. Coordinating actions are considered transparent if the designer of a coordinated component does not have to provide explicit mechanisms for accepting and responding to coordinating actions.

2.2.1 Coordinating Behaviors

The CPI of a group and the coordinating actions which ensue when a message in that CPI is accepted are defined by CE objects in modular and executable entities called *Coordinating Behaviors* (CBs). At any point in time there is exactly one CB, called the **current CB**, executing in a CE object. When the group makes a transition from one CPI to another, the CE object transits from the current CB to a **replacement CB**.

The advantage of modeling a CE object as a collection of CBs is that each CB provides a structured representation of the coordinating activities that can occur for the current CPI of the group. This structuring aids in designing and reasoning about the coordination provided by a CE object.

2.2.2 Observed Events

For each method that is in the CPI of a group, a CB observes the following events:

- The acceptance of the corresponding request message by a component (an **acceptance event**).
- The termination of that method (a **termination event**).

The observation of both the above events are blocking operations since a CB does not know when they occur in a component. When one of the above events occurs in a component, the RH in that component engages in explicit communications with a CE object to notify it. Note that the request handlers are modified (as shown later) to engage in such communications.

Observation of an acceptance event informs a CB that a method in the CPI of the group has been requested and so it must prepare to take the necessary coordinating actions. Since a component is unaware of the current CPI of a group, it reports every acceptance event to a CE object. Out of all the acceptance events reported to a CE object, a CB observes one which requests a method that is in the CPI of the group.

Observing the termination of a method informs a CB that the CPI of the component in which the method executed is finalized and hence it must take steps to finalize the CPI of the group. This event also enables a CB to keep track of and to control the number of concurrently executing components in a group.

A termination event can only be observed if the acceptance event which initiated the method is also observed. This requirement is necessary for the following reason. Since observing method-termination is a blocking operation, if a CB blocks in order to observe the termination of a specific method that was never scheduled for execution (because a request message was never accepted by the component), then the CB deadlocks. Since the CB keeps no record of the corresponding message-acceptance event, it can never recover from this deadlock situation itself.

Acceptance and termination events are observed in the order they are reported to a CE object. Since many such events may occur in parallel, a CE object may receive multiple notifications simultaneously. Thus, a CE object buffers event notifications and processes them in a first-come-first-served (FCFS) basis.

2.2.3 Coordinating Actions

The coordinating actions which a CB may take can be one or more of the following:

- **Ignore** — Explicitly mark one or more acceptance events in a component as unobserved.
- **Schedule** — Schedule the method in an observed acceptance event for execution.
- **Await termination** — Schedule the method in an observed acceptance event for execution and wait for its termination.
- **Block** — Block request messages from being accepted.
- **Unblock** — Unblock blocked messages.
- **Add arguments** — Update a request message by appending new argument values to it.

- **Become** — Specify a replacement CB.
- **Call** — Synchronously invoke both public and private methods in components.
- **Send** — Asynchronously invoke public methods in components.

2.2.3.1 The Ignore Operation

The *ignore* operation enables a CE object to divide the set of acceptance events occurring in all components into two sets: a set which is relevant to the coordination of the group and a set which is not relevant to the coordination of the group. The availability of the ignored events in the CPI of the group is controlled only by the components which accept those messages and not by the CE object. Thus, such events are accepted and executed without any intervention from the CE object thereby speeding up the message-processing cycle of components. Unobserved messages remain unobserved throughout the lifetime of a CE object. Note that if all acceptance events in a component are irrelevant to the coordination of a group, then all such events are unobserved. In order to prevent the explicit marking of all public messages as unobserved, such components do not **register the CE object** (refer to page 31 for more on registering CE objects) thereby allowing them not to post any acceptance events.

2.2.3.2 The Schedule and Await-Termination Operations

The *schedule* operation involves informing a RH that it may proceed with the execution of the method. This action enables a CB to execute in parallel with the scheduled method.

The *await-termination* operation is useful when no coordinating actions must be taken between the scheduling action and the observation of the termination event. In such a case, instead of separately specifying the scheduling action and the observation of the termination event, this action allows for a concise expression.

2.2.3.3 The Block and Unblock Operations

The *block* operation enables a CB to update the CPIs of one or more components so that they become consistent with the desired CPI of the group. As a result of this action, the RH "un-accepts" the blocked message, if it had accepted one, and causes the client to withdraw the request. The component is also marked so that the method in the blocked message is not processed until it is unblocked.

A CB unblocks a blocked message using the *unblock* operation when it determines that reintroducing the method is consistent with the behavior of the group. As a result of this operation, the component is marked to allow the processing of the blocked message.

2.2.3.4 The Add-Arguments Operation

The *add arguments* operation appends new argument values to a request message. After observing an acceptance event and before scheduling it for execution, a CB may append new argument values to the list of arguments which exists in a request message. One possible application of the operation is to supply default argument values. A client would invoke a method with the required arguments and a CB would provide the default argument values by collecting them either from other components in the group or from information maintained by the CE object itself. Thus, the operation may serve as a means of propagating current, local, state information from one component to another or from the group (represented by a CE object) to a component.

2.2.3.5 The Become Operation

The *become* operation enables a CB to control the time at which the transition to the replacement CB is made thereby finalizing the installation of a new CPI of the group. The current CB must specify only a single replacement CB since there must be a unique CPI of the group at any point in time. Moreover, since a CE object allows only a single CB to be active at any point in time, the current CB must specify a replacement CB just before it terminates. Note that the initial CB of a CE object is executed as a result of an explicit external invocation.

2.2.3.6 The Call Operation

A CB, as the coordinator of a group, may synchronously invoke public methods in a component. Synchronous invocation of public methods is resorted to in order to prioritize the coordinating actions of a CB by enabling it to bypass the RH in a component and to force the component to execute the CB's request before those of any clients. Moreover, bypassing the RH allows a CE object to invoke public methods which are either temporarily or permanently blocked from the CPI of a component.

Bypassing the RH in a component to make a synchronous invocation is complicated by the fact that the RH may be engaged in one of the following activities when a CB requests a method execution:

- Executing a method whose acceptance is unobserved by the CE object.
- In the process of deciding whether to accept a request message.
- Waiting for a scheduling action from the current CB after accepting a request message.

Therefore, as a result of a synchronous method invocation by a CB, the following actions must be taken on the RH:

- If the component is executing a method, the CB must wait until it terminates, then temporarily block the RH, execute the method, and finally unblock the RH.
- If the RH is in the process of deciding whether to accept a request message, it must be temporarily blocked when it finishes making its decision, its decision about the request message must be revoked, the method requested by the CB must be executed, and finally the RH must be unblocked.
- If the RH is waiting for a scheduling action from the current CB, it must be made to withdraw the accepted request message and return the client's thread of control. Then the RH must be blocked, the method requested by the CB must be executed, and finally the RH must be unblocked. The reason the RH must be made to withdraw a request message is because after the CB executes the method, the CPI of the component may change and so the client must retry the message against the new CPI.

2.2.3.7 The Send Operation

The *send* operation is used by a CB to asynchronously invoke a public method in a component either when it is not sure about the availability of the method in the CPI of the component or when it does not want its request to have priority over other requests from the clients of the component.

2.2.4 Structure of a CE Object

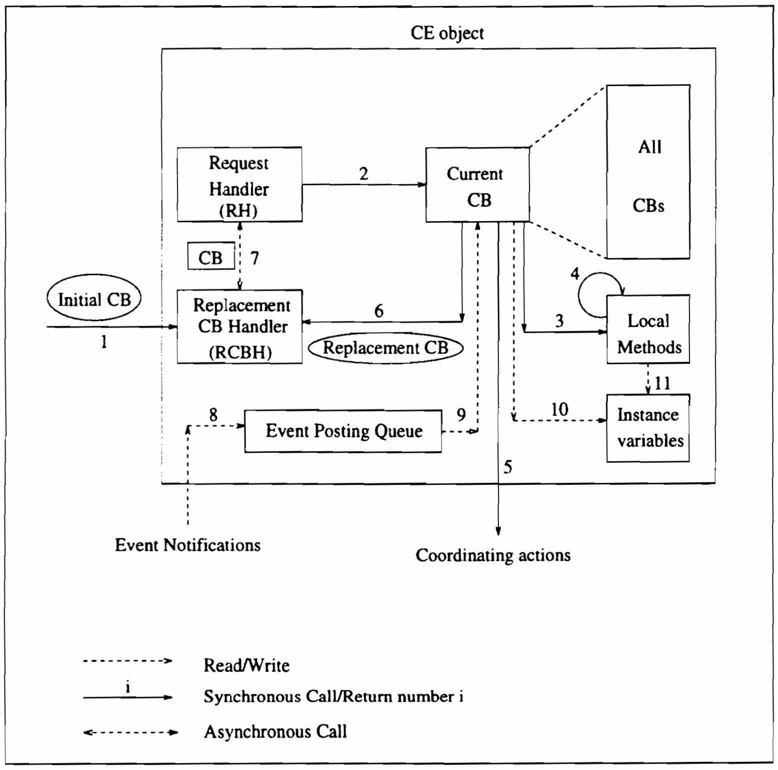

Figure 2.2: The CB-processing cycle of a CE object.

The structure of a CE object and its CB-processing cycle is shown in Figure 2.2. A CE object has local methods which may be invoked either by the current CB (Arc 3 in Figure 2.2) or by other local methods (Arc 4). Both the current CB and the local methods may read and/or write the instance variables of a CE object (Arcs 10 and 11). Components signal the occurrence of events by enqueueing notifications in the *event posting queue* of a CE object (Arc 8). The current CB tries to read and match desired event notifications from the queue (Arc 9) and, on making a successful match, takes coordinating actions on the components (Arc 5).

The initial and replacement CBs of a CE object are managed by a special method of a CE object referred to as the *Replacement CB Handler* (RCBH). The initial CB is established by executing the RCBH and passing the initial CB as an argument to it (Arc 1). The RCBH spawns a new thread of control (that is, makes an *asynchronous call*) to execute another special method of a CE object (Arc 7) referred to as the *Request Handler* (RH). The initial CB is passed as an argument to the RH. The RH executes a CB using its own thread of control (Arc 2). Before terminating, the current CB synchronously invokes the RCBH and passes the replacement CB as an argument to it (Arc 6) thereby ensuring that the replacement CB will execute using a new thread of control. The concurrent execution of both the current and replacement CBs is prevented by ensuring that at any point in time, only one invocation of the RH is active in a CE object. As a result, the asynchronous invocation of RH from the current CB is allowed to proceed only after the RH executing the current CB terminates.

2.3 Enhancing the Object Model to Allow Observation

To enable CBs to observe events in components, the object model is enhanced. Figure 2.3 shows the required enhancements. A new item which is added to the structure of an autonomous object is a list of unobserved and blocked acceptance events. Since every acceptance event is observed by a CE object by default, only those which are not observed are recorded by an autonomous object (since this list has the potential of being smaller in length than the list of observed acceptance events). Moreover, this list can be easily installed by the *Ignore* coordinating action of a CE object. The blocked acceptance events are added to the list as a result of the *Block* coordinating action and blocked events are removed from the list as a result of the *Unblock* coordinating action. Figure 2.3 also shows all the arcs which were shown in Figure 2.1 but does not label them in order to emphasize the new arcs used for observation.

In order to enable the observation of an acceptance event, the behavior of the RH is modified. After being invoked, the RH checks whether the requested message is in the list of blocked messages (Arc A1). If so, the client is made to withdraw the request. If the message

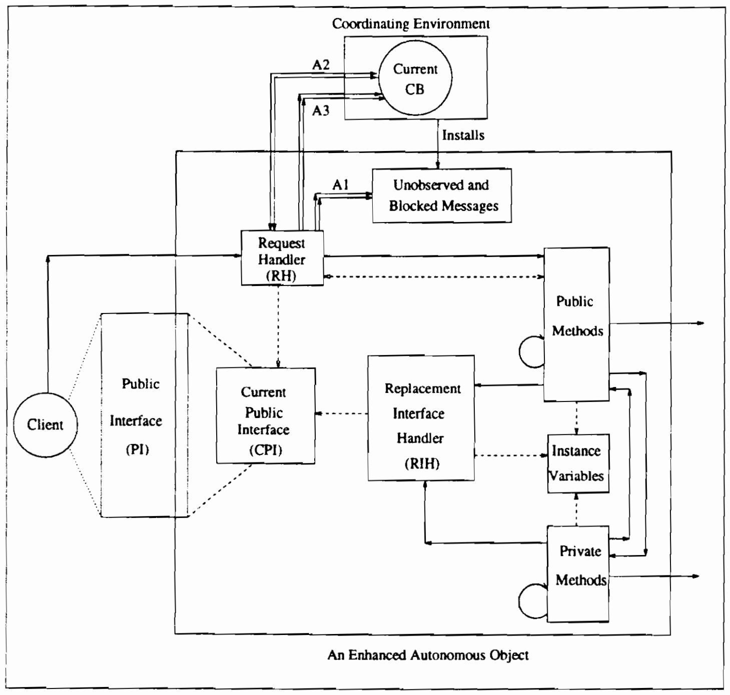

Figure 2.3: The request-message-processing cycle of an enhanced autonomous object.

is not blocked, the RH checks whether the message is in the CPI of the component. If the message is in the CPI, the RH checks whether the acceptance of that message is ignored by the CE object (Arc A1). If so, it proceeds with the execution of the requested method as before. Otherwise, it communicates with the current CB to inform it about the occurrence of the event and waits for the CB to take a coordinating (scheduling) action (Arc A2). If the CB instructs the method to be executed, the RH proceeds with the execution of the method. Thus, observing an acceptance event involves an explicit communication between the RH of an autonomous object and the current CB, but no additional functionality needs to be included in the realization of the methods thereby making the observation *transparent*.

In order to enable the observation of a termination event, the behavior of the RH is further modified. After the requested method terminates and before the next request message is processed, the RH informs the current CB about the termination event (Arc A3) only if the corresponding message acceptance event was observed by the CE object. Note that, unlike acceptance events, the RH does not wait for the CB to actually observe the termination event.

An important point to be noted is that a component records the address of only one CE object in itself. A component **registers a CE object** when it is assigned to participate in the object group which is coordinated by that CE object. Although a component may post events to only that CE object, it may participate in multiple object groups. In such a case, it may not post any events to any of the other CE objects but may execute their synchronous and/or asynchronous method invocations.

2.4 An Object-Oriented Realization of the CEs Model

In order to realize the CEs model, the following steps must be taken:

- A mechanism for enabling a CB to make event observations must be devised.
- The coordinating actions must be realized.
- A protocol between a CB and the components of a group for making observations and taking coordinating actions must be established.

In the following sections, a scheme for event observation is described, the operations which realize coordinating actions are introduced, and the protocol which CBs use to observe events and take coordinating actions is described.

2.4.1 Observing Events

A CB is informed of the occurrences of acceptance and termination events by a RH using **event messages** and a CB observes such events using **event objects**. These two types of objects are discussed in the two following sections.

2.4.1.1 Event Messages

Event messages are sent by the RH in a component to mark the occurrence of acceptance and termination events. The act of transferring an event message to a CE object is called **posting an event**. Conceptually, an event message contains a reference to the method which is requested in the accepted request message, a pointer to the component in which the event occurred, a data item, referred to as *schedule*, which, if set, indicates that the

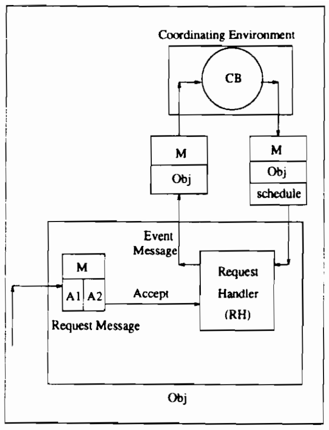

Figure 2.4: Posting an acceptance event.

request message must be accepted, and another data item, referred to as *terminate*, which, if set, indicates that the requested method has terminated.

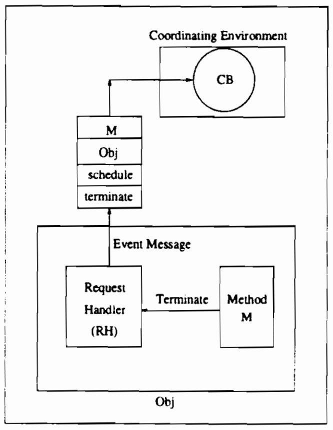

Figure 2.5: Posting a method-termination event.

Figure 2.4 shows how an acceptance event is posted. An object *Obj* receives a request message in which method *M* is requested for execution and which contains two argument values, *A1* and *A2*. After the RH accepts the request message, it posts an event message as shown in the figure and blocks. Both *schedule* and *terminate* are not shown in the posted event message. After observing the event, if the current CB wants to schedule the method (as it does in Figure 2.4), it sets *schedule* (marked by the appearance of the *schedule* slot in the event message in Figure 2.4), returns the event message to the RH, and unblocks it.

Figure 2.5 shows how a termination event is posted. After the method *M* in object *Obj* terminates, the RH determines whether the request message which caused the execution of the terminated method was scheduled by a CB (by checking whether *schedule* is set in the event message). If so, *terminate* is set and the event message is posted. Note that the RH does not block after posting this event.

2.4.1.2 Event Objects

A CE object uses a special object, called an **event object**, to detect an event posting. Event objects act as the glue which bind the computation and coordination models together. Each CE object uses a unique set of event objects which are constructed prior to constructing the CE object itself and which exist until the CE object is deleted. Moreover, all event objects used by a CE object are visible from every CB and a CB may not create any private event objects.

There are two types of event objects (EOs): **elementary** (EEO) and **composite** (CEO). EEOs can be one of two types: **single-constituent** and **multi-constituent**. Both EEOs and CEOs can be used to make observations repeatedly.

A single-constituent EEO can be used to observe acceptance and termination events related to a method in only one component whereas a multi-constituent EEO can be used to observe acceptance and termination events related to a method in more than one component. All components in a multi-constituent EEO must be instantiations of the same class so that each contains the method whose events are observed using the EEO. An EEO of either type must first observe an acceptance event, and if the requested method is scheduled for execution, it must then be used to observe the termination of that method. After this observation cycle, an EEO may be reused to observe another acceptance event.

A CEO, which is a collection of either single-constituent or multi-constituent EEOs, may be used to observe events associated with a single method in multiple components or it may be used to observe events associated with different methods in different components. In the former case, the constituent EEOs are constructed to observe events related to only one method whereas in the latter case, the EEOs are constructed to observe events related to different methods. The EEOs in a CEO are not assumed to be in any specific order. A CEO may be used to observe multiple acceptance events consecutively before observing any termination event. As a result, a CEO may be used to initiate executions of multiple methods whose order of acceptance is nondeterministic. Note that EEOs may also be used to initiate multiple methods, but, in that case, the issue of which EEO is used to observe which event must be explicitly managed.

The way event objects are used to make observations is shown in Figures 2.6 through 2.9.

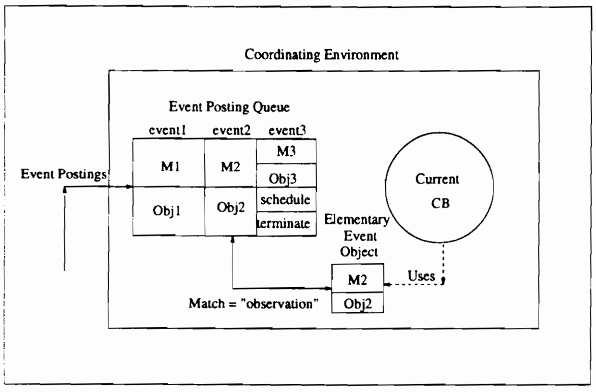

Figure 2.6: Observing an acceptance-event posting using a single-constituent elementary event object.

Figure 2.6 shows the use of a single-constituent EEO to observe an acceptance event. As shown in the figure, a CE object contains an **event posting queue** to buffer event messages from components. Note that although the queue is shown to contain the actual event messages, in reality, however, only pointers to such messages need be enqueued. The queue contains three event postings from objects *Obj1*, *Obj2*, and *Obj3*. *Obj1* and *Obj2* have posted an acceptance event each, whereas *Obj3* has posted a method-termination event. The current CB is interested in observing the acceptance of a message requesting the execution of method *M2* in component *Obj2* and so uses an EEO which contains a reference to the method *M2* and a reference to the component *Obj2*. Note that although event postings are examined in a FCFS basis, **event postings are skipped** if a CB is not interested in observing them. Skipped event postings retain their position in the queue and are processed later.

Observation of an event amounts to searching the event posting queue for an event message which contains a method reference and an object reference which match those in an event object. Thus, in Figure 2.6, the second event posting in the queue matches with the event object used by the CB and when such a match is detected, the CB makes an **"observation"**.

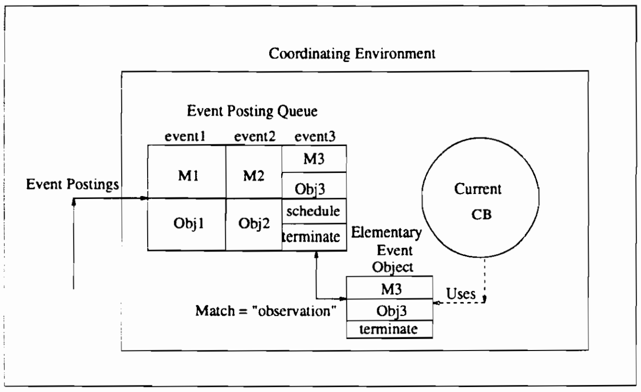

Figure 2.7: Observing a termination-event posting using a single-constituent elementary event object.

Figure 2.7 shows the use of a single-constituent EEO to observe a termination event. A slightly enhanced EEO which contains a field marked *terminate* is used by the CB to observe a method-termination event. Thus, the matching algorithm looks for an event message which has *M3*, *Obj3*, and *terminate* in order to make a successful observation. In the figure, the first event message in the queue satisfies the search causing the CB to make an observation.

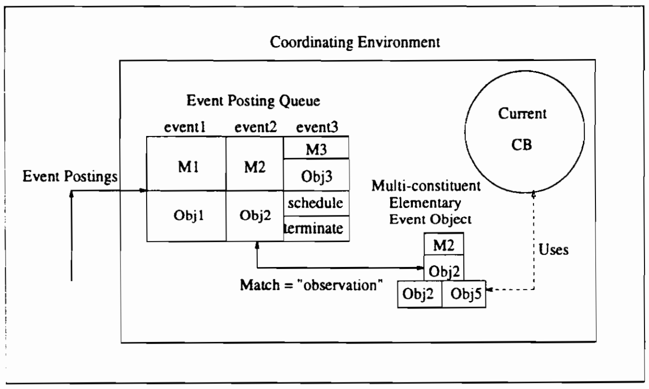

Figure 2.8: Observing an acceptance-event posting using a multi-constituent elementary event object.

Figure 2.8 shows the use of a multi-constituent EEO to observe an acceptance event. The EEO in the figure contains a list of two components, *Obj2* and *Obj5*. In a multi-constituent EEO, one may specify a component from the component list in which the next event must be observed by the EEO. If such a component is not specified, the EEO observes the first event posted by any one of the components in the list. In Figure 2.8, the EEO is marked to observe the next event in component *Obj2* (shown in the second slot of the EEO). Hence an event posting from *Obj2* is observed. Termination events may be observed using multi-constituent EEOs in the same way as single-constituent EEOs.

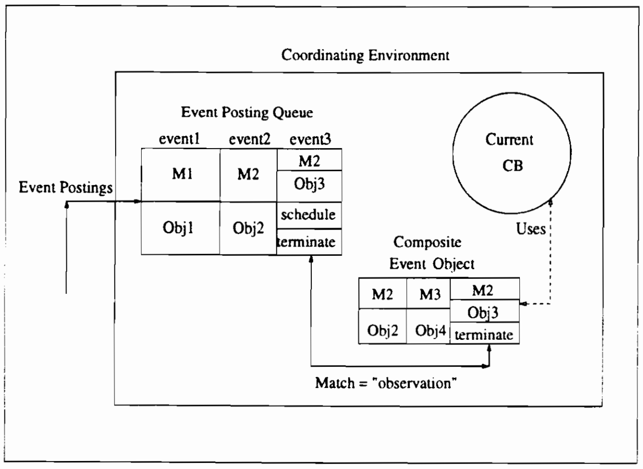

Figure 2.9: Observing a termination-event posting using a composite event object.

Figure 2.9 shows the use of a CEO to observe a termination event. In the figure, the CEO contains three single-constituent EEOs: one to observe events related to method *M2* in component *Obj2*, one to observe events related to method *M3* in component *Obj4*, and one to observe events related to method *M2* in component *Obj3*. The CEO may observe acceptance events in either *Obj2* or *Obj4* and a termination event in *Obj3* (an acceptance event must have been observed in *Obj3* using this CEO in the past). Since the termination event from component *Obj3* was posted the earliest, it is observed by the CEO.

Figure 2.10 shows how different types of event objects are used by a CE object to observe event messages from specific components. In the figure, the CE object uses one single-constituent EEO, one multi-constituent EEO and one CEO. Using the single-constituent EEO, the CE object observes event messages posted by component one. Using the multi-constituent EEO, the CE object observes event messages posted by components two and three. The CEO consists of one single-constituent EEO and one multi-constituent EEO. Using the single-constituent EEO, the CEO observes event messages posted by component four. Using the multi-constituent EEO, the CEO observes event messages posted by components five and six.

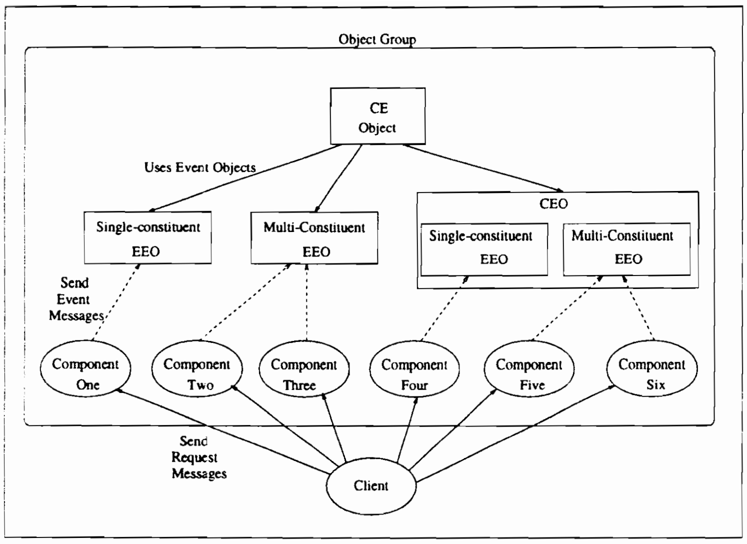

Figure 2.10: How event objects bind a CE object and the coordinated components.

2.4.2 The Coordinating Actions

Some coordinating actions are implemented as methods of event objects and the remaining are implemented as methods of CE objects. In this section, the interfaces of all the C++ classes which realize event objects and CE objects are introduced and the methods which realize coordinating actions are identified. Only enough detail of each class is introduced so that the example coordination problems in the next chapter may be understood. A more detailed implementation of the classes is discussed in Chapter Five.

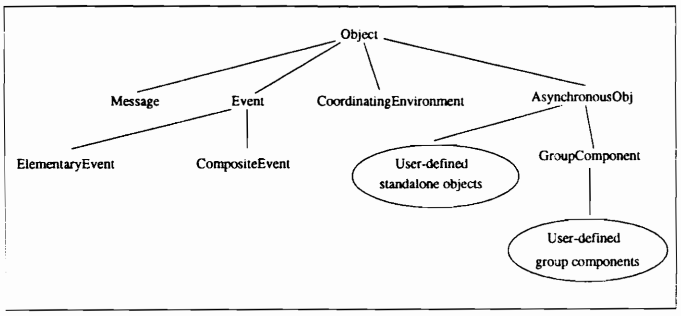

Figure 2.11: The class hierarchy used to realize elementary and composite event objects, CE objects, standalone autonomous objects, and group components.

The class hierarchy in Figure 2.11 shows the classes which are used to implement request-message objects, the two types of event objects, CE objects, standalone autonomous objects, and group components. A predefined abstract class called *Object* is assumed to exist which realizes the abstract behavior of all objects in the system. Request-message objects are instantiated from the *Message* class. Elementary event objects are instantiated from the *ElementaryEvent* class and composite event objects are instantiated from the *CompositeEvent* class. CE objects are instantiated from the *CoordinatingEnvironment* class.

```
enum executionMode {thisThread, newThread};
class Message : public Object {
    ...
public:
    Message(executionMode, methodId, ... );
    int SendOnce(Object*);
    void SendUntilAccepted(Object*);
    ...
};
```

A partial declaration of the *Message* class is shown above. The constructor of the class expects the execution mode of the message which can be either *thisThread* or *newThread.* In the former case, the thread of the client is used to execute the requested method whereas in the latter case a new thread is spawned. A reference to the requested method and zero or more actual argument values required for invoking that method are also supplied as arguments to the constructor. The *SendOnce* method realizes the *only-once* mode of message passing and the *SendUntilAccepted* method realizes the *until-accepted* mode of message passing. Each of the two latter methods expects as an argument the object to which the message must be sent.

```
enum eeoUsage {accept, terminate, noevent};
class ElementaryEvent : public Event {
    ...
public:
    ElementaryEvent(methodId, GroupComponent*, ...);
    Event* EventUsage(eeoUsage);
    Event* NextEventIn(GroupComponent*);
    GroupComponent* WhichComponent();
    void Schedule();
    void Block();
    void Unblock();
    void AwaitTermination(CoordinatingEnvironment*);
```

A partial declaration of the *ElementaryEvent* class is shown above. The method whose acceptance and termination are observed and the components in which such events are observed are passed as arguments to the constructor. The *EventUsage* method sets the usage mode of the EEO. The default usage mode is the observation of an acceptance event (denoted by *accept*). The other usage modes are *terminate* for observing a termination event and *noevent* for observing no events. The *NextEventIn* method marks the component in a multi-constituent EEO in which the next event must be observed. The *WhichComponent* method extracts a pointer to the component in which the last event was observed. The four remaining methods realize four coordinating actions. The *Schedule* method schedules the requested method in the last acceptance event observed using an EEO. The *Block* method blocks the acceptance of the method recorded in an EEO in all the constituent components. The *Unblock* method unblocks blocked methods. The *AwaitTermination* method schedules and waits for the termination of a requested method.

```
enum ceoUsage {accept, terminate, acceptTerminate, noevent};
class CompositeEvent : public Event {
    ...
public:
    CompositeEvent(ElementaryEvent*, ...);
    Event* EventUsage(ceoUsage);
    int IsAccept();
    int IsTerminate();
    GroupComponent* WhichComponent();
    void Schedule();
    void Block();
    void Unblock();
    void AwaitTermination(CoordinatingEnvironment*);
```

A partial declaration of the *CompositeEvent* class is shown above. A list of the constituent EEOs which make up a CEO are passed as arguments to the constructor. The *EventUsage* method sets the type of event which is observed using a CEO. A CEO may be used to observe an acceptance event (denoted by *accept*), or a termination event (denoted by *terminate*), or either an acceptance or a termination event (denoted by *acceptTerminate*). The type of observation must be set before each use of a CEO since the default type is *noevent*. If the observation mode of a CEO is set to *acceptTerminate*, then after making an observation using the CEO, the event which was observed is determined using the methods *IsAccept* and *IsTerminate*. The method *IsAccept* returns the value *true* if an acceptance event was observed and the method *IsTerminate* returns the value *true* if a termination event was observed. The *WhichComponent* method extracts a pointer to the component in which the last event was observed. The four remaining methods realize four coordinating actions. The *Schedule* method schedules the requested method in the last acceptance event observed. The *Block* method blocks the acceptance of the methods recorded in all the EEOs in a CEO. The *Unblock* method unblocks blocked methods. The *AwaitTermination* method schedules and waits for the termination of a requested method.

```
class CoordinatingEnvironment : public Object {
   ...
public:
   ...
   void Ignore(methodId, GroupComponent*, ...);
   void AddArguments(Event*, ...);
   void Become(methodId, ...);
   void Observe(Event*, ...);
```

A partial declaration of the *CoordinatingEnvironment* class is shown above. It defines the methods which realize three more coordinating actions and the *Observe* operation. The *Ignore* method inhibits the posting of an acceptance event associated with a specific method by one or more components. The inhibited method and the components which are affected are passed as parameters. The actual inhibition is done by invoking a special method in each component and passing the method identifier as an argument. The *AddArguments* method appends arguments to the argument list of an event posting. The *Become* method asynchronously invokes the initial or a replacement CB. The *Observe* method implements the search algorithm that matches event objects with event postings. It accepts an arbitrary number of event-object pointers as arguments and applies a predefined algorithm to extract an event posting which matches with one of the event objects. It is a blocking operation which terminates only after an observation has been made. It returns a pointer to the event object using which makes a successful observation.

The objects which constitute an application program may be categorized into two types: those which do not participate in any object group (referred to as *standalone objects*) and those which do participate in object groups (referred to as *group components*). The abstract behavior of standalone objects is captured by the *AsynchronousObj* class and the abstract behavior of group components is captured by the *GroupComponent* class. A partial declaration of the *GroupComponent* class is shown below.

```
class GroupComponent: public AsynchronousObj {
    ...
public:
    ...
    void BlockComponent();
    void UnblockComponent();
    void RegisterCE();
    void BlockMethod();
    void UnblockMethod();
    void InhibitMethod();
```

The *BlockComponent* and *UnblockComponent* methods are used before and after the *Call* coordinating action of a CE object, respectively. These methods block and unblock the RH in a component so that the requested method may execute without interfering with the activities of a component. The *RegisterCE* method is used to record a CE object in a component so that the component may post events to the CE object. The *BlockMethod* method is used by the *Block* coordinating action of a CE object to mark that a specific method is blocked from executing. The *UnblockMethod* method is used by the *Unblock* coordinating action of a CE object to unblock a blocked method. The *InhibitMethod* method is used by the *Ignore* coordinating action of a CE object to mark that a specific method must never post acceptance events.

2.4.3 The Observation-Action Protocol

This section discusses the intended usage of the operations of the classes declared in the last section so that the semantics of the model and the constraints of the current realization are both honored. The operations of the *ElementaryEvent* and the *CompositeEvent* classes will be described in the following sections using *lifecycle diagrams* of the event objects which are instantiated from those classes. Among the operations of the *CoordinatingEnvironment* class, the usage of the *Observe* and the *AddArguments* operations is also shown in the lifecycle diagrams. Out of the remaining public operations of the *CoordinatingEnvironment* class, *Ignore* must be used from a subclass of the *CoordinatingEnvironment* class to mark the methods which must not post acceptance events throughout the lifetime of a CE object. The *Become* operation, which is used to specify the replacement CB, must be the last operation executed by a CB.

2.4.3.1 Using a Single-Constituent Elementary Event Object

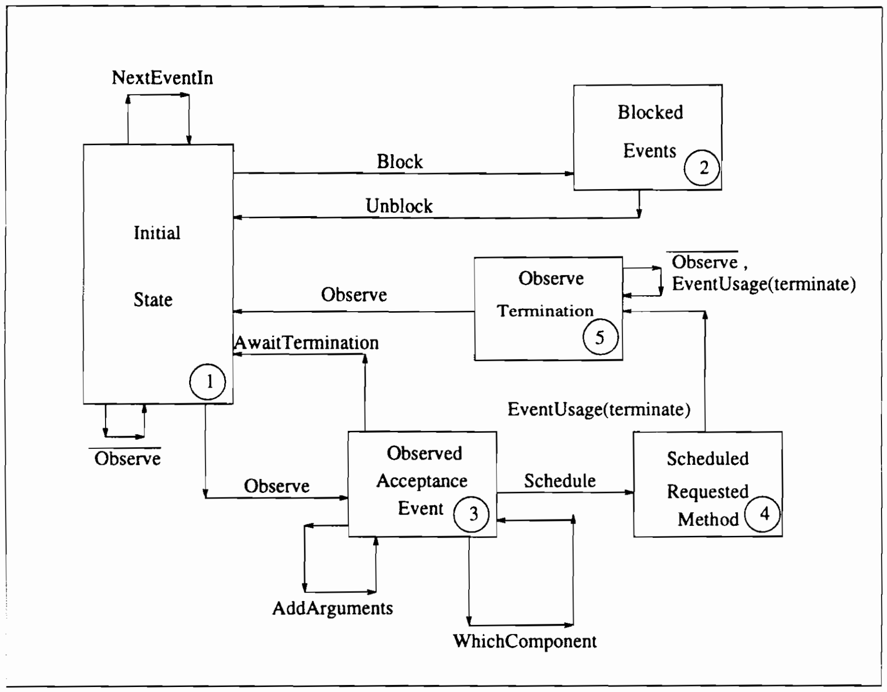

Figure 2.12: Lifecycle of a single-constituent elementary event object.

Figure 2.12 shows the lifecycle of a single-constituent elementary event object instantiated from the *ElementaryEvent* class. The different states in the lifecycle are shown as boxes.

The initial state of the lifecycle diagram is the box marked 1. Labeled arcs emanating from a box mark the state transitions. The labels can be one of either the public operations of the *ElementaryEvent* class, or the *Observe* operation, or $\overline{Observe}$ which indicates the failure of the *Observe* operation to observe an event using the event object, or the *AddArguments* operation.

In the initial state, an event object may be used in one of three ways. First, the *NextEventIn* method may be invoked to specify the component in which the next event must be observed. In a single-constituent EEO this assignment has no special significance and the operation is shown for the sake of completeness. Second, it may be used to block event postings by invoking the *Block* method which causes a transition to state 2. When the *Unblock* method is invoked in state 2, the event object returns to the initial state. Third, it may be used in an *Observe* operation. If an acceptance event is observed using the event object, then the *Observe* arc causes a transition to state 3. If the event object is not used to observe an event, then the event object remains in the initial state when *Observe* terminates (specified by the $\overline{Observe}$ transition).

In state 3, the *AddArguments* and *WhichComponent* methods can be invoked to append argument values to the request message whose acceptance was observed and to retrieve a pointer to the event-posting component, respectively. The latter methods do not cause any state transition. The *AwaitTermination* and the *Schedule* methods cause state transitions from state 3. The *AwaitTermination* method is invoked to both schedule the request message for execution and await the termination of the method and the *Schedule* method is invoked to only schedule the request message for execution. When the scheduled method terminates, *AwaitTermination* terminates and the event object reverts back to the initial state and, when *Schedule* terminates, the event object transits to state 4.

In state 4, the only possible operation on the event object is to use *EventUsage* to set the usage type of the object to *terminate* (to observe termination) and transit to state 5. In state 5, the only operation possible is to use the event object in an *Observe* operation. If a termination event is observed successfully using the event object then a transition is made to the initial state. If not, the event object remains in state 5 and waits to be used in a subsequent *Observe* operation.

2.4.3.2 Using a Multi-Constituent Elementary Event Object

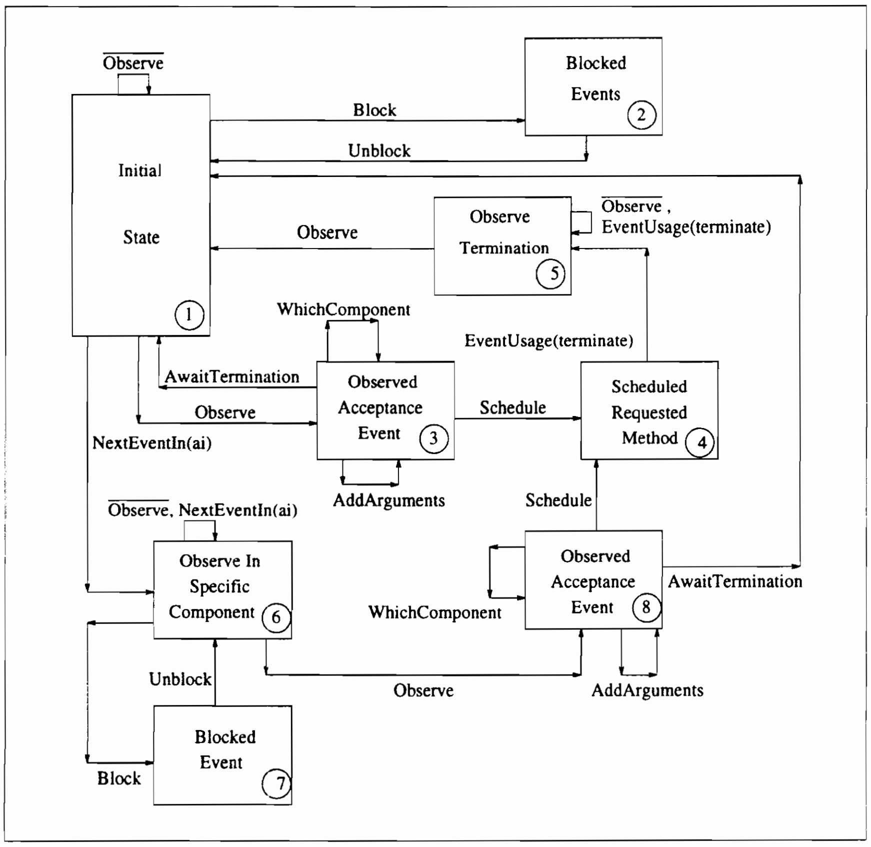

Figure 2.13: Lifecycle of a multi-constituent elementary event object.

Figure 2.13 shows the lifecycle diagram of a multi-constituent elementary event object instantiated from the *ElementaryEvent* class.

The *Block-Unblock* loop from the initial state is similar to the one in Figure 2.12, except that event postings from all the components recorded in the EEO are blocked. The *Observe-AwaitTermination* loop (box 1 - box 3 - box 1) from the initial state has the same interpretation as in Figure 2.12 except that an acceptance event is observed in the first component of the constituents list whose event posting was closest to the head of the event posting queue. The $\overline{\textit{Observe}}$ transition in the initial state, as before, specifies the state of the object when no event is observed using it. Finally, the *Observe-Schedule-EventUsage(terminate)-Observe* loop (box 1 - box 3 - box 4 - box 5 - box 1) from the initial state has the same interpretation as in Figure 2.12.

A new transition which is possible from the initial state is by using the *NextEventIn* method to transit to state 6. The latter method is used to specify the component a_i (which must be a member of the list of components stored in the EEO) in which the next acceptance event must be observed. From state 6, the *Block-Unblock* loop (box 6 - box 7 - box 6) may be initiated which blocks and unblocks event postings from only the component a_i.

Among the other possible transitions which may be initiated from state 6 are *Observe-AwaitTermination* (box 6 - box 8 - box 1) and *Observe-Schedule-EventUsage(terminate)-Observe* (box 6 - box 8 - box 4 - box 5 - box 1). In the first case, an acceptance event in a_i is observed, the request message is scheduled for execution, and the termination of the scheduled method is observed. In the second case, an acceptance event in a_i is observed, the request message is scheduled for execution, then the event object is marked so as to observe a termination event, and finally it is used in an *Observe* operation to observe the termination of the scheduled method. If the *Observe* operation from state 6 does not use the event object to observe an event, then the object remains in state 6 and waits to be used in a subsequent *Observe* operation.

2.4.3.3 Using a Composite Event Object

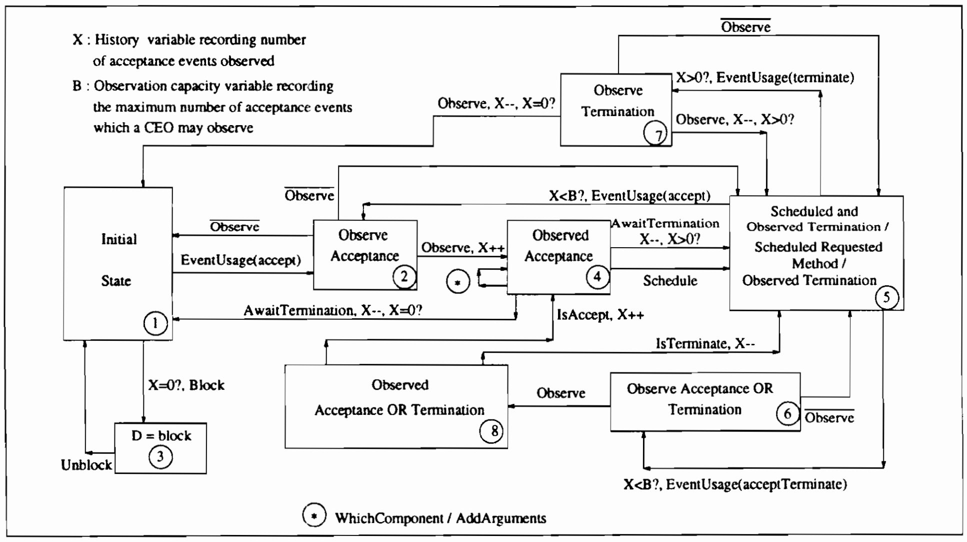
X : History variable recording number of acceptance events observed
B : Observation capacity variable recording the maximum number of acceptance events which a CEO may observe
Observe
Observe Termination
X>0?, EventUsage(terminate)
Observe, X--, X=0?
Observe, X--, X>0?
Observe
X<B?, EventUsage(accept)
Initial State
Observe
EventUsage(accept)
Observe Acceptance
Observe, X++
Observed Acceptance
AwaitTermination X--, X>0?
Schedule
Scheduled and Observed Termination / Scheduled Requested Method / Observed Termination
AwaitTermination, X--, X=0?
IsAccept, X++
IsTerminate, X--
X=0?, Block
Observed Acceptance OR Termination
Observe
Observe Acceptance OR Termination
Observe
D = block
Unblock
X<B?, EventUsage(acceptTerminate)
WhichComponent / AddArguments

Figure 2.14: Lifecycle of a composite event object.

Figure 2.14 shows the lifecycle of an event object instantiated from the *CompositeEvent* class. The history variable X records the number of acceptance events which have been observed using a CEO and the observation capacity variable B records the maximum number of acceptance events which may be observed using a CEO. The different values of X and B are used as pre-conditions, post-conditions, and post-actions to the public operations of the *CompositeEvent* class and to the *Observe* operation. The labels on the transition arcs can be one of either the public operations of the *CompositeEvent* class or one of *Observe* or *AddArguments*.

A possible transition from state 1 is to invoke the *Block* method and transit to state 3. This transition can be made only if X has a value of zero (that is, no event postings have been observed using the event object). Hence, $X=0$ followed by a "?" appears as the pre-condition for applying the *Block* operation. The only transition possible from state 3 is to invoke the *Unblock* operation and transit to state 1.

The other transition possible from the initial state is to state 2 by using the *EventUsage* method to set the usage type of the object to *accept* (to observe an acceptance event). From state 2, the only transition possible is by using the event object in an *Observe* operation. If an acceptance event is observed using the composite event object, X is incremented as a post-action and the object transits to state 4. If the event object is not used to observe an event, then a transition is made to state 1.

The first transition possible from state 4 is through the invocation of the *AwaitTermination* operation to schedule the request message observed by the CEO and wait for the termination of the corresponding method. As a post-action, X is decremented by one and depending on whether X is zero or non-zero, the object either transits to state 1 or to state 5, respectively. The second transition possible from state 4 is to state 5 through the invocation of the *Schedule* method which schedules the request message observed by the CEO. The other transitions possible from state 4 are through the invocations of the *WhichComponent* and *AddArguments* operations which either return a pointer to the component that posted an event or append arguments to the request message observed, respectively.

There are three possible transitions from state 5. The first transition, to state 7, marks the usage type of the object to *terminate* using the *EventUsage* operation if $X > 0$ (that is, if there is any termination event to be observed). The second transition, to state 2, marks the event object to observe an acceptance event using *EventUsage* if all the EEOs have not been used up in observing acceptance events (that is $X<B$ is true). The third transition, to state 6, marks the event object to observe either an acceptance or a termination event

only if there is at least one EEO available which can observe an acceptance (that is, $X<B$ is true). Otherwise, one may try to observe more acceptance events than there are EEOs in a CEO.

There are two possible transitions from state 7. An event object in this state may be used in an *Observe* operation in order to observe a termination event. If the event object is used to observe such an event, then as a post-action, X is decremented. Then depending on whether the new value of X is zero or greater, the event object transits to either state 1 or to state 5, respectively. If the event object is not used to observe an event, then its state reverts back to state 5.

The only transition possible from state 6 is to state 8 through the successful observation of either an acceptance or a termination event by an *Observe* operation which uses this event object. Note that X is not incremented as a post-action when *Observe* terminates because which of the two possible events was observed cannot be inferred. If the event object is not used to observe an event, then its state reverts back to state 5.

The two possible transitions from state 8 inquire whether an acceptance or a termination event was observed using the event object. If acceptance was observed, the method *IsAccept* returns true, X is incremented as a post-action, and a transition is made to state 4. If termination was observed, the method *IsTerminate* returns true, X is decremented as a post-action, and a transition is made to state 5.

Chapter 3

Solving Coordination Problems using Coordinating Environments

3.1 A Method for Developing Reusable Coordination Specifications

This section describes a method for solving coordination problems in C++ using the Coordinating Environments (CEs) model; the next section applies the method to solve several coordination problems. The method makes extensive use of the **inheritance**, the **polymorphism**, and the **dynamic binding** mechanisms of C++ to increase the reusability of user-defined coordination specifications. The two major steps of the method and the ways in which the three mechanisms are utilized by the method are explained in the following.

3.1.1 Step One: Define an Abstract CE class

A coordination problem is described in a class called an **abstract CE class**. An abstract CE class is a subclass of the *CoordinatingEnvironment* class and describes the desired coordination without making any reference to the names of the classes (and their methods) from which the coordinated components are instantiated. The individual steps involved in defining an abstract CE class are described below.

3.1.1.1 Declare Instance Variables

The most significant instance variables of an abstract CE class are pointers to components and pointers to event objects. Pointers to components are declared as pointers to the *AsynchronousObj* and *GroupComponent* classes, because, due to polymorphism, pointers of the latter classes may store pointers to any of their subclasses.

3.1.1.2 Define CB Methods

To hide the behavior of a CE object from clients, a Coordinating Behavior (CB) is realized as a *protected* method of the abstract CE class, called a **CB method**. Corresponding to each state of a group, a CB method is defined that describes the observed events and the corresponding coordinating actions.

3.1.1.3 Define Replacement-CB Methods

A CB method executes a replacement CB by invoking a protected, virtual method called a **replacement-CB method**. A replacement-CB method invokes the *Become* operation. This indirect way of invoking the *Become* operation enables the modification of the behavior of a CE object by redefining replacement-CB methods in subclasses of an abstract CE class.

3.1.1.4 Declare Coordinating-Action Methods

Coordinating actions are not taken directly from the CB methods of an abstract CE class. Instead, special methods called **coordinating-action methods** (which have names suffixed by *CA*) are declared as pure virtual methods in an abstract CE class. A subclass of an abstract CE class provides the actual implementations of these methods and, utilizing the dynamic binding mechanism, the methods in a subclass are executed from the CB methods of the abstract CE class.

3.1.1.5 Define Observe-Event Methods

Special virtual methods, called **observe-event methods**, are defined from which *Observe* operations are invoked. The virtual observe-event methods introduce the possibility of executing redefinitions of such methods in subclasses of an abstract CE class. Such redefinitions may utilize event objects in slightly different ways and can thereby change the observation pattern of CE objects.

3.1.1.6 Define the Initiate Method

The initial CB method is executed by invoking a special method called *Initiate*. *Initiate* is not declared virtual and is not invoked from the constructor of an abstract CE class because the dynamic binding mechanism does not work when virtual methods are invoked from constructors (in C++). Thus, if the initial CB method changes, then a new *Initiate* method must be defined in a subclass of an abstract CE class.

3.1.2 Step Two: Define a Concrete CE class

To enable the reuse of the coordination effect (referred to as the **coordination schema**) embodied in an abstract CE class, **CE objects** are instantiated from subclasses of abstract CE classes. These subclasses are called **concrete CE classes**. Each concrete CE class maps the coordination schema of an abstract CE class to a specific coordination problem.

The following must be done in a concrete CE class. First, implementations of coordinating-action methods must be provided. Second, if any events must be ignored, then a method must be defined which invokes the *Ignore* operation to inhibit the corresponding event postings. This method must be invoked from the constructor of the concrete CE class. Third, the following set of steps may be taken: add new instance variables, redefine CB methods, add new CB methods, redefine replacement-CB methods, add new replacement-CB methods, redefine the *Initiate* method, define new coordinating-action methods, redefine observe-event methods, and add new observe-event methods.

In the next section, the above method is applied to solve several coordination problems.

3.2 Example Coordination Problems

In this section, the utility of the method described in the last section is shown by solving several well-understood and simple coordination problems. The solutions stress the overall philosophy of the CEs model: composability and reusability.

The first problem, coordinating a **panel of buttons** so that only one button remains depressed at any point in time, is the simplest and is used to introduce the basic steps of the method. The solution uses two multi-constituent elementary event objects (EEOs), one coordinating-action method, one replacement-CB method, and two observe-event methods. A concrete CE class which provides an implementation of the coordinating-action method is defined and the steps required to initialize a group of two button objects is described.

The second problem, coordinating a coin acceptor and two slot objects so that the ensemble behaves like a **vending machine**, is a slightly more involving problem. The solution uses a composite event object (CEO) to permanently block a few methods in the components. It also uses the *Block* and *Unblock* operations, the *Ignore* operation, and uses four coordinating-action methods.

The third problem, coordinating a **multi-coin-acceptor vending machine**, demonstrates how the concrete CE class for the vending machine problem may be reused to solve a slightly different coordination problem. The new concrete CE class introduces a new instance variable, redefines two coordinating-action methods and one observe-event method, and defines two new protected methods for currency conversion.

The fourth problem, coordinating a group of **dining philosophers**, demonstrates how a group of concurrently executing components may be coordinated in which deadlock is an important issue. The solution uses a CEO to make observations and demonstrates the utility of CEOs in making multiple event observations.

The fifth problem, coordinating two half-adder components and an OR-gate component so that the ensemble behaves like a **full-adder** circuit, demonstrates the possibility of applying the CEs coordination model in solving coordination problems arising in the software simulation of hardware circuits. The primary coordination issues are the timely application of individual circuit functionalities once the proper input signals are obtained and, once the output is evaluated by a component, the timely transfer of the output values to the input terminals of the next stage. The main advantages of using a CE object to coordinate the components of a hardware circuit are that components do not have to be aware of their communication partners and they are relieved from determining when to apply their output-producing functions.

The sixth problem, connecting two full-adder circuits to form a **ripple-carry adder**, demonstrates how a CE object may compose multiple CE objects. The primary coordination issue is one of state propagation: the carry-bit of the first full-adder circuit must be transferred to the carry-input bit of the second full-adder circuit.

The last problem, coordinating multiple elevator-cars in an **elevator system**, demonstrates the ability of the CEs model to handle non-trivial coordination problems. The solution is structured as a collection of three CE objects which operate in parallel to coordinate the different components which comprise an elevator system. The three CE objects synchronize by observing events in and accessing information from some shared components.

The crucial coordination constraints in each of the above coordination problems is summarized in the following table.

Summary of Coordination Problems

Name of Problem	Crucial Coordination Constraint(s)
Panel of Buttons	• At most one depressed button at a time.
Vending Machine	• No candy without coins. • Transfer inserted amount to slot. • Refund excess amount.
Multi-Coin-Acceptor Vending Machine	• The three constraints of a regular vending machine restricted to one coin acceptor at a time.
Dining Philosophers	• Allow maximum possible concurrency. • Prevent deadlock.
Full Adder	• Timely application of logic function. • Timely transfer of output.
Ripple-Carry Adder	• Composition of CE objects.
Elevator System	• Multiple CE objects cooperating to coordinate a highly concurrent group of components.

3.2.1 Coordinating a Panel of Buttons

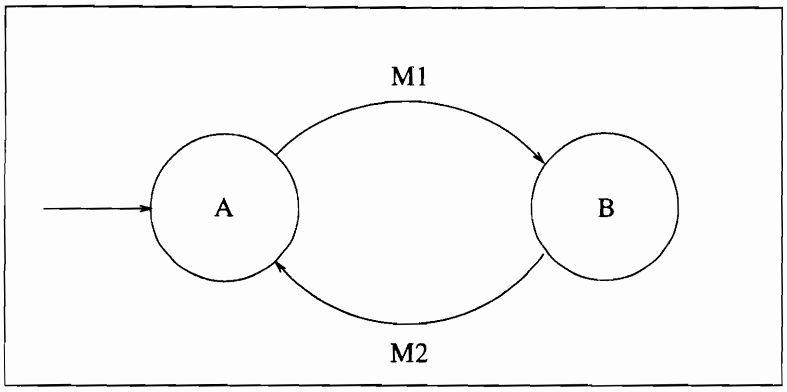

Figure 3.1: State diagram depicting the possible states and the available interface in each state of an autonomous object.

The state diagram in Figure 3.1, shows the behavior of a type of autonomous object which has two methods, *M1* and *M2*, in its public interface. The object can be in one of two states labeled *A* and *B*. In state *A*, the initial state, it accepts a message requesting to execute *M1* (henceforth, *M1 message*) as shown by the arc labeled *M1* which leaves state *A*. The object transits to state *B* after the termination of method *M1*. A message requesting *M2* (henceforth, *M2* message) is not accepted (henceforth, *blocked*) in state A depicted by the absence of any arc labeled *M2* emanating from state A. In state B, the object accepts an *M2* message and transits to state A after method *M2* terminates but blocks an *M1* message. Thus, the current public interface (CPI) in state A consists of method *M1* and the CPI in state B consists of method *M2*.

Consider a group of objects where each object has the behavior described above. The group has the following coordination constraint: at any point in time at most one component may be in state B (assuming that all components start in state A). To enforce the group constraint, before allowing the execution of an *M1* message, a group coordinator must force a component in state B, if any, to transit to state A by sending an *M2* message to that component.

An application of the above coordination schema is to coordinate a panel of button objects which have the following behavioral constraint: at any point in time only one button may remain in the depressed mode. By mapping method *M1* to a method called *Depress*, say, which depresses a button, and by mapping *M2* to a method called *Undepress*, say, which undepresses a button, the above coordination schema may be utilized to coordinate the panel of button objects. The state A in Figure 3.1 will correspond to the undepressed state of a button and state B will correspond to its depressed state. The blocking of *undepress* messages in state A represents the inability to undepress a button which has not been depressed and the blocking of *depress* messages in state B represents the inability to depress a button which is already depressed.

An issue which arises when trying to specify coordination schemas using abstract CE classes is that of choosing the names of the instance variables, event objects, CB methods, and that of the class itself so as to convey the generality of the coordination problem. One way to address this issue is to specify the schema in terms of an example coordination problem which may be solved using the schema. Using that scheme, the coordination schema described above is specified in terms of the button-panel coordination problem. It must be noted that this form retains the reusability of the coordination schema because no specific mapping of *M1* and *M2* are made in the abstract CE class.

The abstract CE class *MultiButtonPanel* is defined below. Note that *THISCE* is a macro name which stands for the predefined name *this*.

```
class MultiButtonPanel : public CoordinatingEnvironment   {
public:
    MultiButtonPanel( ElementaryEvent* e1, ElementaryEvent* e2) {
        depressedButton = Null;
        depressButton = e1; undepressButton = e2;
    };
    virtual void Initiate()
        { Become(&MultiButtonPanel::NoneDepressed); };
protected:
    GroupComponent* depressedButton;
    ElementaryEvent* depressButton, undepressButton;
    virtual void UndepressButtonCA() = 0;
    virtual void ReplacementCB2()
        { Become(&MultiButtonPanel::OneDepressed); };
    virtual Event* ObserveEvent1()
        { return Observe(depressButton); }
    virtual Event* ObserveEvent2();
        { return Observe(depressButton, undepressButton); };


    void NoneDepressed( ) {
        Event* whichEvent = ObserveEvent1();
        depressedButton = whichEvent→WhichComponent();
        whichEvent→AwaitTermination(THISCE);
        ReplacementCB2();
    };


    void OneDepressed( ) {
        Event* whichEvent = ObserveEvent2();
        if (whichEvent == depressButton) {
            UndepressButtonCA();
            depressedButton = whichEvent→WhichComponent();
            whichEvent→AwaitTermination(THISCE);
            ReplacementCB2(); };
        else {
            whichEvent→AwaitTermination(THISCE);
            depressedButton = Null;
            Initiate(); };
    };
};
```

There are two states of the group of button objects: a state in which all buttons are undepressed and a state in which one button is depressed and all the rest are undepressed. The former state is the initial state of the group which is realized by the *NoneDepressed* CB method and the latter state is realized by the *OneDepressed* CB method.

Two event objects, *depressButton* and *undepressButton* which are instantiated from the *ElementaryEvent* class, are used to observe events. These event objects can be either one-constituent or multi-constituent and each stores all the button objects which are being coordinated as constituents. The *depressButton* EEO is used to observe events associated with the *Depress* method and the *undepressButton* EEO is used to observe events associated with the *Undepress* method. The button object which is in the depressed mode at any point in time is stored in the *depressedButton* instance variable. The constructor of the abstract CE class expects pointers to two EEOs and it initializes *depressedButton*, *depressButton*, and *undepressButton*.

An important point to be noted about the abstract CE definition is that it does not explicitly record the number of components that are coordinated. That information is stored in the event objects which allows the CE object to coordinate an unspecified number of components.

In the *NoneDepressed* CB method, the *depressButton* EEO is used to observe an acceptance event and the component in which the event is observed is recorded in *depressedButton*. Then, the observed request message is scheduled for execution and on termination of the scheduled method, a transition is made to the *OneDepressed* CB method by executing the *ReplacementCB2* method.

In the *OneDepressed* CB method, the *Observe* operation is invoked using both the EEOs as arguments. The button which is depressed cannot process a *Depress* request message since that method will not be in the CPI of that button. Hence, if the acceptance of a *Depress* message is observed, then it must be in a component which is undepressed. The latter observation is utilized in coding the *OneDepressed* CB method. If *Observe* returns a pointer to *depressButton* then the *OneDepressed* CB method undepresses the currently depressed button using the coordinating-action method *UndepressButtonCA*, records the new depressed button in *depressedButton*, schedules the *Depress* method and waits for its termination, and invokes the *OneDepressed* CB method once again.

If *Observe* returns a pointer to *undepressButton* in *OneDepressed*, then the button which was depressed must have been undepressed because no undepressed button will accept an *Undepress* message (since it will not be in the CPI of that button). Thus, the CB schedules the *Undepress* method and waits for its termination, clears the *depressedButton* instance variable, and reverts back to the initial state of the group by invoking the *Initiate* method.

The concrete CE class which may be used to instantiate a panel using button objects instantiated from a *Button* class is shown below.

```
class ButtonPanel : public MultiButtonPanel {
public:
    ButtonPanel(ElementaryEvent* e1, ElementaryEvent* e2) :
        MultiButtonPanel(e1, e2) { };
protected:
   void UndepressButtonCA() {
      depressedButton→BlockComponent();
      ((Button*) depressedButton)→Undepress();
      depressedButton→UnblockComponent(); };
};
```

The constructor of the *ButtonPanel* class expects pointers to two EEOs which are aupplied as arguments to the constructor of the abstract CE class.

The concrete CE class also provides an implementation of the *UndepressButtonCA* method from which the *Undepress* coordinating-action method in the component stored in *depressedButton* is invoked. In order to ensure that the latter invocation does not interfere with the activities of the request handler (as discussed in chapter two), the *BlockComponent* method is invoked on the depressed button object. The latter invocation causes the request handler to "un-accept" any *Undepress* message which it has already accepted and prevents it from processing any request messages until the coordinating action is over.

After *BlockComponent* terminates, the *Undepress* method is invoked from *UndepressButtonCA*. After the latter invocation terminates, the request handler is unblocked by invoking the *UnblockComponent* method.

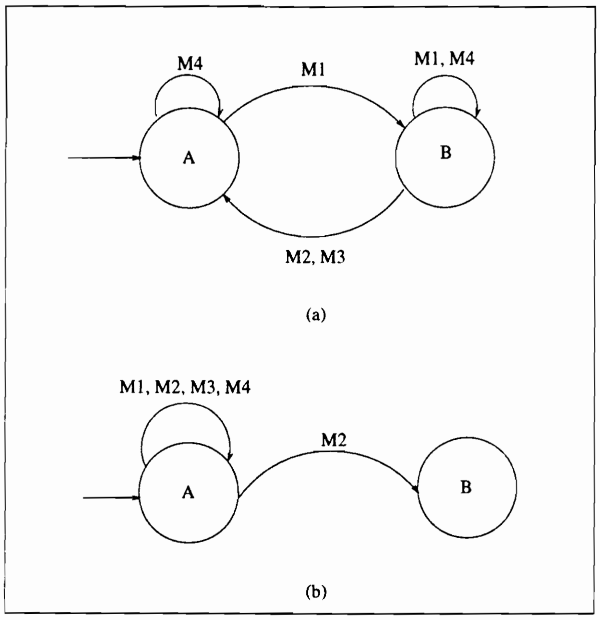

Figure 3.2: State diagram depicting the possible states and the available interface in each state of two types of autonomous objects.

Using one object of the type shown in Figure 3.2a and multiple objects of the type shown in Figure 3.2b, one may construct a group which represents one coin acceptor and several slots, respectively, of a vending machine.

The methods in Figure 3.2a may be mapped as follows. Method *M1* may stand for a method *Insert* using which coins may be inserted into the coin acceptor. The value of the coin inserted is passed as an argument to the method. Method *M2* may stand for a method *RefundAll* using which the full amount deposited in the coin acceptor may be returned. Method *M3* may stand for a method *RefundExcess* using which the amount in excess, if any, to the amount inserted may be returned. Method *M4* may stand for a method *InsertedAmount* using which the accumulated value of the coins inserted in the coin acceptor may be retrieved. The blocking of *RefundAll* and *RefundExcess* messages in state *A* represents the inability to extract coins from the coin acceptor when none has been inserted. The acceptance and execution of multiple *Insert* messages in state *B* represents the capability of inserting multiple coins before extracting an item.

The methods in Figure 3.2b may be mapped as follows. Method *M1* may stand for a method *RequestItem* using which the desire to extract an item from the slot may be expressed. Method *M2* may stand for a method *DispenseItem* using which an item is extracted from a slot. Method *M3* may stand for a method *HowManyItems* using which the number of items available in a slot may be determined. Method *M4* may stand for a method *Price* using which the price of an item in a slot may be determined. The *RequestItem* method expects an amount value as an argument. To increase the reusability of a slot object (for example, in situations when items may be extracted without depositing any money), the argument of the *RequestItem* method is defined as a default argument and is assigned a value of zero. *RequestItem* determines if the amount value is sufficient for the slot to dispense an item. If so, *RequestItem* invokes the *DispenseItem* method to dispense an item and decrements the count of stored items by one. Otherwise, no item is dispensed and the method terminates. If, after dispensing an item, the number of items becomes zero, the slot becomes inactive by transiting to state *B* in which it disregards all request messages.

The obligations of a coordinator for the vending machine are that it must ensure that no slots may be opened before coins are inserted, ensure that after inserting coins only one slot dispenses an item, and ensure that, after an item has been dispensed, any overpaid amount is refunded and the coin acceptor is prepared to repeat the coin insertion cycle.

The *MultiComponentMachine* abstract CE class defined below realizes the coordination required by multi-component machines similar to a vending machine.

```
class MultiComponentMachine : public CoordinatingEnvironment    {
public:
   MultiComponentMachine(ElementaryEvent* e1, ElementaryEvent* e2,
         ElementaryEvent* e3, CompositeEvent c1) {
      insertEvent = e1; refundEvent = e2;
      openAnySlot = e3; blockEvents = c1;
      coinAcceptor = Null;
      openAnyslot→Block();
      blockEvents→Block();
      IgnorePriceEvent(); };
   virtual void Initiate()
      {Become(&MultiComponentMachine::AwaitCustomer);};


protected:
   ElementaryEvent* insertEvent, refundEvent, openAnySlot;
   CompositeEvent* blockEvents;
   GroupComponent* coinAcceptor;
   void IgnorePriceEvent() { };
```

```
    virtual int DepositedAmountCA() = 0;
    virtual int PriceOfItemCA(GroupComponent*) = 0;
    virtual int NumberOfItemsCA(GroupComponent*) = 0;
    virtual void RefundExcessCA(int) = 0;
    virtual void ReplacementCB2()
      {Become(&MultiComponentMachine::ProcessRequest);};
    virtual Event* ObserveEvent1()
      { return Observe(insertEvent); };
    virtual Event* ObserveEvent2()
      { return Observe(insertEvent, refundEvent, openAnySlot); };


    void AwaitCustomer( ) {
      Event* whichEvent = ObserveEvent1();
      openAnySlot→Unblock();
      coinAcceptor = whichEvent→WhichComponent();
      whichEvent→AwaitTermination(THISCE);
      ReplacementCB2(); };


    void ProcessRequest( ) {
      Event* whichEvent = ObserveEvent2();
      if (whichEvent == insertEvent) {
        whichEvent→AwaitTermination(THISCE);
        ReplacementCB2(); };
      if (whichEvent == refundEvent) {
        whichEvent→AwaitTermination(THISCE);
        Initiate(); };
      if (whichEvent == openAnyslot) {
        GroupComponent* whichSlot = whichEvent→WhichComponent();
        int price = PriceOfItemCA(whichSlot);
        int amount = DepositedAmountCA();
        int numOfItems = NumberOfItemsCA(whichSlot);
        AddArguments(whichEvent, amount);
        whichEvent→AwaitTermination(THISCE);
        int newNumOfItems = NumberOfItemsCA(whichSlot);
        if (newNumOfItems − numOfItems == 1) {
          RefundExcessCA(price);
          whichEvent→Block();
          Initiate(); }
        else ReplacementCB2();
      };
    }
};
```

There are two states of a vending machine: a state in which it awaits a client to start inserting coins and a state in which it either accepts more coins, or a request to refund, or a request to dispense an item. The former state is the initial state of the group which is realized by the *AwaitCustomer* CB method and the latter state is realized by the *ProcessRequest* CB method.

Four event objects, the three EEOs *insertEvent*, *refundEvent*, and *openAnyslot*, and the CEO *blockEvents*, are used to observe and block events. The *insertEvent* and the *refundEvent* EEOs are one-constituent event objects storing the coin acceptor as the only constituent and *openAnyslot* is a multi-constituent event object storing all the slots in the vending machine as constituents. The *insertEvent* EEO is used to observe events associated with the *Insert* method and *refundEvent* is used to observe events associated with the *Refund* method, both in the coin acceptor. The *openAnyslot* EEO is used to observe events associated with the *RequestItem* method in any slot. The *blockEvents* CEO is used to block the *RefundExcess* and the *InsertedAmount* methods in the coin acceptor component and the *DispenseItem* and *HowManyItems* methods in every slot component. These methods are blocked throughout the lifetime of the CE object thereby hiding these methods from the view of the clients of the vending machine. A pointer to the coin acceptor is stored in the *coinAcceptor* instance variable.

The constructor of the class expects pointers to four event objects using which it initializes the event object pointers and also sets *coinAcceptor* to *Null*. The constructor invokes the *Block* method on the *openAnyslot* EEO which blocks the acceptance of *RequestItem* messages by all the slots in the vending machine. Note that the *Block* operation causes a slot object to "un-accept" any *RequestItem* message it may already have accepted and marks this method as blocked in all slot components. The constructor also invokes the *Block* method on the *blockEvents* CEO to permanently block several methods in the coin acceptor and the slots. Finally, the constructor invokes the *IgnorePriceEvent* method which inhibits the posting of *Price* messages by the slot components. The reason a virtual *IgnorePriceEvent* method is not used in the abstract class is because the virtual method-invocation mechanism does not work when a virtual method is invoked from a constructor. As a result, in order to allow the abstract CE class to be independent of the name of the class of a slot component, an empty *InhibitPriceEvent* method is defined.

An important point to be noted about the abstract CE definition is that it does not explicitly record the number of slots which are coordinated. That information is stored in the event objects which allows the CE to handle an unspecified number of slots.

The *Initiate* method is invoked to execute the initial CB method *AwaitCustomer*. In that CB method, the *ObserveEvent1* method is invoked to observe an event using the *insertEvent* EEO. The event object returned by *ObserveEvent1* is stored in the *whichEvent* variable. After observing the acceptance of an *Insert* message, the *Unblock* method is invoked on the *openAnyslot* EEO which prepares the slots to accept *RequestItem* messages. Then, a pointer to the coin acceptor is extracted using the *whichEvent* variable and is stored in *coinAcceptor*. Finally, the observed *Insert* message is scheduled for execution and on termination of the *Insert* method, a transition is made to the *ProcessRequest* CB method by executing the *ReplacementCB2* method.

In the *ProcessRequest* CB method, the *Observe* operation is invoked using all the EEOs of the CE. If *Observe* returns a pointer to *insertEvent*, then the *Insert* method is scheduled for execution and, on its termination, *ReplacementCB2* is invoked to initiate the *ProcessRequest* CB method once again. If *Observe* returns a pointer to *refundEvent*, then the *Refund* method is scheduled for execution and, on its termination, *Initiate* is invoked to initiate the *AwaitCustomer* CB method.

If *Observe* returns a pointer to *openAnyslot* in *ProcessRequest*, then the following coordinating actions are taken. First, the slot in which *RequestItem* was activated is recorded in the variable *whichSlot*. Next, the *PriceOfItemCA* method is invoked to determine the price of an item in the requested slot and the price is stored in the *price* variable. Then, the *DepositedAmountCA* method is invoked to determine the amount accumulated in the coin acceptor and the amount is stored in the *amount* variable. Then, the *NumberOfItemsCA* method is invoked to determine the number of items present in the slot which received the *RequestItem* message and the number is stored in the *numOfItems* variable. After that, the amount retrieved from the coin acceptor is appended to the argument list supplied to the *RequestItem* method by the client. As a result, the group coordinator reduces the burden on the client by supplying the value of the default argument. After augmenting the argument list, the *RequestItem* message is scheduled for execution.

On the termination of the *RequestItem* method, the *NumberOfItemsCA* method is invoked once again to retrieve the number of items present in the slot which finished executing the *RequestItem* method. If no item was dispensed (determined using the number of items recorded before scheduling *RequestItem* and the number of items present after that method terminated), the *ReplacementCB2* method is invoked to initiate the *ProcessRequest* CB method once again. If an item was dispensed by the slot, the *RefundExcessCA* method is invoked to refund any overpaid amount and to reset the accumulated amount in the coin acceptor to zero. The *Block* method is invoked on the *openAnyslot* EEO to block the acceptance of *RequestItem* messages by all the slots in the vending machine until further coin insertion. Finally, the *Initiate* method is invoked to transit to the initial CB method.

The concrete CE class which may be used to instantiate a group of objects which behave like a vending machine is shown below. A coin acceptor is instantiated from the class *CoinAcceptor* and the slots are instantiated from the class *Slots*.

```
class VendingMachine : public MultiComponentMachine {
public:
    VendingMachine(ElementaryEvent* e1, ElementaryEvent* e2,
      ElementaryEvent* e3, CompositeEvent* c1,
      Slots* allSlots, int numOfSlots):
        MultiComponentMachine(e1, e2, e3, c1) {
      IgnorePriceEvent(allSlots, numOfSlots); };
protected:
   void IgnorePriceEvent(Slots* allSlots, int numOfSlots) {
      int i;
      for (i = 0; i < numOfSlots; i++)
         Ignore(&Slots::Price, allSlots[i]); };
   int DepositedAmountCA() {
      coinAcceptor→BlockComponent();
      int a = ((CoinAcceptor*) coinAcceptor)→InsertedAmount();
      coinAcceptor→UnblockComponent();
      return a; };
   int PriceOfItemCA(GroupComponent* theSlot) {
      theSlot→BlockComponent();
      int a = ((Slots*) theSlot)→Price();
      theSlot→UnblockComponent();
      return a; };
   int NumberOfItemsCA(GroupComponent* theSlot) {
        theSlot→BlockComponent();
        int n = ((Slots*) theSlot)→HowManyItems();
        theSlot→UnblockComponent();
        return n; };
    void RefundExcessCA(int price) {
        coinAcceptor→BlockComponent();
        ((CoinAcceptor*) coinAcceptor)→RefundExcess(price);
        coinAcceptor→UnblockComponent(); };
};
```

The constructor of the *VendingMachine* class expects pointers to four event objects which are supplied as arguments to the constructor of the abstract CE class, an array of pointers to all the slot components, and the total number of slots in the machine. The constructor uses the latter two arguments to invoke the *IgnorePriceEvent* method which inhibits the slots from posting any events related to the *Price* method. The concrete CE class also provides implementations of the four coordinating-action methods.

A two-slot vending machine may be instantiated as follows:

```
Slots* slots[2];
CoinAcceptor* CA = new CoinAcceptor();
int numOfItems = 10;
int itemPrice = 50;
slots[0]  = new Slots(numOfItems, itemPrice);
numOfItems = 10;
itemPrice = 70;
slots[1]  = new Slots(numOfItems, itemPrice);
ElementaryEvent* ev1 = new ElementaryEvent(&CoinAcceptor::Insert, CA);
ElementaryEvent* ev2 = new ElementaryEvent(&CoinAcceptor::RefundAll, CA);
ElementaryEvent* ev3 = new ElementaryEvent(&Slots::RequestItem, slots[0], slots[1]);
ElementaryEvent* blockEv1 =
    new ElementaryEvent(&CoinAcceptor::RefundExcess, CA);
ElementaryEvent* blockEv2 =
    new ElementaryEvent(&CoinAcceptor::InsertedAmount, CA);
ElementaryEvent* blockEv3 =
    new ElementaryEvent(&Slots::DispenseItem, slots[0], slots[1]);
ElementaryEvent* blockEv4 =
    new ElementaryEvent(&Slots::HowManyItems, slots[0], slots[1]);
CompositeEvent* blockedEvents =
    new CompositeEvent(blockEv1, blockEv2, blockEv3, blockEv4);
VendingMachine* VMCE = new VendingMachine(ev1, ev2, ev3, blockedEvents, slots, 2);
```

```
CA→RegisterCE(VMCE);
slots[0]→RegisterCE(VMCE);
slots[1]→RegisterCE(VMCE);
VMCE→Initiate();
```

A coin acceptor object is instantiated from the *CoinAcceptor* class and two slot objects are instantiated from the *Slots* class. The number of items and the price of an item in a slot are passed as arguments to the constructor of the *Slots* class. Three EEOs, *ev1* through *ev3*, are instantiated which are used by the CE object to make event observations. A CEO *blockedEvents* is constructed out of four EEOs. The CEO is used to block events permanently. Note that the latter CEO contains two single-constituent and two multi-constituent EEOs. After constructing the CE object *VMCE*, the coin acceptor and the two slots are made to register the CE object by invoking the *RegisterCE* method on them. Finally, the *Initiate* method is invoked on *VMCE* to execute the initial CB method.

An interesting variant of the vending machine may be realized by not blocking and unblocking *RequestItem* messages in the abstract CE class. In that case, if a *RequestItem* message is accepted by one of the slots prior to any insertion of coins, the slot will await a signal from the CE object as to what to do with the accepted message. On observing an *Insert* message, the CE object will transit to the *ProcessRequest* CB method and observe and execute the waiting *RequestItem* message thereby dispensing an item. Thus, unlike the vending machine realized above which enables item selection only after coins are inserted, the new machine will allow the choice of an item to be made before coins are inserted.

This variant of the vending machine may behave unexpectedly in certain situations. For example, if a customer sends a *RequestItem* message but does not insert any coins, the next customer who inserts adequate coins will receive the item that was selected by the previous customer. Note that from the standpoint of the second customer this is an "unexpected" behavior of the vending machine but from the standpoint of the machine, its behavior is in compliance with the behavior of the external environment.

3.2.3 Coordinating a Vending Machine with Multiple Coin Acceptors

As an example of reusing the abstract CE class of the last section, consider a vending machine with multiple coin acceptors which share a bank of slots. Each acceptor accepts coins of a unique currency. A customer may use only a single coin acceptor to insert coins before selecting an item.

Such a multi-coin-acceptor vending machine contains multiple objects of the type shown in Figure 3.2a. The new CE object which coordinates such a machine replaces the single-constituent EEOs, *insertEvent* and *refundEvent*, with multi-constituent EEOs each having as many constituents as there are coin acceptors. Moreover, the CE object must ensure that after observing the insertion of a coin in one of the acceptors, it processes further insert, refund, and item extraction requests for that acceptor only. Thus, in the *ObserveEvent2* method, the *insertEvent* and *refundEvent* EEOs must be set to observe an event in the coin acceptor in which the first insertion event was observed.

Another effect of introducing multiple coin acceptors is that the slots have to store the price of each item in more than one currency in order to determine whether the inserted amount is sufficient. But instead of extending the *Slots* class to incorporate that functionality, the new concrete CE class can be made to convert the inserted amount into a single currency which the slots recognize. As a result, the *DepositedAmountCA* method has to be redefined. Also, before refunding the excess amount deposited, the price of the item dispensed must be converted to an amount in the currency in which the refund must be made (so that the coin acceptor may determine the amount to be refunded). As a result, the *RefundExcessCA* method has to be redefined.

In the following, a new concrete CE class called *MultiCAVendingMachine* is defined as a subclass of the concrete CE class defined in the last section. As before, a coin acceptor is instantiated from the class *CoinAcceptor* and the slots are instantiated from the class *Slots*. It is assumed that the *CoinAcceptor* class can handle coins of different currencies.

```
class MultiCAVendingMachine : public VendingMachine {
public:
    MultiCAVendingMachine(ElementaryEvent* e1, ElementaryEvent* e2,
          ElementaryEvent* e3, CompositeEvent* c1, Slots* allSlots,
          int numOfSlots, GroupComponent* coinAcceptors):
             VendingMachine(e1, e2, e3, c1, allSlots, numOfSlots)
       { allCoinAcceptors = coinAcceptors; };
protected:
   GroupComponent* allCoinAcceptors;
```

```
    virtual int
        ToFixedCurrency(GroupComponent* coinAcceptor, int insertedAmount) { ... };
    virtual int ToCACurrency(GroupComponent* coinAcceptor, int price) { ... };
    int DepositedAmountCA() {
        coinAcceptor→BlockComponent();
        int insertedAmount= ((CoinAcceptor*) coinAcceptor)→InsertedAmount();
        coinAcceptor→UnblockComponent();
        int newInsertedAmount = ToFixedCurrency(coinAcceptor, insertedAmount);
        return newInsertedAmount; };
    void RefundExcessCA(int price) {
        int newPrice = ToCACurrency(coinAcceptor, price);
        coinAcceptor→BlockComponent();
        ((CoinAcceptor*) coinAcceptor)→RefundExcess(newPrice);
        coinAcceptor→UnblockComponent(); };
    Event* ObserveEvent2()
        { return Observe(insertEvent→NextEventIn(coinAcceptor),
                refundEvent→NextEventIn(coinAcceptor), openAnyslot); };
};
```

The new concrete CE class introduces an instance variable, *allCoinAcceptors*, which stores an array of pointers to instantiations of the *CoinAcceptor* class. This array is used in the two new virtual methods, *ToFixedCurrency* and *ToCACurrency*, which convert an amount in any currency to a fixed currency and vice versa, respectively. A redefinition of the *DepositedAmountCA* method is provided which, after obtaining the amount inserted into a coin acceptor, converts it to the standard currency value understood by the slots using the *ToFixedCurrency* method. A redefinition of the *RefundExcessCA* method is provided which, before refunding the excess amount, converts it to the currency value understood by the coin acceptor using the *ToCACurrency* method. A new definition of the *ObserveEvent2* method is also provided in which the *insertEvent* and *refundEvent* EEOs are marked to observe events in the coin acceptor recorded in the *coinAcceptor* instance variable (by invoking the *NextEventIn* method on them). This ensures that the vending machine observes events associated with only one coin acceptor until an item is extracted.

A two-slot vending machine having two coin acceptors which accept U.S. and Canadian currencies, respectively, may be instantiated as follows:

```
int usCurrency = 1;
int canCurrency = 2;
CoinAcceptor* coinAcc;
coinAcc[0] = new CoinAcceptor(usCurrency);
coinAcc[1] = new CoinAcceptor(canCurrency);
int numOfItems = 10;
int itemPrice = 50;
slots[0]  = new Slots(numOfItems, itemPrice);
numOfItems = 10;
itemPrice = 70;
slots[1]  = new Slots(numOfItems, itemPrice);
ElementaryEvent* ev1 =
   new ElementaryEvent(&CoinAcceptor::Insert, coinAcc[0], coinAcc[1]);
ElementaryEvent* ev2 =
   new ElementaryEvent(&CoinAcceptor::RefundAll, coinAcc[0], coinAcc[1]);
ElementaryEvent* ev3 =
   new ElementaryEvent(&Slots::RequestItem, slots[0], slots[1]);
ElementaryEvent* blockEv1 =
   new ElementaryEvent(&CoinAcceptor::RefundExcess, coinAcc[0], coinAcc[1]);
ElementaryEvent* blockEv2 =
   new ElementaryEvent(&CoinAcceptor::InsertedAmount, coinAcc[0], coinAcc[1]);
ElementaryEvent* blockEv3 =
   new ElementaryEvent(&Slots::DispenseItem, slots[0], slots[1]);
ElementaryEvent* blockEv4 =
   new ElementaryEvent(&Slots::HowManyItems, slots[0], slots[1]);
CompositeEvent* blockedEvents =
   new CompositeEvent(blockEv1, blockEv2, blockEv3, blockEv4);
MultiCAVendingMachine* VMCE =
   new MultiCAVendingMachine(ev1, ev2, ev3, blockedEvents, slots, 2, coinAcc);
coinAcc[0]→RegisterCE(VMCE);
coinAcc[1]→RegisterCE(VMCE);
slots[0]→RegisterCE(VMCE);
slots[1]→RegisterCE(VMCE);
VMCE→Initiate();
```

Two coin acceptor objects are instantiated from the *CoinAcceptor* class, one accepting U.S. currency and the other accepting Canadian currency. The remaining initializations are the same as the one-coin-acceptor vending machine of the last section except that *ev1*, *ev2*, *blockEv1*, and *blockEv2* are multi-constituent EEOs and the *coinAcc* array is passed as an argument to the constructor of the *MultiCAVendingMachine* class.

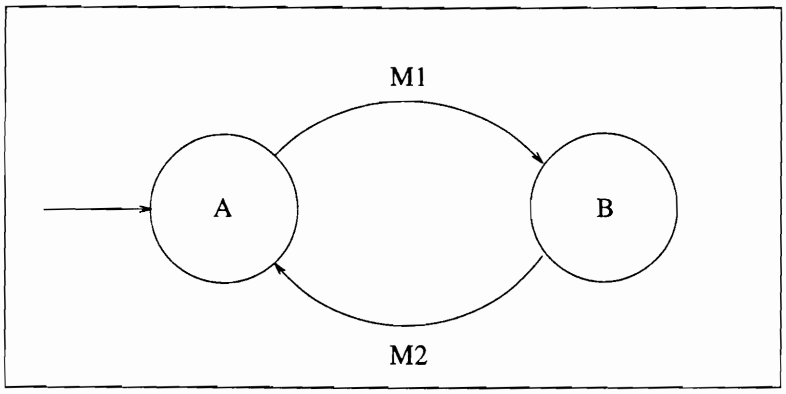

Figure 3.3: State diagram depicting the possible states and the available interface in each state of an autonomous object.

3.2.4 Coordinating The Dining Philosophers

Consider the state diagram of an autonomous object that was shown in Figure 3.1. The state diagram is repeated in Figure 3.3 for the sake of convenience. In this example, the method *M2* is assumed to be using multiple resources during its lifetime. The resources are acquired after the method starts executing and they are released before it terminates. If any resource cannot be acquired immediately, *M2* blocks until it becomes available.

Using multiple objects of the type shown in Figure 3.3, one may construct a group which represents the philosophers in the dining philosopher's problem who compete for a limited number of resources, namely, forks. The group to be coordinated consists only of the philosophers and not the forks used by them.

The methods in Figure 3.3 may be mapped as follows. Method *M1* may stand for a method *Think* and method *M2* may stand for a method *Eat*. In the *Eat* method, a philosopher acquires a pair of forks, eats with them, and releases them after eating. It is assumed that philosophers receive *Think* and *Eat* messages alternately starting with a *Think* message. Thus, each philosopher engages in an infinite sequence of activities. There are no connections among the philosophers and a philosopher may send a message to a fork without any restrictions (that is, there are no semaphores guarding access to the forks).

Unlike the previous coordination problems, the dining philosophers form a group in which there can be multiple, concurrently executing components and in which deadlock is an important issue. The main task for a coordinator of such a group is to ensure freedom from deadlock without compromising the possible degree of concurrency. The maximum concurrency possible (without causing deadlock) in a group of N philosophers sharing N forks is (N - 1). That is because, at least one of the (N-1) philosophers will be able to finish eating and release a pair of forks causing others to be unblocked eventually.

The *CompetingComponents* abstract CE class defined below realizes the coordination required by multiple competing components similar to a group of dining philosophers.

```
class CompetingComponents : public CoordinatingEnvironment    {
public:
    CompetingComponents(int numOfComponents, CompositeEvent* cev) {
      numberCompeting = 0;
      if (numOfComponents > 1)
         maxCompeting = numOfComponents - 1;
      else maxCompeting = 1;
      IgnoreThink();
      eatEvents = cev; };
   virtual void Initiate()
      {Become(&CompetingComponents::NoneCompeting);};

protected:
   int numberCompeting, maxCompeting;
   CompositeEvent* eatEvents;
   virtual void ReplacementCB2()
      {Become(&CompetingComponents::SomeCompeting);};
   virtual void ReplacementCB3()
      {Become(&CompetingComponents::MaxCompeting);};
   void IgnoreThink() { };
   virtual void OneMoreCompetitor() {
      numberCompeting++;
      if (numberCompeting == maxCompeting) ReplacementCB3();
      else ReplacementCB2(); };
   virtual void OneLessCompetitor() {
      numberCompeting--;
      if (numberCompeting == 0) Initiate();
      else ReplacementCB2(); };
   virtual void ObserveEvent1() {
      Observe(eatEvents→EventUsage(accept));
      eatEvents→Schedule(); };
   virtual void ObserveEvent2() {
      Observe(eatEvents→EventUsage(acceptTerminate)); };
   virtual void ObserveEvent3() {
      Observe(eatEvents→EventUsage(terminate)); };
    void NoneCompeting() {
        ObserveEvent1();
        OneMoreCompetitor(); };

    void SomeCompeting() {
        ObserveEvent2();
        if (eatEvents→IsAccept()) {
            eatEvents→Schedule();
            OneMoreCompetitor(); };
        else if (eatEvents→IsTerminate())
            OneLessCompetitor();
    };

    void MaxCompeting() {
        ObserveEvent3();
        OneLessCompetitor();
    };
};
```

There are three states of a group of philosophers: a state in which no philosopher is competing for forks, a state in which some philosophers are competing for forks, and a state in which the maximum number of philosophers who can compete for forks are competing. The first state is the initial state of the group and is realized by the *NoneCompeting* CB method. The two subsequent states are realized by the *SomeCompeting* and *MaxCompeting* CB methods, respectively.

Unlike previous examples, the dining philosophers use CEOs not to block events. Instead, the group uses the CEO *eatEvents*, which is instantiated from the *CompositeEvent* class, to observe acceptance and termination events associated with the *Eat* methods in the different philosophers. Since the *Think* method plays no role as far as coordinating the philosophers is concerned, no events associated with that method are observed.

Two salient features of event observations in this group are, first, the termination of an *Eat* method need not be observed immediately after it is scheduled for execution. In fact, several *Eat* methods may be scheduled for execution consecutively before observing the termination of one of them. Thus, several separate EEOs are required to handle multiple observations of acceptance events. Second, the termination of *Eat* methods need not be observed in any specific order because it does not matter which philosopher finishes eating as long as one does. As a result, which specific EEO is used to observe a termination is not important in this group.

Both the above features are properties of a CEO: multiple acceptance events may be observed using a CEO (since it is a collection of EEOs) and termination events may be observed by a CEO in an unspecified order. As a result, a CEO is used to observe events which allows the abstract CE class to be independent of the number of philosophers which are coordinated (since the abstract CE class does not have to explicitly handle a specific number of EEOs).

The constructor of the abstract CE class expects a pointer to a CEO using which it initializes *eatEvents*. It also expects the number of components being coordinated because it initializes the *maxCompeting* variable to one less than the number of coordinated components. If only a single component is coordinated (a trivial case in which a group coordinator should not be used), *maxCompeting* is set to 1. The *numberCompeting* instance variable is initialized to zero and is used to keep track of the number of philosophers who are competing for forks at any point in time. The *IgnoreThink* method which is invoked from the constructor inhibits the posting of *Think* messages (the actual implementation of the method is provided by a concrete CE class).

The *Initiate* method is invoked to execute the initial CB method *NoneCompeting*. In that method, the *ObserveEvent1* method is invoked to observe the acceptance of an *Eat* message and to schedule the method for execution. Then, the *OneMoreCompetitor* method is invoked in which the following is done. First, *numberCompeting* is incremented by one. After that, if it is found that the maximum number of *Eat* methods have been scheduled, *ReplacementCB3* is invoked to initiate the *MaxCompeting* CB method. Otherwise, *ReplacementCB2* is invoked to initiate the *SomeCompeting* CB method.

In the *SomeCompeting* CB method, the *ObserveEvent2* method is invoked in which the *EventUsage* method is invoked on the CEO to set it for observing either an acceptance or a termination event. After an event is observed it is determined whether an acceptance or a termination event was observed. If an acceptance was observed, the method is scheduled for execution and the *OneMoreCompeting* method is invoked whose actions have been already described. If a termination was observed, the *OneLessCompetitor* method is invoked. In the latter method, *numberCompeting* is decremented by one. Then, if *numberCompeting* contains a zero, the *Initiate* method is invoked to activate the *NoneCompeting* CB method. Otherwise, *ReplacementCB2* is invoked to initiate the *SomeCompeting* CB method.

In the *MaxCompeting* CB method, the *ObserveEvent3* method is invoked in which the *EventUsage* method is invoked on the CEO to set it to observe a termination event. After a termination event is observed, the *OneLessCompetitor* method is invoked from the *MaxCompeting* CB method to update the number of philosophers who are eating.

The concrete CE class which may be used to instantiate a group of philosophers where each philosopher is instantiated from the class *Philosopher* is shown below.

```
class DiningPhilosophers : public CompetingComponents {
public:
   DiningPhilosophers(Philosopher* allPhilos, int numOfPhilos, CompositeEvent* cev):
         CompetingComponents(numOfPhilos, cev) {
      IgnoreThink(numOfPhilos, allPhilos); };
protected:
   void IgnoreThink(int numOfPhilos, Philosopher* allPhilos) {
      int i;
      for (i = 0; i < numOfPhilos; i++)
         Ignore(&Philosopher::Think, allPhilos[i]); };
};
```

The constructor of the concrete CE class expects a pointer to an array of all the philosophers in order to be able to inhibit the *Think* method in all of them from posting events. A group of five philosophers may be instantiated as follows:

```
Forks f1, f2, f3, f4, f5;
Philosopher* philos[5];
Philosopher* philos[0] = new Philosopher(&f1, &f5);
Philosopher* philos[1] = new Philosopher(&f1, &f2);
Philosopher* philos[2] = new Philosopher(&f2, &f3);
Philosopher* philos[3] = new Philosopher(&f3, &f4);
Philosopher* philos[4] = new Philosopher(&f4, &f5);
ElementaryEvent* ev1 = new ElementaryEvent(&Philosopher::Eat, philos[0]);
ElementaryEvent* ev2 = new ElementaryEvent(&Philosopher::Eat, philos[1]);
ElementaryEvent* ev3 = new ElementaryEvent(&Philosopher::Eat, philos[2]);
ElementaryEvent* ev4 = new ElementaryEvent(&Philosopher::Eat, philos[3]);
```

```
ElementaryEvent* ev5 = new ElementaryEvent(&Philosopher::Eat, philos[4]);
CompositeEvent* compEv = new CompositeEvent(ev1, ev2, ev3, ev4, ev5);
DiningPhilosophers* DPCE = new DiningPhilosophers(philos, 5, compEv);
philos[0]→RegisterCE(DPCE);
philos[1]→RegisterCE(DPCE);
philos[2]→RegisterCE(DPCE);
philos[3]→RegisterCE(DPCE);
philos[4]→RegisterCE(DPCE);
DPCE→Initiate();
```

Five philosopher objects are instantiated from the *Philosopher* class and are stored in the *philos* array. Next, five EEOs are instantiated to observe events related to the *Eat* method in the five philosopher objects. Using the latter EEOs, the CEO *compEv* is constructed. Next, a CE object is instantiated and is used as an argument in the invocation of the *RegisterCE* method in the five philosopher objects. Finally, the *Initiate* method is invoked on the CE object to execute the initial CB method.

3.2.5 Coordinating Components of a Full-Adder Circuit

Consider the state diagrams of two types of autonomous objects shown in Figures 3.4a and 3.4b. The object in Figure 3.4a has five public methods, *M1* through *M5*, and seven states, *A* through *G*. In state A, the object either accepts and executes an *M1* message and transits to state *B* or accepts and executes an *M2* message and transits to state *C*. The only transitions possible from states *B* and *C* are to state *D* through accepting and executing either an *M2* or an *M1* message, respectively. The only transition possible from state *D* is to state *E* by accepting and executing an *M3* message. In state *E*, the object either accepts and executes an *M4* message and transits to state *F* or accepts and executes an *M5* message and transits to state *G*. The only transitions possible from states *F* and *G* are to state *A* through accepting and executing either an *M5* or an *M4* message, respectively.

The object in Figure 3.4b has four public methods, *M1* through *M4*, and five states, *A* through *E*. In state A, the object either accepts and executes an *M1* message and transits to state *B* or accepts and executes an *M2* message and transits to state *C*. The only transitions possible from states *B* and *C* are to state *D* through accepting and executing either an *M2* or an *M1* message, respectively. The only transition possible from state *D* is to state *E* by

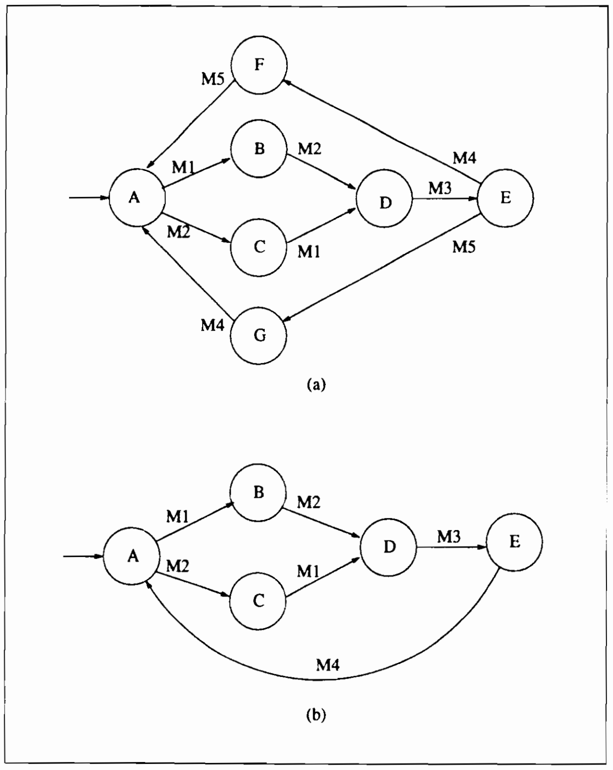

Figure 3.4: State diagram depicting the possible states and the available interface in each state of two types of autonomous objects.

accepting and executing an *M3* message. In state *E*, the object accepts and executes an *M4* message and transits to state *A*.

Using two objects of the type shown in Figure 3.4a and one object of the type shown in Figure 3.4b, one may construct a group which represents the two half-adder circuits and one OR-gate circuit, respectively, of a full-adder circuit that adds three bits to produce a sum bit and a carry bit.

The methods in Figure 3.4a may be mapped as follows. Methods *M1* and *M2* may stand for methods *SetInput1* and *SetInput2*, respectively, using which the two input bits of a half-adder circuit may be set. The value of the input bits are passed as arguments to the methods. Method *M3* may stand for a method *ApplyHalfAdder* using which the actual half-adder-circuit logic is applied to the input bits. Methods *M4* and *M5* may stand for methods *Output1* and *Output2*, respectively, using which the two output bits of a half-adder circuit may be read. Each of the latter methods expects an argument using which the output value is returned. The block diagram of a half-adder component is shown in Figure 3.5a.

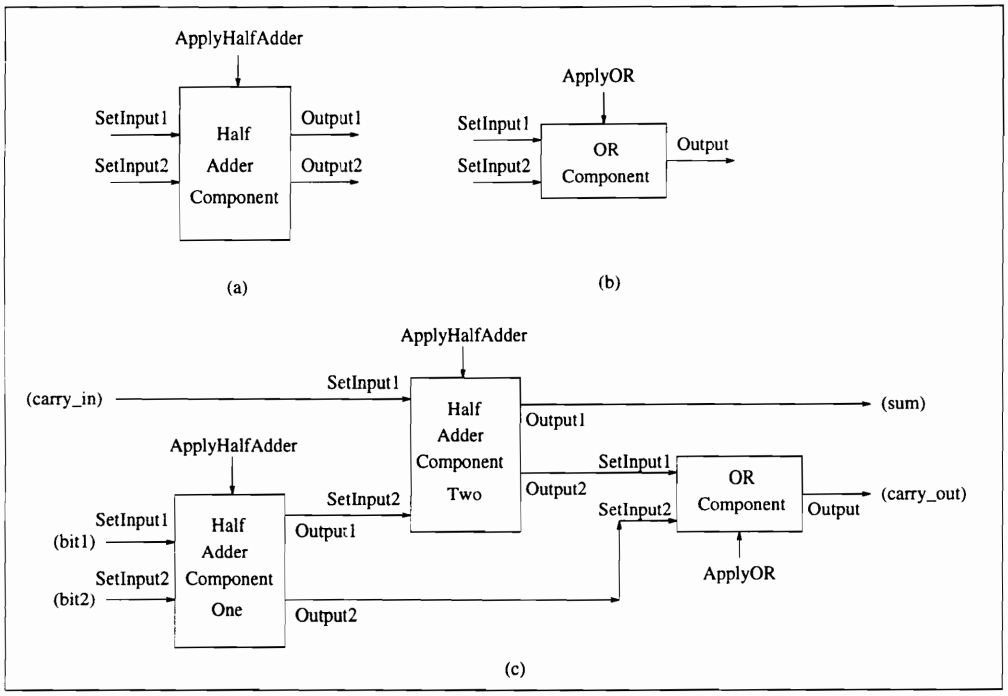

Figure 3.5: Block diagrams of (a) a half adder, (b) an OR gate, and (c) a full adder.

The methods in Figure 3.4b may be mapped as follows. Methods *M1* and *M2* may stand for methods *SetInput1* and *SetInput2*, respectively, using which the two input bits of an OR gate may be set. The value of the input bits are passed as arguments to the methods. Method *M3* may stand for a method *ApplyOR* using which the actual OR-gate logic is applied to the input bits. Method *M4* may stand for method *Output* using which the output bit of an OR gate may be read. The block diagram of an OR gate component is shown in Figure 3.5b.

The block diagram of a full-adder circuit is shown in Figure 3.5c. Two half-adder components and one OR gate is composed in a specific order to construct a full adder. The coordinating obligations of a coordinator for the three components of a full adder are the following. First, every method except for *SetInput1* of the half adders, *SetInput2* of half-adder one, *Output1* of half-adder two, and *Output* of the OR gate must be blocked so that they cannot be invoked by a client. Second, output bits must be transferred to the proper input ports of components since there is no explicit connection among them. Third, components must be excited in the right order so that the output-input dependencies are satisfied. For example, the second half-adder component must be supplied its second input bit only after the first half-adder component has computed its output.

The *MultiBitAdder* abstract CE class defined below realizes the coordination required by multiple components similar to those of a full-adder circuit.

```
class MultiBitAdder : public CoordinatingEnvironment {
public:
    MultiBitAdder(GroupComponent* ha1, GroupComponent* ha2,
      GroupComponent* or1, CompositeEvent* c1, CompositeEvent* c2,
      CompositeEvent* c3, int nOfInput = 3, int nOfOutput = 2) {
        halfAdder1 = ha1; halfAdder2 = ha2; OR = or1;
        inputEvents = c1; blockEvents = c2; outputEvents = c3;
        numOfInput = nOfInput; numOfOutput = nOfOutput;
        blockEvents→Block(); };
   virtual void Initiate() {
      Become(&MultiBitAdder::GetInputsApplyFAdder); };

protected:
   GroupComponent* halfAdder1, halfAdder2, OR;
   CompositeEvent* inputEvents, blockEvents, outputEvents;
   int numOfInput, numOfOutput;
   virtual void ReplacementCB2() {
      Become(&MultiBitAdder::ExtractOutputs); };
   virtual void ApplyFAdder() = 0;
   virtual void ObserveEvent1() {
      Observe(inputEvents→EventUsage(accept));
      inputEvents→Schedule(); };
   virtual void ObserveEvent2() {
      Observe(inputEvents→EventUsage(terminate)); };
   virtual void ObserveEvent3() {
      Observe(outputEvents→EventUsage(accept));
      outputEvents→Schedule(); };
   virtual void ObserveEvent4() {
      Observe(outputEvents→EventUsage(terminate)); };

   void GetInputsApplyFAdder() {
      int i;
      for (i = 0; i < numOfInput; i++)
         ObserveEvent1();
      for (i = 0; i < numOfInput; i++)
         ObserveEvent2();
      ApplyFAdder();
      ReplacementCB2() };
    void ExtractOutputs() {
        int i;
        for (i = 0; i < numOfOutput; i++)
            ObserveEvent3();
        for (i = 0; i < numOfOutput; i++)
            ObserveEvent4();
        Initiate(); };
};
```

There are two states of a full-adder group: a state in which all the input bits are collected and the logic is applied to them and a state in which the output produced is forwarded to the external environment. The first state is the initial state of the group and is realized by the *GetInputsApplyFAdder* CB method. The second state is realized by the *ExtractOutputs* CB method.

The group stores pointers to the two half-adder components in the instance variables *halfAdder1* and *halfAdder2*, respectively, and to the OR gate in the instance variable *OR*. These component pointers are required so that coordinating actions may be taken on them.

The group uses three CEOs to observe and block events. The *blockEvents* CEO is used to block the following methods: *Output1*, *Output2*, and *ApplyHalfAdder* of the first half-adder component, *SetInput2*, *Output2*, and *ApplyHalfAdder* of the second half-adder component, and *SetInput1*, *SetInput2*, and *ApplyOR* of the OR-gate component. These methods remain blocked throughout the lifetime of the CE object thereby hiding these methods from the clients of the full adder. The *inputEvents* CEO is used to observe the acceptance and termination of the three input providing methods, namely, *SetInput1* of the first half-adder component, and *SetInput1* and *SetInput2* of the second half-adder component. The use of the latter CEO enables client(s) to supply inputs to the full adder in any order. The *outputEvents* CEO is used to observe the acceptance and termination of the two output providing methods, namely, *Output1* of the second half-adder component and *Output1* of the OR-gate component. The use of the latter CEO enables client(s) to extract outputs from the full adder in any order.

The constructor of the abstract CE class expects pointers to the three components and pointers to the three event objects using which it initializes the instance variables. The constructor may also be supplied with two more optional arguments which are assigned default values if not supplied. The first, *nOfInput*, records the number of input bits which must be supplied to the group and using which the *numOfInput* instance variable is assigned. The second, *nOfOutput*, records the number of output bits produced by the group and using which the *numOfOutput* instance variable is assigned. These instance variables are used by the CB methods and by changing their values (and also the behavior of the group) one may use the abstract CE class to coordinate other groups with different numbers of inputs and outputs. A coordinating action taken from the constructor is to invoke the *Block* method on the *blockEvents* CEO to block several methods from the CPIs of the components.

The *Initiate* method is invoked to execute the initial CB method *GetInputsApplyFAdder*. In that method, the *ObserveEvent1* method is invoked three times consecutively to observe the acceptance of the three input methods of the full adder. This way of observing the acceptance events enables client(s) to provide input in an arbitrary order. After each acceptance is observed, the requested method is scheduled for execution in *ObserveEvent1*. After observing and scheduling the three acceptance events, the termination of the three scheduled methods is observed consecutively by invoking the *ObserveEvent2* method three times. After that, the *ApplyFAdder* method is invoked which applies the full-adder logic to the input bits and whose implementation is provided by a concrete CE class. Its function is discussed along with the the concrete CE class below. After the logic has been applied, the *ReplacementCB2* method is invoked to initiate the *ExtractOutputs* CB method.

The behavior of *ExtractOutputs* CB method is similar to that of the *GetInputsApplyFAdder* CB method except for applying the full-adder logic. The *ObserveEvent3* method is invoked two times consecutively to observe the acceptance of the two output methods of the full adder. This way of observing the acceptance events enables client(s) to extract output in an arbitrary order. After each acceptance is observed, the requested method is scheduled for execution in *ObserveEvent3*. After observing and scheduling the two acceptance events, the termination of the two scheduled methods is observed consecutively by invoking the *ObserveEvent4* method two times. After that, the *Initiate* method is invoked to initiate the initial CB method.

The concrete CE class which may be used to instantiate a group similar to a full-adder where each half adder is instantiated from the class *HalfAdder* and the OR gate is instantiated from the class *ORGate*, is shown below.

```
class FullAdder : public MultiBitAdder {
public:
    FullAdder(HalfAdder* h1, HalfAdder* h2, ORGate* or,
      CompositeEvent* c1, CompositeEvent* c2, CompositeEvent* c3) :
      MultiBitAdder( (GroupComponent*) h1, (GroupComponent*) h2,
         (GroupComponent*) or, c1, c2, c3) { };
protected:
   void ApplyFAdder() {
      halfAdder1→BlockComponent();
      ((HalfAdder*) halfAdder1)→ApplyHalfAdder();
      int out1 = ((HalfAdder*) halfAdder1)→Output1();
      int out2 = ((HalfAdder*) halfAdder1)→Output2();
      halfAdder1→UnblockComponent();
      halfAdder2→BlockComponent();
      ((HalfAdder*) halfAdder2)→SetInput2(out1);
      ((HalfAdder*) halfAdder2)→ApplyHalfAdder();
      int out3 = ((HalfAdder*) halfAdder2)→Output2();
      halfAdder2→UnblockComponent();
      OR→BlockComponent();
      ((ORGate*) OR)→SetInput1(out3);
      ((ORGate*) OR)→SetInput2(out2);
      ((ORGate*) OR)→ApplyOR();
      OR→UnblockComponent();
   };
};
```

The constructor of the *FullAdder* class expects pointers to the components and pointers to the event objects using which the constructor of the abstract CE class is invoked.

The *FullAdder* class also provides an implementation of the *ApplyFAdder* coordinating-action method. In that method, three sets of actions are taken on the three components. First, after blocking the RH in the first half-adder component, the half-adder logic is applied, the output produced is recorded in two local variables, *out1* and *out2*, and the RH is unblocked. Second, after blocking the RH in the second half-adder component, the bit recorded in *out1* is provided as the second input to it, the half-adder logic is applied, the output produced is recorded in a local variable, *out3*, and the RH is unblocked. Third, after blocking the RH in the OR-gate component, the bits recorded in *out3* and *out2* are provided as the two inputs to it, the OR-gate logic is applied, and the RH is unblocked.

A full-adder may be instantiated as follows:

```
XOR* x1 = new XOR();
AND* a1 = new AND();
HalfAdder* ha1 = new HalfAdder(x1, a1);
XOR* x2 = new XOR();
AND* a2 = new AND();
HalfAdder* ha2 = new HalfAdder(x2, a2);
ORGate* or = new ORGate();
ElementaryEvent* e1 = new ElementaryEvent(&HalfAdder::SetInput1, ha1);
ElementaryEvent* e2 = new ElementaryEvent(&HalfAdder::SetInput2, ha1);
ElementaryEvent* e3 = new ElementaryEvent(&HalfAdder::SetInput1, ha2);
CompositeEvent inputEvents = new CompositeEvent(e1, e2, e3);
ElementaryEvent* e4 = new ElementaryEvent(&HalfAdder::Output1, ha1);
ElementaryEvent* e5 = new ElementaryEvent(&HalfAdder::Output2, ha1);
ElementaryEvent* e6 = new ElementaryEvent(&HalfAdder::ApplyHalfAdder, ha1);
ElementaryEvent* e7 = new ElementaryEvent(&HalfAdder::SetInput2, ha2);
ElementaryEvent* e8 = new ElementaryEvent(&HalfAdder::Output2, ha2);
ElementaryEvent* e9 = new ElementaryEvent(&HalfAdder::ApplyHalfAdder, ha2);
ElementaryEvent* e10 = new ElementaryEvent(&ORGate::SetInput1, or);
ElementaryEvent* e11 = new ElementaryEvent(&ORGate::SetInput2, or);
ElementaryEvent* e12 = new ElementaryEvent(&ORGate::ApplyOR, or);
CompositeEvent blockEvents =
   new CompositeEvent(e4, e5, e6, e7, e8, e9, e10, e11, e12);
ElementaryEvent* e13 = new ElementaryEvent(&HalfAdder::Output1, ha2);
ElementaryEvent* e14 = new ElementaryEvent(&ORGate::Output1, or);
CompositeEvent outputEvents = new CompositeEvent(e13, e14);
FullAdder* fa = new FullAdder(ha1, ha2, or, inputEvents, blockEvents, outputEvents);
ha1→RegisterCE(fa);
ha2→RegisterCE(fa);
or→RegisterCE(fa);
fa→Initiate();
```

Two half-adder components are instantiated using the *HalfAdder* class. The constructor of the latter class expects pointers to an XOR-gate component and an AND-gate component as arguments. The OR-gate component is instantiated using the *ORGate* class. All these classes are assumed to be pre-defined subclasses of the *GroupComponent* class. Finally, three CEOs are instantiated using several EEOs, the CE object is instantiated, the half adders and the OR-gate are made to register the CE object, and the *Initiate* method is invoked on the CE object to execute the initial CB method.

3.2.6 Composing Full Adders: A Ripple-Carry Adder

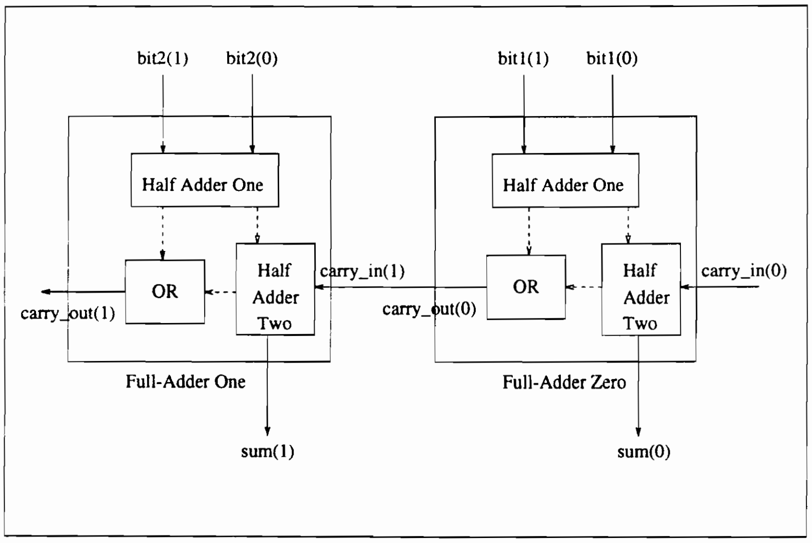

Figure 3.6: Block diagram of a two-stage ripple-carry adder.

Figure 3.6 shows the composition of two full adders to realize a ripple-carry adder which can add two two-bit numbers, "bit2(0)bit1(0)" and "bit2(1)bit1(1)", and produce the sum "sum(1)sum(0)" and carry "carry_out(1)". The "carry_in(0)" bit is assumed to be a constant value of "0". The connections between the components in each full adder is shown in dotted arcs since they are invisible to the external environment.

The only action necessary to connect the two full adders is to transfer the "carry_out(0)" output bit of full-adder zero (once it is evaluated) to the "carry_in(1)" input bit of full-adder one. The *ConnectFAdders* concrete CE class, defined below, captures the coordination required for composing two full-adder CE objects. Note that due to the very special purpose served by the latter class, it is not defined as an abstract CE class and it is derived directly as a subclass of the *CoordinatingEnvironment* class.

```
class ConnectFAdders : public CoordinatingEnvironment {
public:
    ConnectFAdders(HalfAdder* ha, ORGate* o) {
      inputHalfAdder = ha; outputOR = o; };
  virtual void Initiate() {
      become(&ConnectFAdders::Link); };
protected:
    HalfAdder* inputHalfAdder;
    ORGate* outputOR;
    void Link() {
        int carryOut;
        Cbox* cb = new Cbox();
        Message* m1 = new Message(thisThread, &ORGate::Output, cb);
        m1→SendUntilAccepted(outputOR);
        cb→Receive(carryOut);
        Message* m1 = new Message(thisThread, &HalfAdder::SetInput1, carryOut);
        m1→SendUntilAccepted(inputHalfAdder);
        Initiate(); };
};
```

There is only one state of the group coordinated by CE objects instantiated from the above class: the state in which an output value is extracted from one component and is provided as input to the other component. This state is captured by the *Link* CB method.

When a CE object is instantiated from the above class to coordinate the activities of two full-adder components of a ripple-carry adder, the OR-gate of the full-adder from which the output bit is extracted and the half-adder of the full-adder to which that bit is supplied are supplied as arguments to the constructor.

A salient feature of the above group is that the coordinating actions are not triggered by the observation of events in components. Instead, in the *Link* method, a message is sent to the OR-gate component requesting the execution of the *Output* method. The OR-gate accepts and executes the latter method only when full-adder zero has produced its output. Note that the CB method cannot block the RH in the OR-gate component and execute *Output* because it does not know when the output is produced by the full adder. Hence it must await for *Output* to appear in the CPI of the OR-gate. After obtaining the output of the OR-gate component, a message is sent to the half-adder component requesting the execution of the *SetInput1* method and passing the output value as an argument. Once the half-adder accepts and executes the latter method, the CB method invokes the *Initiate* method to repeat the above cycle.

The above CB method also demonstrates the use of a Cbox to exchange data values among components. A Cbox is instantiated from the *Cbox* class (a subclass of the *Object* class). A pointer to a Cbox, stored in *cb*, is sent in the request message to the ORgate component. After the latter message is accepted and executed, *Link* extracts the return value of the *Output* method invocation by invoking the *Receive* method on the Cbox *cb*. The *Receive* method expects as argument a variable passed by reference in which the extracted value is returned. Each invocation of *Receive* extracts one data value from the Cbox (permanently) and assigns it to the argument variable.

3.2.7 Coordinating Components of an Elevator System

An elevator system comprises of a collection of elevator cars, several button panels using which cars are requested from multiple floors, and displays on each floor which inform the current locations of the cars. A car is comprised of an internal display using which the current location of the car is displayed and a button panel using which requests to open the door, to close the door, and to move the car to a destination floor are made.

For the purpose of this example, an elevator system, serving K floors, is considered to be the composition of the following components:

- Light-panel objects, one for each floor and K for each car. Using these K objects, the current position of a car is displayed on all the floors.

- Light-panel objects, one for each car. Using this object, the position of a car is displayed inside the car.

- Button-panel objects, one for each car. Using such an object, requests to move a car to a destination floor is made from inside the car.

- A global request-store object, one for an elevator system. This object stores all the requests for cars which are made from all the K floors.

- Up/Down button-panel objects, one for each floor. Passengers request cars using the buttons of this panel from a floor.

- Car objects, one or more.

The responsibility of coordinating the composition of the above components is divided into three different types of CE objects as shown in Figure 3.7. In Figure 3.7a, a CE object is shown which coordinates K light-panel objects on the K floors and the light-panel object

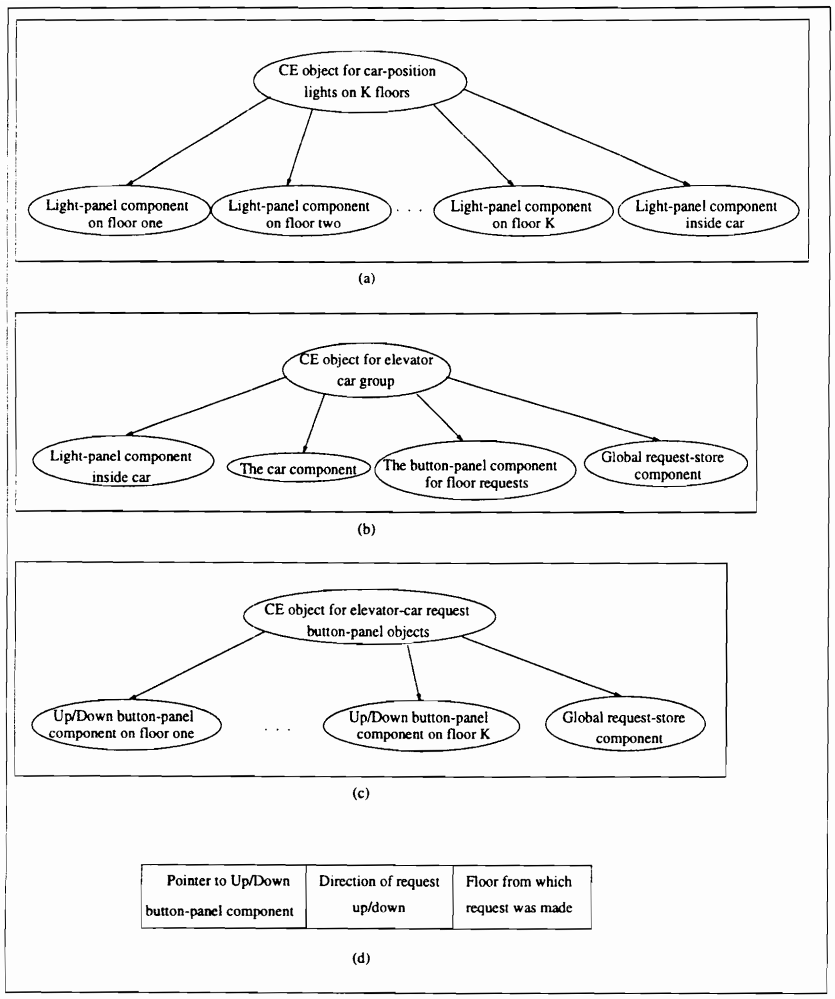
CE object for car-position lights on K floors
Light-panel component on floor one
Light-panel component on floor two
. . .
Light-panel component on floor K
Light-panel component inside car
(a)
CE object for elevator car group
Light-panel component inside car
The car component
The button-panel component for floor requests
Global request-store component
(b)
CE object for elevator-car request button-panel objects
Up/Down button-panel component on floor one
. . .
Up/Down button-panel component on floor K
Global request-store component
(c)
Pointer to Up/Down button-panel component
Direction of request up/down
Floor from which request was made
(d)

Figure 3.7: (a) The group displaying the position of a car on all the floors. (b) The group involving an elevator car which transfers passengers between floors. (c) The group using which passengers request cars from a floor. (d) The structure of a request stored in the global request-store object.

inside a car. For each car, there is a CE object of this type. The coordination responsibility of this CE object is to observe the request to change the display in the light-panel object inside the car, to determine the floor corresponding to which the light is turned on in the latter display, and to turn on the light corresponding to the same floor in all the K display objects thereby displaying the current location of the car on all floors.

In Figure 3.7b, a CE object is shown which coordinates a light-panel object inside a car, the car object, the button-panel object inside the car, and the global request-store object. There is one such CE object for each car in the elevator system. The coordination responsibility of this CE object is to control the opening and closing of the doors of the car, to collect requests from both the button-panel object inside the car and the global request-store object, to decide which floor to travel to next, and to move the car to a destination floor.

In Figure 3.7c, a CE object is shown which coordinates K up/down button-panel objects on K floors and the global request-store object. There is only one CE object of this type for an elevator system. The coordination achieved by this CE object is to observe the pressing of an up or down button on a panel, construct a data record having the structure shown in Figure 3.7d, and enqueue it at the global request-store. The data record stores a pointer to the up/down button-panel object in which the acceptance event was observed, the direction of travel that was requested, and the floor from which the request was made.

The behavior of the components and the CE classes for each of the above CE objects are described in the following sections. For the sake of brevity, the state diagrams of components are shown with the names of the methods supported by them instead of general purpose names like *M1*, *M2*, etcetera.

3.2.7.1 The Car-Position-Lights Group

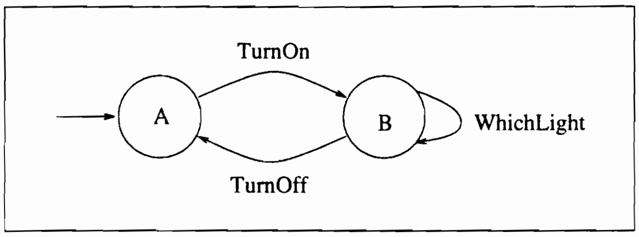

Figure 3.8: The state-diagram of a light-panel component.

The group displaying the position of a car on each floor consists of (K+1) light-panel components, each of which has the behavior shown in Figure 3.8. The component has three public methods and two states, *A* and *B*, where *A* is the initial state. The method *TurnOn*, available in the initial state, may be used to turn on a specific light on the panel. Each light-panel object has K private light objects which are identified by the numbers 1 through K. The particular light which must be turned on is provided as an argument to the method *TurnOn*. The method *WhichLight*, available in state *B*, returns the number of the light which is currently on. The method *TurnOff*, available in state *B*, turns off the light whose number is supplied as an argument to it.

The K light-panel components on the floors for each elevator car do not post any events to the CE object (they are acquaintances of the CE object but they do not register the CE object). Moreover, in the elevator system, these K components are known only to the CE object which coordinates them. Thus, when the CE object intends to invoke a method in one of these K components, it does not have to disable and enable the request handlers since no other clients may use the components.

The light-panel inside a car is shared between the CE object displaying the position of a car and the CE object which controls the movement of that car. These two CE objects are the only objects in an elevator system which interact with the light panel inside a car. The light-panel component registers the CE object displaying car position and is just an acquaintance of the CE object controlling car movement. The former CE object blocks the *WhichLight* method in the light-panel component and allows it to post events associated with the *TurnOn* and *TurnOff* methods only. As a result, the CE object controlling car movement may send only *TurnOn* and *TurnOff* messages to the light-panel inside a car.

The *MultiPanelDisplay* abstract CE class defined below realizes the coordination required by multiple components similar to those of multiple light-panel objects.

```
class MultiPanelDisplay : public CoordinatingEnvironment {
public:
   MultiPanelDisplay(GroupComponent* fp, GroupComponent* cp,
     ElementaryEvent* e1, ElementaryEvent* e2, ElementaryEvent* e3, int nf) {
        floorPanels = fp; carPanel = cp;
        blockEvent = e1; turnOnLight = e2; turnOffLight = e3;
        numOfFloors = nf; lightNumber = 1;
        blockEvent→Block(); };
   virtual void Initiate() {
     Become(&MultiPanelDisplay::UpdatePanels); };


protected:
   GroupComponent* floorPanels, carPanel;
   ElementaryEvent* blockEvent, turnOnLight, turnOffLight;
   int numOfFloors, lightNumber;
   virtual void UpdateFloorPanelsCA() = 0;
```

```
    virtual void ObserveEvent1() {
        Observe(turnOffLight);
        turnOffLight→AwaitTermination(THISCE); };
    virtual void ObserveEvent2() {
        Observe(turnOnLight);
        turnOnLight→AwaitTermination(THISCE); };

    void UpdatePanels() {
        ObserveEvent1();
        ObserveEvent2();
        UpdateFloorPanelsCA();
        Initiate(); };
};
```

There is one state of the light-panels group: a state in which the turning-off and the subsequent turning-on of lights in the display inside the car is observed, the number of the light which is on in the latter display is obtained, and all the panels on the K floors are updated to reflect the current position of the car. This state is realized by the *UpdatePanels* CB method.

The group stores pointers to the K light-panel components on the floors in the instance variable *floorPanels* and a pointer to the light-panel component inside a car in the instance variable *carPanel.* The total number of floors served by the elevator system is stored in the instance variable *numOfFloors* and the number of the light that is on in the displays on the floors (denoting the current position of the elevator car) is stored in the instance variable *lightNumber.*

The group uses three EEOs to block and observe events. The *blockEvent* EEO is used to block the *WhichLight* method of the light-panel inside the car throughout the lifetime of the CE object thereby permanently removing the method from the CPI of the component. The *turnOnLight* and *turnOffLight* EEOs are used to observe the acceptances and terminations of the *TurnOn* and *TurnOff* methods, respectively, in the light panel component inside the car.

The constructor of the class expects as arguments pointers to the (K+1) components, pointers to the three event objects, and an integer representing the number of floors. Using all the supplied argument values the constructor initializes the instance variables. The *lightNumber* instance variable is assigned a value of 1 in the constructor indicating that the initial position of a car must be on floor one and on all the (K+1) light-panel components, light number one must be lit. A coordinating action taken from the constructor is to invoke the *Block* method on the *blockEvent* EEO to block the *WhichLight* method in the light panel stored inside the car.

The *Initiate* method is invoked to execute the initial CB method *UpdatePanels.* In the *UpdatePanels* method, the *ObserveEvent1* method is invoked to observe the acceptance and the termination of the *TurnOff* method in the light panel inside the car. After *ObserveEvent1* terminates, the *ObserveEvent2* method is invoked to observe the acceptance and the termination of the *TurnOn* method in the light panel inside the car. After *ObserveEvent2* terminates, the *UpdateFloorPanelsCA* coordinating-action method is invoked which updates the light panels on the K floors. The coordinating-action method is realized and discussed below. After *UpdateFloorPanelsCA* terminates, the *Initiate* method is invoked to initiate the *UpdatePanels* CB method once again.

The concrete CE class which may be used to instantiate a group similar to the (K+1) light panels in an elevator system where each light panel is instantiated from the class *LightPanel*, is shown below.

```
class ElevatorPositionDisplays : public MultiPanelDisplay {
public:
    ElevatorPositionDisplays(LightPanel* fp, LightPanel* cp,
      ElementaryEvent* e1, ElementaryEvent* e2, ElementaryEvent* e3, int nf) :
      MultiPanelDisplay( (GroupComponent*) fp, (GroupComponent*) cp,
        e1, e2, e3, nf) { };
protected:
   void UpdateFloorPanelsCA() {
      ((LightPanel*) carPanel)→BlockComponent();
      int lightOn = ((LightPanel*) carPanel)→WhichLight();
      ((LightPanel*) carPanel)→UnblockComponent();
      int j;
      for (j = 0; j < numOfFloors; j++) {
         ((LightPanel*) floorPanels[j])→TurnOff(lightNumber);
         ((LightPanel*) floorPanels[j])→TurnOn(lightOn); };
      lightNumber = lightOn;
   };
};
```

The constructor of the *ElevatorPositionDisplays* class expects pointers to the components and pointers to the event objects using which the constructor of the abstract CE class is invoked.

The above class also provides an implementation of the *UpdateFloorPanelsCA* method. In that method, two sets of actions are taken on the components. First, after blocking the RH in the *carPanel* component, the number of the light which is on in the display inside the car is retrieved and stored in the local variable *lightOn*, and the RH is unblocked. Second, for each floor, the light which is lit in the light panel on that floor is turned off (using the *TurnOff* method and passing *lightNumber* as an argument to it) and the light number corresponding to the new position of the car is lit in that panel (using the *TurnOn* method and passing *lightOn* as an argument to it). Note that blocking and unblocking of the RH is not required when invoking the methods of the K light panels because these components are acquaintances of only this CE object.

3.2.7.2 The Up-Down Request-Panels Group

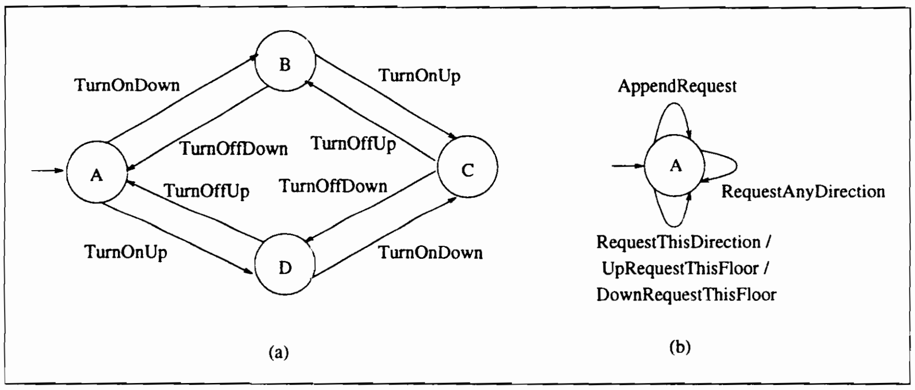

Figure 3.9: (a) The state-diagram of an up-down button-panel component.
(b) The state diagram of the global request-store component.

The group ensuring that a request for a car made by a passenger is conveyed to the relevant car consists of K up-down button-panel components and the global request-store component. The behavior of an up-down button-panel component is shown in Figure 3.9a. It has four public methods and four states, *A* through *D*, where *A* is the initial state. The methods *TurnOnUp* and *TurnOnDown*, available in the initial state, may be used to request a car for travel in either upward or downward directions, respectively. In state B, either a previous downward-travel request may be acknowledged by using the *TurnOffDown* method or a new upward-travel request may be registered using the *TurnOnUp* method. In state D, either a previous upward-travel request may be acknowledged by using the *TurnOffUp* method or a new downward-travel request may be registered using the *TurnOnDown* method. In state C, no new travel requests may be registered since both requests are pending. The only event possible in state C is the acknowledgment of either a pending upward or a pending downward travel request.

The behavior of the global request-store component is shown in Figure 3.9b. It has only one state A which is the initial state and it has five public methods which are all available in state A. The global request-store component is nothing but a queue with certain special search capabilities. It enqueues data-items which have the structure shown in Figure 3.7d.

Among the methods of the request-store component, the *AppendRequest* method may be used to enqueue a new data item by supplying a pointer to a button-panel object, a direction value (1 for up and 2 for down), and a floor number as arguments. The *RequestAnyDirection* method may be used to extract an item from the store which contains a floor number that is closest to a floor number supplied as an argument to the method. If m is the floor number supplied, then the smallest of all floor numbers greater than m, *m1* say, and the greatest of all floor numbers smaller than m, *m2* say, are determined. Then, out of *m1* and *m2*, the number which yields the least value when $\mid m - m1 \mid$ and $\mid m - m2 \mid$ are computed, satisfy the search criterion. If there is a tie, then any one of the values is returned.

The *RequestThisDirection* method of the request-store component may be used to extract an item which contains a floor number that is closest to a floor number supplied as an argument to the method but which is either lesser than or greater than (as the case may be) a second number that is also supplied as an argument. If m and k are the first and second argument values, respectively, and $k > m$, then the smallest of all floor numbers greater than m but smaller than k satisfies the search criterion. But, if $k < m$, then the greatest of all floor numbers smaller than m but greater than k satisfies the search criterion.

The *UpRequestThisFloor* method of the request-store component may be used to determine whether the store has an entry that is a request to go up from the floor that is supplied as an argument to the method. The *DownRequestThisFloor* method of the request-store component may be used to determine whether the store has an entry that is a request to go down from the floor that is supplied as an argument to the method.

An up-down button panel object posts events associated with only the *TurnOnDown* and the *TurnOnUp* methods and the CE object permanently blocks the *TurnOffDown* and the *TurnOffUp* methods. This ensures that the two latter methods are never available in the CPI of the up-down buttons and are therefore hidden from the clients who use those buttons.

The global request-store component is shared between the CE object coordinating the up-down request buttons and the CE objects which control the movement of all the cars in the elevator system. These CE objects are the only objects which interact with the global store in an elevator system. The global store does not post events to any of these CE objects (it is an acquaintance of all the CE objects but it does not register any one of them).

The *MultipleTwoButtonPanels* abstract CE class defined below realizes the coordination required by multiple components similar to those described above.

```
class MultipleTwoButtonPanels : public CoordinatingEnvironment {
public:
    MultipleTwoButtonPanels(GroupComponent* bp, AsynchronousObj* rs,
      ElementaryEvent* e1, ElementaryEvent* e2,
      ElementaryEvent* e3, ElementaryEvent* e4, int nf) {
        floorButtonPanels = bp; requestStore = rs;
        upRequest = e1; downRequest = e2;
        blockTurnOffUp= e3; blockTurnOffDown= e4; numOfFloors = nf;
        blockTurnOffUp→Block();
        blockTurnOffDown→Block(); };
   virtual void Initiate() {
      Become(&MultipleTwoButtonPanels::ObserveAppendRequests); };


protected:
   GroupComponent* floorButtonPanels;
   AsynchronousObj* requestStore;
   ElementaryEvent* blockTurnOffUp, blockTurnOffDown, upRequest,
       downRequest;
   int numOfFloors;
   int DetermineFloor(GroupComponent* floorButton) {
      int i;
      for (i = 0; i < numOfFloors; i++)
```

```
        if (floorButton == floorButtonPanels[i])
            return (i + 1);
    };
    virtual void AppendRequestCA(GroupComponent*, int, int) = 0;
    virtual Event* ObserveEvent1() {
        return Observe(upRequest, downRequest); };

    void ObserveAppendRequests() {
        int direction, reqFromFloor;
        Event* whichEvent = ObserveEvent1();
        GroupComponent* floorButton = whichEvent→WhichComponent();
        whichEvent→AwaitTermination(THISCE);
        if (whichEvent == upRequest)
            direction = 1;
        else direction = 2;
        reqFromFloor = DetermineFloor(floorButton);
        AppendRequestCA(floorButton, direction, reqFromFloor);
        Initiate(); };
};
```

There is one state of the up-down button-panels group: a state in which an upward or downward travel request is observed and an entry corresponding to the request is enqueued at the global request store. This state is realized by the *ObserveAppendRequests* CB method.

The group stores pointers to the K up-down button-panel components on the floors in the instance variable *floorButtonPanels* and a pointer to the global request-store component in the instance variable *requestStore*. The total number of floors served by the elevator system is stored in the instance variable *numOfFloors*.

The group uses four multi-constituent EEOs to block and observe events. Each EEO stores pointers to K up-down button-panel components. The *blockTurnOffUp* EEO is used to block the *TurnOffUp* method in all the K up-down button-panel components. The *blockTurnOffDown* EEO is used to block the *TurnOffDown* method in all the K up-down button-panel components. The *upRequest* and the *downRequest* EEOs are used to observe the acceptance and termination of the *TurnOnUp* and the *TurnOnDown* methods, respectively, in the up-down button-panel components.

The constructor of the class expects as arguments pointers to the K up-down button-panel components, a pointer to the request-store component, pointers to the four event objects, and an integer representing the number of floors. Using these the constructor initializes the instance variables. Two coordinating actions taken from the constructor are to invoke the *Block* methods on the *blockTurnOffUp* and *blockTurnOffDown* EEOs to block the *TurnOffUp* and *TurnOffDown* methods, respectively.

The *Initiate* method is invoked to execute the initial CB method *ObserveAppendRequests*. In that CB method, the *ObserveEvent1* method is invoked to observe the acceptance of either a *TurnOnUp* or a *TurnOnDown* message in one of the K up-down button-panel components. The EEO using which an event is observed is returned by *ObserveEvent1* and is stored in the variable *whichEvent*. The button using which the observed request was made is extracted from the EEO using the *WhichComponent* method and stored in the variable *floorButton*. The requested method is then scheduled for execution and its termination awaited. After the termination is observed, the direction of the request is determined based on which event object was returned by *ObserveEvent1* and stored in the variable *direction* (1 for up and 2 for down). Then the *DetermineFloor* method is invoked to determine the floor number from which the request was made and it is stored in the *reqFromFloor* variable. The latter method accepts a pointer to a up-down button-panel component as argument, searches it in the *floorButtonPanels* array for a match, and returns a number based on the position of the match in the array. After *DetermineFloor* terminates, the *AppendRequestCA* coordinating-action method is invoked which is responsible for enqueueing a request item at the global request-store component. The coordinating-action method is realized and discussed below. After *AppendRequestCA* terminates, the *Initiate* method is invoked to initiate the *ObserveAppendRequests* CB method once again.

The concrete CE class which may be used to instantiate a group similar to the K up-down request-button panels in an elevator system where each button panel is instantiated from the class *UpDownButtonPanel* and the request store is instantiated from the class *RequestStore*, is shown below.

```
class ElevatorCarRequestPanels : public MultipleTwoButtonPanels {
public:
    ElevatorCarRequestPanels(UpDownButtonPanel* bp, RequestStore* rs,
        ElementaryEvent* e1, ElementaryEvent* e2,
        ElementaryEvent* e3, ElementaryEvent* e3, int nf) :
```

```
    MultipleTwoButtonPanels( (GroupComponent*) bp, (AsynchronousObj*) rs,
       e1, e2, e3, e4, nf) { };
protected:
  void AppendRequestCA(GroupComponent* fbutton, int dir, int floornum) {
    Message* m1 =
      new Message(thisThread, &RequestStore::AppendRequest, fbutton, dir, floornum);
    m1→SendUntilAccepted(requestStore);
  };
};
```

The constructor of the *ElevatorCarRequestPanels* class expects pointers to the components and pointers to the event objects using which the constructor of the abstract CE class is invoked.

The above class also provides an implementation of the *AppendRequestCA* method. In that method, a message object is constructed and sent to the global request-store component. The message requests the execution of the *AppendRequest* method and provides a pointer to a up-down button-panel component, a direction value, and the number of a floor as arguments. The request store uses the supplied argument values to construct an item similar to the one shown in Figure 3.7d and appends it to its internal queue.

3.2.7.3 The Elevator-Car Group

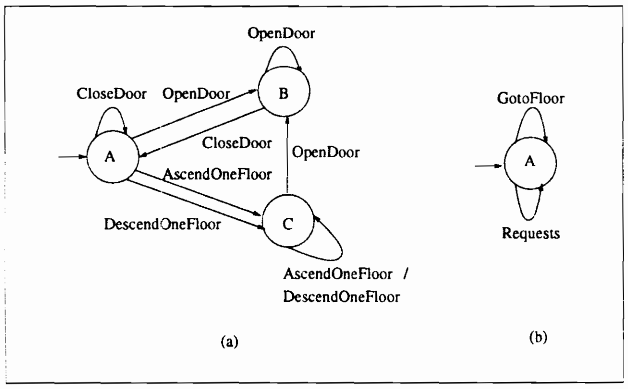

Figure 3.10: (a) The state-diagram of an elevator-car component. (b) The state diagram of a button-panel component.

The group controlling the movement of an elevator car consists of four components: a light-panel displaying the current position of the car inside the car, the elevator car, a button panel using which destination floors are requested from inside the car, and the global request-store. The behaviors of a light-panel and the request-store components have been described before. Figure 3.10a shows the behavior of an elevator-car component. It has four public methods and three states, A through C, where A is the initial state. In the initial state, an elevator car is assumed to be stationary, its doors closed, and the four following methods are available for invocation. The *OpenDoor* method may be used to open the doors of the car which causes a transition to state B. The *CloseDoor* method may be used to close the doors and remain in the initial state. The *AscendOneFloor* and *DescendOneFloor* methods may be used to move the car up or down, one floor at a time, respectively. In state B, either the *OpenDoor* method may be invoked repeatedly without causing any state transition or the *CloseDoor* method may be invoked which causes a transition back to the initial state. In state C, the car may be moved up or down by repeated applications of the *AscendOneFloor* and *DescendOneFloor* methods, respectively, or the *OpenDoor* method may be used to open the doors. Two important features of the behavior of a car are, first, it moves one floor at a time in either direction and, second, it may move only if its doors are closed.

The behavior of a button-panel component is shown in Figure 3.10b. It has only one state A which is the initial state and it has two public methods which are available in state A. A button-panel component is nothing but a queue which stores the numbers of the buttons which are depressed using the *GotoFloor* method. The number of the button (an integer from 1 through K) is supplied as an argument to the *GotoFloor* method which adds it to the queue. The *Requests* method may be used to extract the contents of the internal queue. The stored numbers are copied into an array which is supplied as an argument to the method and after the extraction is over, the contents of the queue are erased.

The button-panel, the light-panel, and the global request-store components do not post any events to the CE object (they are acquaintances of the CE object but do not register the CE object). Moreover, the CE object permanently blocks the *AscendOneFloor* and *DescendOneFloor* methods in a car object so that they are never available in the CPI of the car component. This enables the CE object to have exclusive control over the movements of the car.

The *Conveyor* abstract CE class defined below realizes the coordination required by multiple components similar to those which form an elevator car.

```
class Conveyor : public CoordinatingEnvironment {
public:
    Conveyor(GroupComponent* lp, AsynchronousObj* car, AsynchronousObj* bp,
      AsynchronousObj* rs, CompositeEvent* c1, CompositeEvent* c2) {
        lightPanel = lp; theCar = car;
        buttonPanel = bp; requestStore = rs; pendingRequests = Null;
        blockOpenCloseDoor = c1; blockAscendDescend = c2;
        currentFloor = 1; nextFloor = 0;
        requestedDirection = 0; moveNumFloors = 0;
        currentTravelDirection = 0; upDownButton = Null;
        blockAscendDescend→Block(); };
   virtual void Initiate() {
      Become(&Conveyor::AwaitRequests); };


protected:
   GroupComponent* lightPanel, upDownButton;
   AsynchronousObj* theCar, buttonPanel, requestStore;
   RequestList* pendingRequests;
   int currentFloor, nextFloor, requestedDirection, moveNumFloors;
   int currentTravelDirection;
   CompositeEvent* blockOpenCloseDoor, blockAscendDescend;
   virtual void UpdateRequests(RequestList*);
   virtual int DecideNextMove();
   virtual void ReplacementCB2() {
      Become(&Conveyor::PickupDeliverPassengers); };
   virtual void LookForRequestsCA() = 0;
   virtual void CloseDoorsCA() = 0;
   virtual void AscendFloorsCA() = 0;
   virtual void DescendFloorsCA() = 0;
   virtual void OpenDoorsCA() = 0;
   virtual void ResetUpDownButtonCA() = 0;
   virtual void DetermineNextMoveCA() = 0;


   void AwaitRequests() {
      LookForRequestsCA();
      if (nextFloor) {
         blockOpenCloseDoor→Block();
         ReplacementCB2(); };
      else Initiate(); };
    void PickupDeliverPassengers() {
        CloseDoorsCA();
        if (nextFloor > currentFloor) {
            currentTravelDirection = 1;
            moveNumFloors = nextFloor - currentFloor;
            AscendFloorsCA();
            currentFloor = currentFloor + moveNumFloors; };
        else if (nextFloor < currentFloor) {
            currentTravelDirection = 2;
            moveNumFloors = currentFloor - nextFloor;
            DescendFloorsCA();
            currentFloor = currentFloor - moveNumFloors; };
        blockOpenCloseDoor→Unblock();
        OpenDoorsCA();
        ResetUpDownButtonCA();
        DetermineNextMoveCA();
        blockOpenCloseDoor→Block();
        CloseDoorsCA();
        if (nextFloor)
            ReplacementCB2();
        else Initiate(); };
    };
};
```

There are two states of the elevator-car group: a state in which the car waits for the the enqueueing of a request in the global request-store and a state in which it responds to such a request by picking up passengers and taking them to destination floors. The former state is realized by the *AwaitRequests* CB method and is the initial state of the group and the latter state is realized by the *PickupDeliverPassengers* CB method.

The group stores a pointer to the light-panel component inside the car in the instance variable *lightPanel*, a pointer to the elevator car in the instance variable *theCar*, a pointer to the button-panel inside the car used to make travel requests in the instance variable *buttonPanel*, and a pointer to the global request-store component in the instance variable *requestStore*. The requests which are yet to be serviced by an elevator-car are stored in the *pendingRequests* instance variable. Before moving to a destination floor, the instance variable *nextFloor* stores the destination floor number and the instance variable *moveNumFloors* stores the number of floors which must be traveled. While moving to a destination floor, the instance variable *currentFloor* stores the last floor number on which the car was stationary and the current direction in which the car is moving is stored in the *currentTravelDirection* instance variable. When servicing a request to arrive at a certain floor, the *requestedDirection* instance variable stores the direction of travel made in the request and the *upDownButton* instance variable stores a pointer to the up-down button-panel component using which the request was made.

The group uses two CEOs, each containing two EEOs, to block events. The *blockAscendDescend* CEO is used to block the *AscendOneFloor* and *DescendOneFloor* methods in the elevator-car component. The *blockOpenCloseDoor* CEO is used to block the *OpenDoor* and *CloseDoor* methods in the elevator-car component.

The constructor of the class expects as arguments a pointer to the light-panel component, a pointer to the elevator-car component, a pointer to the button-panel component, a pointer to the request-store component, and pointers to the two event objects to initialize the instance variables. The *currentFloor* variable is initialized to 1 indicating that in the initial state of the group, the elevator-car must be on floor 1. A coordinating action taken from the constructor is to invoke the *Block* method on the *blockAscendDescend* CEO to block the *AscendOneFloor* and *DescendOneFloor* methods in the elevator-car component.

The above abstract CE class uses the seven pure virtual methods *LookForRequestsCA*, *CloseDoorsCA*, *AscendFloorsCA*, *DescendFloorsCA*, *CloseDoorsCA*, *ResetUpDownButtonCA*, and *DetermineNextMoveCA* to take coordinating actions. The implementation and the function of each of these methods is discussed later in connection with a concrete CE class.

The *Initiate* method is invoked to execute the initial CB method *AwaitRequests.* In that method, the *LookForRequestsCA* method is invoked to poll the global request-store component every five seconds to see whether any requests are pending. If there is a pending request, the *nextFloor* variable stores the non-zero floor number to which the car must travel. On determining the non-zero value in *nextFloor*, the CB method prepares to move the car by blocking the *OpenDoor* and *CloseDoor* methods in the car component using the *blockOpenCloseDoor* CEO and by invoking the *ReplacementCB2* method to initiate the *PickupDeliverPassengers* CB method. If the *nextFloor* variable contains a zero, the *Initiate* method is re-invoked to continue the polling for a pending request.

In the *PickupDeliverPassengers* CB method, the *CloseDoorsCA* method is invoked to close the doors of the elevator car. Then, it is determined whether the car is traveling up or down using the contents of the *nextFloor* and the *currentFloor* variables. If going up, *currentTravelDirection* is set to 1, the number of floors to travel is stored in *moveNumFloors*, the *AscendFloorsCA* method is invoked to move the car to the destination floor, and, on termination of the latter method, the *currentFloor* variable is updated to record the current position of the car. If going down, *currentTravelDirection* is set to 2, the number of floors to travel is stored in *moveNumFloors*, the *DescendFloorsCA* method is invoked to move the car to the destination floor, and, on termination of the latter method, the *currentFloor* variable is updated to record the current position of the car. After reaching the destination floor, the CB method prepares to allow passengers to disembark by unblocking the *OpenDoor* and *CloseDoor* methods in the car component using the *blockOpenCloseDoor* CEO. Then, the *OpenDoorsCA* method is invoked to open the doors of the car. Note that the door may be opened due to a request from a passenger before the CB methods tries to open it. Then, the *ResetUpDownButtonCA* method is invoked to turn off an up or down request button in the up-down request-panel on the floor, if any are on. The latter resetting is done if the car arrived to the floor in response to a request or if the direction of movement of the car matches with the requested direction. Then, the *DetermineNextMoveCA* method is invoked which waits for thirty seconds to allow passengers to enter the car and make their selection of floors. Then, the latter method updates the pending requests array by collecting all the new requests which have been made after arriving at the current floor from the button-panel component (any requests made using the button-panel after this point are processed after the car arrives at the next floor). Then, the CB method decides the next move of the car by considering all the local pending requests and the requests in the global request store. The reason the global request store is searched is to pickup passengers from a floor which is between the current floor and the next destination of the car. On termination of *DetermineNextMoveCA*, the CB method prepares to move the car by blocking the *OpenDoor* and *CloseDoor* methods in the car component using the *blockOpenCloseDoor* CEO. Then, the *CloseDoorsCA* method is invoked to close the doors of the elevator car. Then, if the *nextFloor* variable has a non-zero value, the *ReplacementCB2* method is invoked to initiate the *PickupDeliverPassengers* CB method. Otherwise, the *Initiate* method is invoked to return to the initial state of the group.

The concrete CE class which may be used to instantiate a group similar to an elevator-car group in an elevator system where the light-panel is instantiated from the class *LightPanel*, the elevator car is instantiated from the class *ElevatorCar*, the button-panel is instantiated from the class *ButtonPanel*, and the request store is instantiated from the class *RequestStore*, is shown below.

```
class ElevatorCar : public Conveyor {
public:
    ElevatorCar(LightPanel* lp, ElevatorCar* car, ButtonPanel* bp,
        RequestStore* rs, CompositeEvent* c1, CompositeEvent* c2):
            Conveyor((GroupComponent*) lp, (AsynchronousObj*) car,
            (AsynchronousObj*) bp, (AsynchronousObj*) rs, c1, c2) { };
protected:
   void LookForRequestsCA();
   void CloseDoorsCA();
   void AscendFloorsCA();
   void DescendFloorsCA();
   void OpenDoorsCA();
   void ResetUpDownButtonCA() {
   void DetermineNextMoveCA();
};
```

The constructor of the *ElevatorCar* class expects pointers to the components and pointers to the event objects using which the constructor of the abstract CE class is invoked.

The above class implements all the coordinating-action methods and those methods are discussed below.

```
   void ElevatorCar:: LookForRequestsCA() {
      // wait for 5 seconds;
      Cbox* cbox = new Cbox();
      Message* m1 =
         new Message(thisThread, &RequestStore::RequestAnyDirection,
            currentFloor, cbox);
      m1→SendUntilAccepted(requestStore);
      cbox→Receive(upDownButton);
      cbox→Receive(requestedDirection);
      cbox→Receive(nextFloor);
   };
```

In the above method, the five-second delay realizes the interval at which the global request-store component is polled by each car. After that period, a message is sent to the request store to execute the *RequestAnyDirection* method with the value of the *currentFloor* variable and a Cbox pointer as arguments. The *RequestAnyDirection* method either returns an extracted item in *cbox* or returns zeros in it. After the message is accepted and executed, control returns to the *LookForRequestsCA* method in which the *upDownButton*, *requestedDirection*, and the *nextFloor* variables are assigned by extracting values returned in *cbox*. Note that a Cbox may be used to return multiple data items (three in this case). The receiver must be aware of the number of items that are returned and must explicitly extract them by invoking the proper number of *Receive* methods on the Cbox.

```
void ElevatorCar:: CloseDoorsCA() {
   ((ElevatorCar*) theCar→CloseDoor(); };
```

In the above method, the *CloseDoor* method is synchronously invoked in the *theCar* component. Note that since all the methods in the elevator-car component are blocked when the above method is invoked, the RH need not be blocked and unblocked.

```
void ElevatorCar:: AscendFloorsCA() {
   int i;
   for (i = 0; i < moveNumFloors; i++) {
      ((ElevatorCar*) theCar)→AscendOneFloor();
      Message* m1 =
         new Message(thisThread, &LightPanel::TurnOff, (currentFloor + i));
      m1→SendUntilAccepted(lightPanel);
      Message* m1 =
         new Message(thisThread, &LightPanel::TurnOn, (currentFloor + i + 1) );
      m1→SendUntilAccepted(lightPanel); };
};
```

In the above method, the elevator car is moved up, one floor at a time, by synchronously invoking the *AscendOneFloor* method in the *theCar* component. Since all the methods in the elevator-car component are blocked when the above method is invoked, the RH need not be blocked and unblocked. After *AscendOneFloor* terminates (that is, the car ascends one floor), the light-panel inside the car is updated by sending a synchronous message to execute the *TurnOff* method followed by a synchronous message to execute the *TurnOn* method. The light corresponding to the floor which the car was on is turned off and the light for the floor to which the car has arrived is turned on.

```
void ElevatorCar:: DescendFloorsCA() {
   int i;
   for (i = 0; i < moveNumFloors; i++) {
      ((ElevatorCar*) theCar)→DescendOneFloor();
      Message* m1 =
         new Message(thisThread, &LightPanel::TurnOff, (currentFloor - i));
      m1→SendUntilAccepted(lightPanel);
      Message* m1 =
         new Message(thisThread, &LightPanel::TurnOn, (currentFloor - i - 1) );
      m1→SendUntilAccepted(lightPanel); };
};
```

In the above method, the elevator car is moved down, one floor at a time, by synchronously invoking the *DescendOneFloor* method in the *theCar* component. Since all the methods in the elevator-car component are blocked when the above method is invoked, the RH need not be blocked and unblocked. After *DescendOneFloor* terminates (that is, the car descends by one floor), the light-panel inside the car is updated by sending a synchronous message to execute the *TurnOff* method followed by a synchronous message to execute the *TurnOn* method. The light corresponding to the floor which the car was on is turned off and the light for the floor to which the car arrived is turned on.

```
void ElevatorCar:: OpenDoorsCA() {
   Message* m1 = new Message(thisThread, &ElevatorCar::OpenDoor);
   m1→sendUntilAccepted(theCar);
};
```

In the above method, a synchronous message is sent to execute the *OpenDoor* method in the *theCar* component. After the *OpenDoor* method has terminated, the *OpenDoorsCA* method terminates.

```
void ElevatorCar:: ResetUpDownButtonCA() {
    int anyFloor, requestMade;
    if (requestedDirection == 1) {
        ((UpDownButtonPanel*) upDownButton)→BlockComponent();
        ((UpDownButtonPanel*) upDownButton)→TurnOffUp();
        ((UpDownButtonPanel*) upDownButton)→UnblockComponent();
        requestedDirection = 0;
        upDownButton = Null; };
    else if (requestedDirection == 2) {
        ((UpDownButtonPanel*) upDownButton)→BlockComponent();
        ((UpDownButtonPanel*) upDownButton)→TurnOffDown();
        ((UpDownButtonPanel*) upDownButton)→UnblockComponent();
        requestedDirection = 0;
        upDownButton = Null; };
    else if (currenTravelDirection == 1) {
        Cbox* cbox = new Cbox();
        Message* m1 =
            new Message(thisThread, &RequestStore::UpRequestThisFloor,
                currentFloor, cbox);
        m1→SendUntilAccepted(requestStore);
        cbox→Receive(requestMade);
        if (requestMade) {
            ((UpDownButtonPanel*) upDownButton)→BlockComponent();
            ((UpDownButtonPanel*) upDownButton)→TurnOffUp();
            ((UpDownButtonPanel*) upDownButton)→UnblockComponent(); };
    };
    else if (currenTravelDirection == 2) {
        Cbox* cbox = new Cbox();
        Message* m1 =
            new Message(thisThread, &RequestStore::DownRequestThisFloor,
                currentFloor, cbox);
        m1→SendUntilAccepted(requestStore);
        cbox→Receive(requestMade);
        if (requestMade) {
            ((UpDownButtonPanel*) upDownButton)→BlockComponent();
            ((UpDownButtonPanel*) upDownButton)→TurnOffDown();
            ((UpDownButtonPanel*) upDownButton)→UnblockComponent(); };
    };
```

The above method is used to turn off either the up or the down request-button in the up-down request-button component on a floor, after the elevator car arrives at that floor. If the arrival of the car is due to a response to such a request, either the *TurnOffUp* method (if *requestedDirection* is 1) or the *TurnOffDown* method (if *requestedDirection* is 2) is synchronously invoked (after blocking the RH) in the button-panel component stored in the *upDownButton* variable. After the latter invocation terminates, *requestedDirection* is set to zero and *upDownButton* is set to *Null* indicating that the button has been turned off.

If the arrival of the car is not due to a response to a request made using the up-down button-panel, then the *requestedDirection* variable contains a 0. In that case, in order to inform the passenger(s) that the car is ready to honor either an up or a down request, if any, the contents of the *currentTravelDirection* variable is tested, which must contain either a 1 (showing that the car arrived from a lower-numbered floor) or a 2 (showing that the car arrived from a higher-numbered floor). If *currentTravelDirection* is 1, a synchronous request message is sent to the global request-store component to execute the *UpRequestThisFloor* method with the number of the current floor as an argument. The latter method returns a non-zero value if the store contains an up-travel request made from the current floor. If such a request was made, the *TurnOffUp* method is synchronously invoked in the button-panel component stored in *upDownButton* (after blocking the RH). If *currentTravelDirection* is 2, a synchronous request message is sent to the global request-store component to execute the *DownRequestThisFloor* method with the number of the current floor as an argument. The latter method returns a non-zero value if the store contains a down-travel request made from the current floor. If such a request was made, the *TurnOffDown* method is synchronously invoked in the button-panel component stored in *upDownButton* (after blocking the RH).

```
void ElevatorCar:: DetermineNextMoveCA() {
    GroupComponent* temp1;
    int anyFloor, temp2;
    // wait for 30 seconds.
    ((ButtonPanel*) buttonPanel)→BlockComponent();
    RequestList* tempList= ((ButtonPanel*) buttonPanel)→Requests();
    ((ButtonPanel*) buttonPanel)→UnblockComponent();
```

```
      UpdateRequests(tempList);
      nextFloor = DecideNextMove();
      if (nextFloor) {
         Cbox* cbox = new Cbox();
         Message* m1 = new Message(thisThread, &RequestStore::RequestThisDirection,
                        currentFloor, nextFloor, cbox);
         m1→SendUntilAccepted(requestStore);
         cbox→Receive(temp1);
         cbox→Receive(temp2);
         cbox→Receive(anyFloor);
         if (anyFloor) {
            nextFloor = anyFloor;
            upDownButton = temp1;
            requestedDirection = temp2; };
      };
   };
```

In the above method, a thirty-second delay is introduced to allow passengers to enter their requests using the button-panel component. After that, the *Requests* method is synchronously invoked using the *buttonPanel* variable (after blocking the RH) to obtain all the requests made after the elevator car arrived at the current floor. The requests are returned by the latter method in an array, a pointer to which is stored in *tempList*. After the *Requests* method terminates and the RH is unblocked, the *UpdateRequests* method is invoked with *tempList* as an argument. The latter method updates the *pendingRequests* list using the requests in *tempList*. After *UpdateRequests* terminates, the *DecideNextMove* method is invoked which determines and returns the next floor to which the elevator car must be moved. The returned value is stored in *nextFloor*. If *nextFloor* is non-zero, then a request message is sent to the global request-store component to execute the *RequestThisDirection* method which returns a request, if any, that has been made from a floor that is in between the *currentFloor* and the *nextFloor* values. If such a request is found, it is returned in *cbox* and the *upDownButton*, *requestedDirection*, and *nextFloor* instance variables are reassigned.

3.2.7.4 Initializing a Two-Car Elevator System

In this section, a two-car elevator system serving four floors will be instantiated. Five CE objects will be required to realize the latter system: one CE object coordinating four up-down request panels on the four floors, two CE objects, one for each car, coordinating a car and the related components, and two CE objects, one for each car, coordinating four light-panels on the four floors which display the current position of the car.

The CE object for the up-down request panels may be initialized as follows:

```
UpDownButtonPanel* panels[4];
panels[0] = new UpDownButtonPanel();
panels[1] = new UpDownButtonPanel();
panels[2] = new UpDownButtonPanel();
panels[3] = new UpDownButtonPanel();
RequestStore* reqStore = new RequestStore();
ElementaryEvent* upRequest =
   new ElementaryEvent(&UpDownButtonPanel::TurnOnUp, panels[0], panels[1],
      panels[2], panels[3]);
ElementaryEvent* downRequest =
   new ElementaryEvent(&UpDownButtonPanel::TurnOnDown, panels[0], panels[1],
      panels[2], panels[3]);
ElementaryEvent* blockTurnOffUp =
   new ElementaryEvent(&UpDownButtonPanel::TurnOffUp, panels[0], panels[1],
      panels[2], panels[3]);
ElementaryEvent* blockTurnOffDown =
   new ElementaryEvent(&UpDownButtonPanel::TurnOffDown, panels[0], panels[1],
      panels[2], panels[3]);
ElevatorCarRequestPanel* ECRP =
      new ElevatorCarRequestPanel(panels, reqStore, upRequest,
         downRequest, blockTurnOffUp, blockTurnOffDown, 4);
panels[0]→RegisterCE(ECRP);
panels[1]→RegisterCE(ECRP);
panels[2]→RegisterCE(ECRP);
panels[3]→RegisterCE(ECRP);
ECRP→Initiate();
```

Four up-down button panels and a request-store are instantiated. Next, using the four panels, four EEOs are instantiated. A CE object is instantiated next using all the instantiated components and the EEOs. The number 4, passed as an argument to the constructor of the CE object, marks the number of floors. After instantiating the CE object, it is registered in the four request-panel components but not in the request-store component since the latter does not post any events to the CE object. Finally, the *Initiate* method is invoked on the CE object to execute the initial CB method.

A CE object coordinating a car and the related components may be initialized as follows. Note that the second CE object for the second car may be initialized similarly.

```
LightPanel* lightPanel = new LightPanel();
lightPanel→TurnOn(1);
ElevatorCar* theCar = new ElevatorCar();
ButtonPanel* buttonPanel = new ButtonPanel();
ElementaryEvent* event1 = new ElementaryEvent(&ElevatorCar::OpenDoor, theCar);
ElementaryEvent* event2 = new ElementaryEvent(&ElevatorCar::CloseDoor, theCar);
CompositeEvent* blockOpenCloseDoor = new CompositeEvent(event1, event2);
ElementaryEvent* event3 =
    new ElementaryEvent(&ElevatorCar::AscendOneFloor, theCar);
ElementaryEvent* event4 =
    new ElementaryEvent(&ElevatorCar::DescendOneFloor, theCar);
CompositeEvent* blockAscendDescend = new CompositeEvent(event3, event4);
ElevatorCar* elevatorCar1 =
    new ElevatorCar(lightPanel, theCar, buttonPanel, reqStore,
        blockOpenCloseDoor, blockAscendDescend);
elevatorCar1→Initiate();
```

An elevator car, a light panel, and a button panel are instantiated. The *TurnOn* method is invoked on the *lightPanel* component with an argument of 1 to indicate that the position of the car, when the elevator system is initialized, is on floor one. Next, two CEOs are instantiated from four different EEOs which the CE object must use to block events. Finally, the CE object is instantiated using all the instantiated components and the two CEOs and then the *Initiate* method is invoked on it to execute the initial CB. Note that the *reqStore* component is used once again in the above CE object since it is a shared component. Another important point to note is that none of the components in the above group registers the CE object since none of them posts any events to it.

The CE object coordinating the light panels on the four floors, for the elevator car instantiated above, may be initialized as follows. Note that the other CE object which coordinates the light panels for the second car may be initialized similarly.

```
LightPanel* floorPanels[4];
floorPanels[0] = new LightPanel();
floorPanels[1] = new LightPanel();
floorPanels[2] = new LightPanel();
floorPanels[3] = new LightPanel();
floorPanels[0]→TurnOn(1);
floorPanels[1]→TurnOn(1);
floorPanels[2]→TurnOn(1);
floorPanels[3]→TurnOn(1);
ElementaryEvent* blockEvent =
   new ElementaryEvent(&LightPanel::WhichLight, lightPanel);
ElementaryEvent* turnOnLight =
   new ElementaryEvent(&LightPanel::TurnOn, lightPanel);
ElementaryEvent* turnOffLight =
   new ElementaryEvent(&LightPanel::TurnOff, lightPanel);
ElevatorPositionDisplays* positionDisplay =
   new ElevatorPositionDisplays(floorPanels,
      lightPanel, blockEvent, turnOnLight, turnOffLight, 4);
lightPanel→RegisterCE();
positionDisplay→Initiate();
```

The above CE object shares the light-panel component inside an elevator car , stored in *lightPanel*, which was declared when the CE object for the car was initialized. Four more light-panel components are initialized, one for each floor, and the light corresponding to floor one is turned on in each one of them, indicating the initial position of the car. Next, three EEOs are instantiated which are used by the CE object to block and observe events in the light-panel component inside the car. Finally, a CE object is instantiated using all the components and the three EEOs, the light-panel component inside the car is made to register this CE object (the remaining light panels do not post any events to this CE object), and the *Initiate* method is invoked on the CE object to execute the initial CB method.

Chapter 4

Towards a Calculus of Coordinating Environments

4.1 Modeling Object-Group Behavior in CCS

In this chapter, an abstract, formal approach, based on the Calculus of Communicating Systems (CCS) [1], for describing and reasoning about the coordination of a group of independently conceived, autonomous agents is developed. The formalism developed seeks to separate the coordination specification from the specifications of the agents being coordinated. The advantages of realizing such a separation are ease of specifying general-purpose components, increasing the reuse potential of the specifications of both the components and the coordinating agents, and gaining the ability to specify software systems by *composing* component specifications.

Studying coordination independent of the programming paradigms used to realize agents is relatively new. Several proposals [16, 17, 18, 19, 20, 11, 9, 13, 21] have considered coordination and communication among software processes using high-level process-abstractions. Also, the issue of coordination among objects in object-oriented programming languages (OOPLs), both sequential and concurrent, has received significant attention recently [22, 4, 2, 3, 5, 23, 24, 10, 25, 41]. However, an abstract, formal study of coordination enables the separate semantic study of coordination constructs in different programming languages, enables a comparative study of related coordination schemes, and the semantic translations provide insights into the realization of better coordination primitives in programming languages.

Using CCS (and its variants [26, 27]) directly to specify coordination has two weaknesses. First, coordination is modeled at a very low level in CCS by making agents engage in explicit communications. Such low-level specifications are very poor candidates for specifying designs of software components which must satisfy software engineering criteria like *separation of concerns* and *reusability.* Second, as shown in the next section, when the computation steps of the composition of agents are determined using the Expansion Law of CCS, many terms are generated which represent incorrect coordination sequences among the agents.

Among other notable abstract formalisms which can capture coordination among agents, the foremost is the *Chemical Abstract Machine* (CHAM) [28] paradigm, based on the *Gamma* computation model [29]. A CHAM is an abstract machine which captures asynchronous concurrent computations among agents. The primary goal of the CHAM paradigm is to liberate high-level parallel programming methodologies from managing concurrency. It deviates from the extant models of concurrency by viewing concurrent agents as freely "moving" entities which communicate when they come in contact. The CHAM paradigm introduces the concept of *membranes* which encapsulate collections of agents to facilitate abstraction and hierarchical programming. Although very useful for modeling concurrency, the CHAM paradigm is too general purpose to serve as a formalism for specifying communication, concurrency, and coordination among "objects" in the concurrent object-oriented paradigm (COOP). The concept of *membranes* cannot be used to model coordinating agents because a *membrane* does not have an independent identity and it cannot capture the behavior of a coordinating agent explicitly. The ability to separately capture the behavior of a coordinating agent allows reasoning about its behavior and facilitates the separation of concerns among coordinating and coordinated agents.

Motivated by the above observations, the Calculus of Coordinating Environments (CCE) is proposed to study coordination as the *behavioral* union of two types of agents: agents requiring coordination and agents which elicit coordinated behavior from them. CCE views a coordinating agent as a "container" agent which establishes a *transparent* boundary around the coordinated agents (through which the agents are visible to the environment) and elicits correct behavior from the group by *observing* actions of the coordinated agents and taking coordinating actions on them. The modeling of the *observe-coordinate* property of coordinating agents in CCE was inspired by the Theory of Contexts developed in [30]. Note that

CCE is not claimed to be a general purpose calculus. Instead, the goal is to augment the COOP paradigm by laying the foundation for a special-purpose calculus which will enable the modeling of communication and coordination among concurrent objects.

The chapter is organized as follows. Section 4.2 motivates the need for designing a high-level formal abstraction in CCS for specifying coordination. Section 4.3 introduces CCE. Sections 4.4 and 4.5 specify the panel of buttons and the vending machine coordination problems in CCE, respectively.

4.2 Motivation

Autonomous agents can be coordinated in one of three ways. First, agents may bear the full responsibility of coordinating themselves and engage in explicit communication with each other for the purpose of coordination. Second, the responsibility of coordinating the agents may be delegated to a central group coordinator agent which hides the coordinated agents from the external environment. Third, the responsibility of coordination may be shared among the agents and a central group coordinator which does not hide the coordinated agents from the external environment. Each of these three ways of modeling coordination will be illustrated in this section using a simple coordination problem specified in CCS. The shortcomings of each of the CCS specifications will be discussed and the need for a better coordination abstraction will be shown.

The coordination problem considered is as follows: Two button agents are assumed to be working as a group to realize a panel of two buttons. Each button may be either in the depressed state or in the undepressed state. The environment in which the panel exists is assumed to provide the stimuli for changing the states of the buttons. The constraint that gives rise to the need for coordination is that at any point in time only one button may remain in the depressed state. Thus, when a button is depressed, the other button, if already depressed, must be undepressed.

4.2.1 Coordination Using Explicit Communication

The most direct and straightforward way of modeling the coordination among the two button agents using explicit communication is to design a button which, when depressed, queries the state of the other button and undepresses it. Such a strategy yields the following specification of the two buttons:

$(B1 \mid B2)$, where

$$B1 \overset{def}{=} depress1.(\overline{undepress2}.B1' + B1')$$
$$B1' \overset{def}{=} undepress1.B1$$
$$B2 \overset{def}{=} depress2.(\overline{undepress1}.B2' + B2')$$
$$B2' \overset{def}{=} undepress2.B2$$

The above specification makes use of three *combinators* (since they *combine* simpler agents to form more complex agents) of CCS: "|" (the *composition combinator*), "+" (the *choice combinator*), and "." (the *prefix combinator*). The composition combinator is used to specify the parallel composition of two agents which may either communicate among themselves or with other agents. The choice combinator is used to specify alternative paths of actions which an agent may engage in. The prefix combinator is used to specify a sequence of actions which an agent engages in. Actions of an agent manifest themselves at named outlets called *ports* of the agent. Two agents engage in a communication (a form of *handshake*) using two complementary ports. Complementary ports are specified in CCS by using a bar over one of the port names (for example, a and $\bar{a}$ are complementary ports). A port name without an overbar is interpreted as an *input* port (the receiver of the handshake) and a port name with an overbar is interpreted as an *output* port (the initiator of the handshake).

In the above specification, agent *B1*, modeling the behavior of button one, has ports *depress1*, *undepress1*, and $\overline{undepress2}$. Agent *B2*, modeling the behavior of button two, has ports *depress2*, *undepress2*, and $\overline{undepress1}$. *B1* is defined (using the $\overset{def}{=}$ combinator) as an agent which engages in an input action at its port *depress1* after which it has a choice of behaving either as the agent *B1'* or engaging in an output action at port $\overline{undepress2}$ before behaving as the agent *B1'*. Similarly, *B2* is defined as an agent which engages in an input action at its port *depress2* after which it has a choice of behaving either as the agent *B2'* or engaging in an output action at port $\overline{undepress1}$ before behaving as the agent

B2'. Agent *B1'* is defined as the agent which engages in an input action at port *undepress1* before behaving as *B1* and agent *B2'* is defined as the agent which engages in an input action at port *undepress2* before behaving as *B1*.

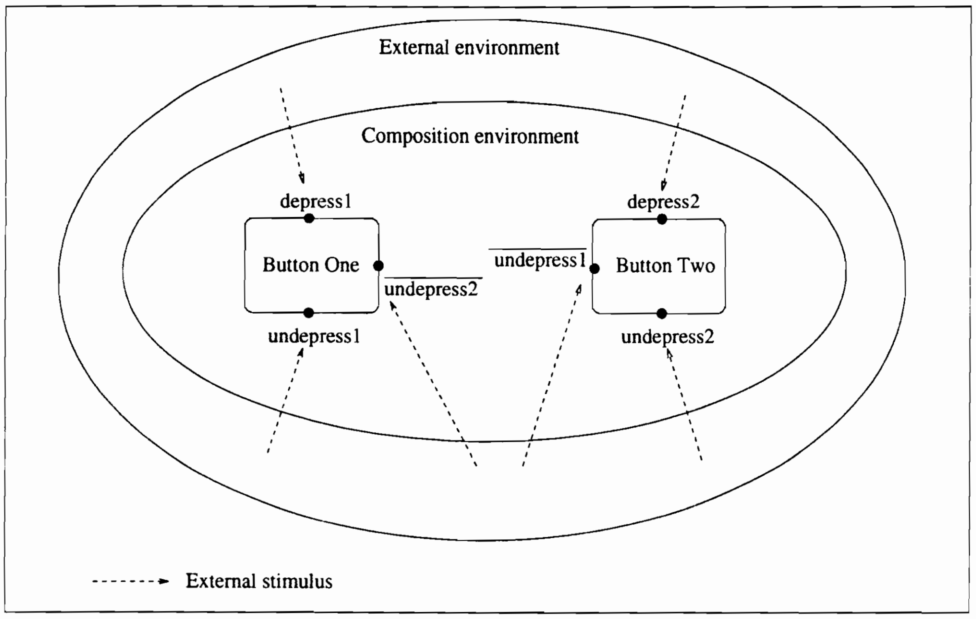

Figure 4.1: The button agents, their ports, and the external environment.

The CCS agents modeling the buttons, their ports, the environment in which they exist, and the relationship of their environment with the external environment is captured in Figure 4.1. The agents exist in an environment which will be referred to as the **composition environment** which results due to the parallel composition of the two button agents. Unless explicitly hidden, the ports of the button agents are visible to the agents in the external environment. The agents are stimulated by actions originating in the external environment and the sequence of events resulting from such stimuli are controlled by the properties of the composition environment.

The rule of CCS which defines the properties of a composition environment is called the *Expansion Law.* The latter rule defines how the behavior of agents, composed using the composition combinator, evolve. Using the Expansion Law, one may generate a *derivation tree* which depicts all possible sequences of actions which the agents in the composition environment may engage in. A partial derivation tree for the composition of the two button agents is shown in Figure 4.2. At the root of the tree (level 0) is the composed agents

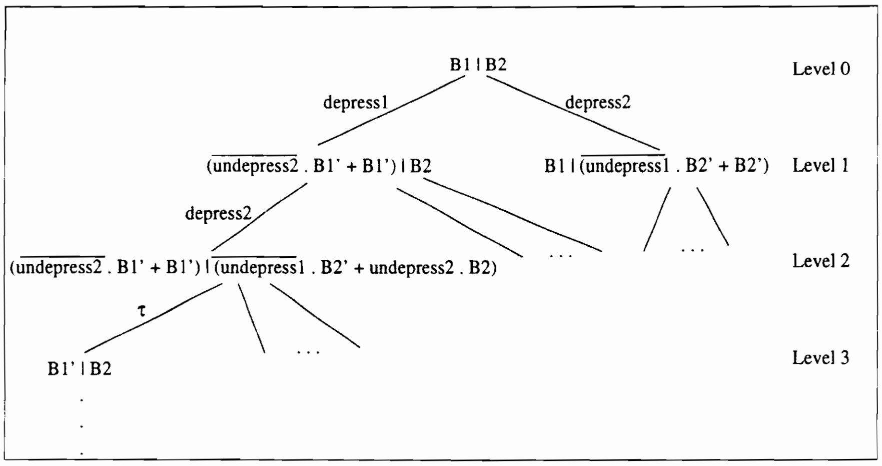
B1 | B2
depress1
depress2
Level 0
(undepress2 . B1' + B1') | B2
B1 | (undepress1 . B2' + B2')
Level 1
depress2
(undepress2 . B1' + B1') | (undepress1 . B2' + undepress2 . B2)
Level 2
τ
B1' | B2
Level 3

Figure 4.2: A partial derivation tree for the composed button agents.

$B1 \mid B2$. The transitions from a node, A say, at level i, to the children nodes at level (i+1) in the derivation tree, captures the evolution of the behavior of the agent at node A in one step of computation. Every arc between a node and its children is labeled by the name of the port using which the agent may interact with other agents in that step of the computation. Thus, the two arcs emanating from the root node labeled *depress1* and *depress2* signify that the agent in the root node may engage in either an input action at the *depress1* port or an input action at the *depress2* port. Each child of a node is an alternative way in which the behavior may evolve and only one of the alternatives is selected in an actual computation. Thus, the one-step evolution of the behavior of the agent at the root node may be expressed as:

$$
\begin{aligned}
&(B1 \mid B2)\\
&= depress1.((\overline{undepress2}.B1' + B1') \mid B2)\\
&+depress2.(B1 \mid (\overline{undepress1}.B2' + B2'))
\end{aligned}
$$

When further steps of the computation are derived using the Expansion Law, the composition of the two button agents yield the following behavior, as shown in Figure 4.2:

$(B1 \mid B2)$

$= depress1.depress2.((\overline{undepress2}.B1' + B1') \mid (\overline{undepress1}.B2' + undepress2.B2))$

$+ \ldots$

$= depress1.depress2.\tau.(B1' \mid B2) + \ldots$

The special action symbol τ captures the communication among complementary ports and is called an **internal action** (since the names of the ports involved in producing the internal action are not visible to an agent in the external environment). In the above case, the internal action is produced due to the handshake between the ports $\overline{undepress2}$ and $undepress2$.

The above execution path represents an incorrect coordination sequence: on pressing button two after button one, instead of button one being undepressed, button two is undepressed. Note that the above inconsistent behavior of the group of buttons is not induced by the the external environment. Instead, it is the ability of composed agents to **engage in unrestricted internal communications** which leads to the inconsistent behavior. The inconsistent behavior would not have arisen if there was a way of specifying that after the input action at port *depress2*, there must be an internal communication between port $\overline{undepress1}$ of button agent *B2* and port *undepress1* of button agent *B1*.

Note that the above example must not be interpreted as showing the inability of CCS to model the communication among a pair of radio buttons. CCS is an extremely expressive calculus and it can model the correct interactions among the buttons. But, such modeling is possible only at the cost of simplicity and conciseness. The buttons must introduce hidden ports and engage in an elaborate sequence of internal communications to model the correct, coordinated behavior. Such a specification would not be a direct and simple representation of a conceptually simple problem and would be difficult to understand and maintain.

4.2.2 Group Coordinators

If a group coordinator is used to coordinate the buttons, then the group can be specified as follows:

$(B1 \mid B2 \mid GC)\backslash\{depress1, undepress1, depress2, undepress2\}$, where

$B1 \overset{def}{=} depress1.B1'$; $B1' \overset{def}{=} undepress1.B1$

$B2 \overset{def}{=} depress2.B2'$; $B2' \overset{def}{=} undepress2.B2$

$GC \overset{def}{=} depressButton1.\overline{depress1}.GC' + depressButton2.\overline{depress2}.GC''$

$GC' \overset{def}{=} depressButton2.\overline{undepress1}.\overline{depress2}.GC''$

$\quad +undepressButton1.\overline{undepress1}.GC$

$GC'' \overset{def}{=} depressButton1.\overline{undepress2}.\overline{depress1}.GC'$

$\quad +undepressButton2.\overline{undepress2}.GC$

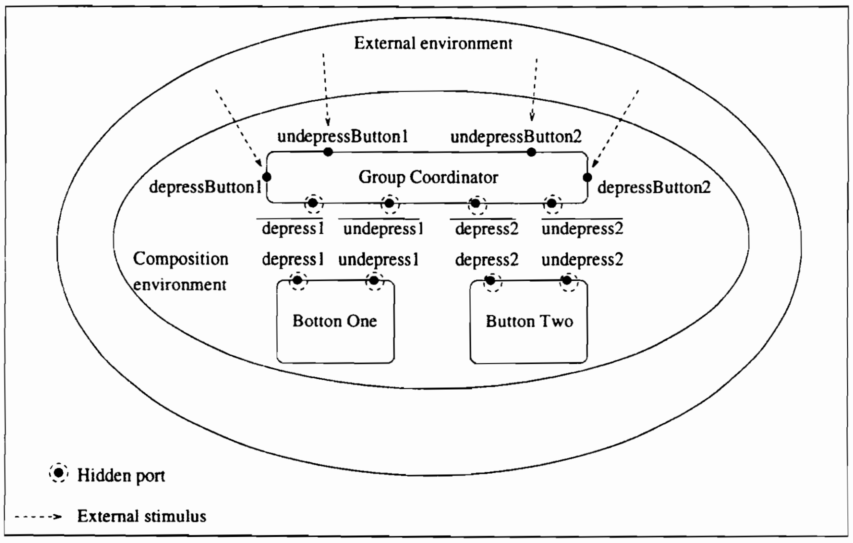

Figure 4.3: The group coordinator agent, the button agents, and the hidden and visible ports.

The above specification uses "\", the *restriction combinator* of CCS. Using this combinator, an agent is prevented from interacting with external agents using all or a subset of its ports. Although hidden from the view of external agents, restricted ports may be used by agents to communicate with each other in the composition environment (yielding τ actions). Figure 4.3 shows the group coordinator agent, the button agents, and the hidden and visible ports. Note that the restriction combinator hides both a port and its complement (the name with an overbar). For example, since *depress1* appears in the restricted set, $\overline{depress1}$ is also automatically restricted. Thus, the group coordinator hides the two button agents completely from the view of the external environment.

The button agents in the above group do not engage in any communications with each other and also do not allow any external agent to communicate with them. The only communications they engage in are internal communications with the group coordinator. Due to this centralized control over the scheduling of all communications inside the group and the hiding of the ports of the component agents, the Expansion Law does not yield any terms that display uncoordinated behavior of the group. However, since the agent GC completely hides the ports of the component agents by its own ports *depressButton1, depressButton2, undepressButton1*, and *undepressButton2*, it has a master-slave relationship with the component agents. But, as discussed in Chapter Two, such hiding of agents is not conducive for modeling point-to-point, direct communication among clients and servers in the concurrent object-oriented programming paradigm. Also, since the agent GC bears the full responsibility of enforcing coordination constraints on the group, it may become very complicated.

4.2.3 Non-intrusive Group Coordinators

Unlike the group coordinator described above, a central coordinator which does not hide coordinated components and which allows components to share part of the coordination responsibility is called a *non-intrusive group coordinator.* Due to the non-intrusion property, such group coordinators allow component agents to retain their autonomy by allowing them to interact with external agents and to make decisions about the availability of their own operations. In such a coordination scheme, the buttons panel can be specified as follows:

$$(C \mid B1 \mid B2)\backslash\{button1Depressed, button1Undepressed,$$
$$button2Depressed, button2Undepressed\}, \text{ where}$$

$$B1 \overset{def}{=} depress1.\overline{button1Depressed}.B1'$$
$$B1' \overset{def}{=} undepress1.\overline{button1Undepressed}.B1$$
$$B2 \overset{def}{=} depress2.\overline{button2Depressed}.B2'$$
$$B2' \overset{def}{=} undepress2.\overline{button2Undepressed}.B2$$

$$C \overset{def}{=} button1Depressed.C' + button2Depressed.C''$$
$$C' \overset{def}{=} button2Depressed.\overline{undepress1}.C'' + button1Undepressed.C$$
$$C'' \overset{def}{=} button1Depressed.\overline{undepress2}.C' + button2Undepressed.C$$

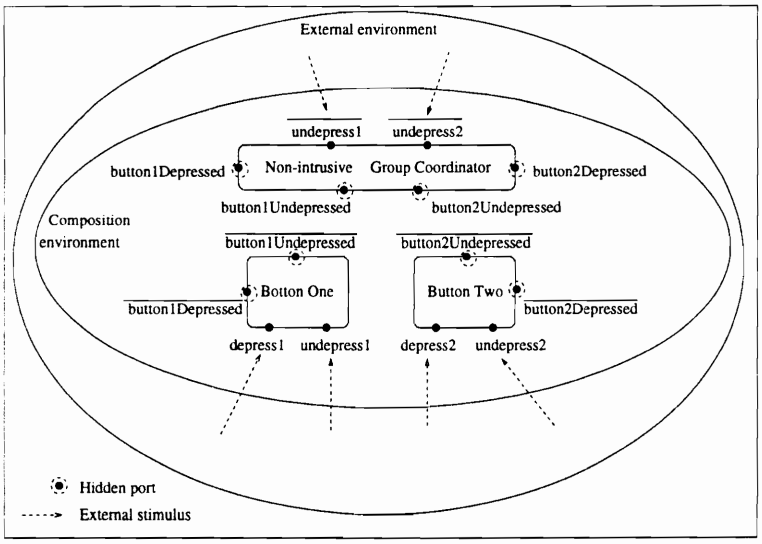

Figure 4.4: The non-intrusive group coordinator agent, the button agents, and the hidden and visible ports.

Figure 4.4 shows the non-intrusive group coordinator agent, the button agents, and the hidden and visible ports. Since the ports of the button agents are not hidden, they communicate directly with external agents. For example, button one may be depressed by communicating with it at port *depress1*. By not hiding the ports of the button agents, the group coordinator allows buttons to share part of its coordination responsibility, namely, operation scheduling: Each button decides when to schedule its own operations and controls the availability of its operations by making transitions between its states.

Although the button agents do not communicate among themselves, they engage in internal communications with the group coordinator agent in order to inform it about every communication they participate in. Such interactions are necessary in order to determine whether a communication with an external agent is consistent with the constraints of the group. For example, after being depressed, button agent *B1* explicitly engages in a communication with the group coordinator agent *C* at the output port $\overline{button1Depressed}$ before progressing with its computations. Since the latter port is restricted, the communication results in an internal action τ. This explicit communication with the central coordinator leads to complications when designing components since components must be designed with regard to their possible use in groups. If a component is not used in a group, then provisions must be made to capture its stand-alone behavior. An ideal approach would be to provide a formal abstraction that could *transparently observe* the operation scheduling decisions made by components thereby relieving them from engaging in such explicit internal communications.

The internal communications cause much more serious problems when the Expansion Law is used to expand the composition of the radio buttons and the coordinator by generating the following terms:

$$\ldots$$

$$depress1.depress2.((button1Depressed.C' + button2Depressed.C'') \mid$$
$$\overline{button1Depressed}.B1' \mid \overline{button2Depressed}.B2') + \ldots$$
$$= depress1.depress2.\tau.(C'' \mid \overline{button1Depressed}.B1' \mid B2') + \ldots$$
$$= depress1.depress2.\tau.\tau.(\overline{undepress2}.C' \mid B1'$$
$$\mid undepress2.\overline{button2Undepressed}.B2) + \ldots$$
$$= depress1.depress2.\tau.\tau.\tau.(C' \mid B1' \mid \overline{button2Undepressed}.B2) + \ldots$$

---

which represent the same incorrect coordination sequence that occurred in the case of explicit communication: when button two is depressed after button one, button two is undepressed instead of button one. The latter problem occurs because the depressing of button one is informed to the coordinator after the depressing of button two is informed. Terms corresponding to the correct sequence, informing the depressing of button one before informing the depressing of button two, is also generated along with the above incorrect sequence. Note that this must not be interpreted as a race condition in which button two succeeds in informing the group coordinator before button one can. Instead, the problem is due to the generation of every possible action of an agent in every possible order by the Expansion Law. Some of the alternatives must be prevented from occurring since they represent incorrect coordination of the components.

Even if only those terms are considered in which the occurrence of events at the button agents are informed to the group coordinator in the correct order, the visibility of the *depress1*, *depress2*, *undepress1*, and *undepress2* ports to the external environment causes two problems. First, the following terms are generated:

$$\ldots$$

$$\begin{aligned}
&depress1.\tau.depress2.\tau.(\overline{undepress1}.C'' \mid \\
&\quad undepress1.\overline{button1Undepressed}.B1 \mid B2') + \ldots \\
&= depress1.\tau.depress2.\tau.undepress1.(\overline{undepress1}.C'' \mid \\
&\quad \overline{button1Undepressed}.B1 \mid B2') + \ldots
\end{aligned}$$

---

which represent an incorrect coordination sequence. On depressing button two after button one, the coordinator's action of undepressing button one must be accepted by button one. Instead, in the above expression, button one decides not to interact with the coordinator and prepares to interact with the external environment thereby deadlocking the coordinator. The second problem is that the following terms are generated:

$$\ldots$$

$$\begin{aligned}
&depress1.\tau.depress2.\tau.(\overline{undepress1}.C'' \mid undepress1.\overline{button1Undepressed}.B1 \mid B2') + \ldots \\
&= depress1.\tau.depress2.\tau.\overline{undepress1}.(C'' \mid undepress1.\overline{button1Undepressed}.B1 \mid B2') + \\
&\ldots
\end{aligned}$$

---

which represent another incorrect coordination sequence in which the coordinator, instead of taking the coordinating action to undepress button one, decides to take it on the external environment thereby yielding an inconsistent state of the group. This happens due to the visibility of the $\overline{undepress1}$ port of the group coordinator to the external environment, as shown in Figure 4.4.

Thus, the inability to control the order of occurrence of internal actions and the inability to control the visibility of the ports of agents which are both participating in a dialogue with the environment and are being coordinated, leads to the generation of possible execution states which represent uncoordinated behavior of a group. What is required is a calculus with non-intrusive, hybrid group coordinator agents which: (i) transparently observe events at specific ports of other agents (thereby obviating the need for internal communications), (ii) allow a dynamic control over the visibility of component ports, and (iii) compose with coordinated agents to yield a pruned state-space. Such a calculus is proposed in the following section.

4.3 A Calculus of Coordinating Environments

In this section, the Calculus of Coordinating Environments (CCE) is proposed. The prime contribution of the calculus is that it enables a simple and direct specification of non-intrusive, hybrid group coordinator agents, called **Coordinating Environment agents** (CE agents), which coordinate compositions of CCS process-agents which have the **two following properties**:

- The CCS process-agents do not engage in any internal communications.
- The CCS process-agents do not hide any ports.

The first property relieves a CE agent from the burden of dealing with τ actions which, as demonstrated in the last section, causes problems in coordinating groups. Instead, if two agents ever has to communicate, the CE agent implements that communication. The second property enables a CE agent to dynamically control the visibility of the ports of the CCS process-agents instead of permanently hiding them (as done by the restriction combinator of CCS).

Figure 4.5 captures the purpose of a coordinating environment and a CE agent at a very abstract level. The behavior of the coordinating environment is embodied in the CE agent which is **not visible to the external environment**. A CE agent controls the composition environment so that the external environment may not take actions on the agents in the

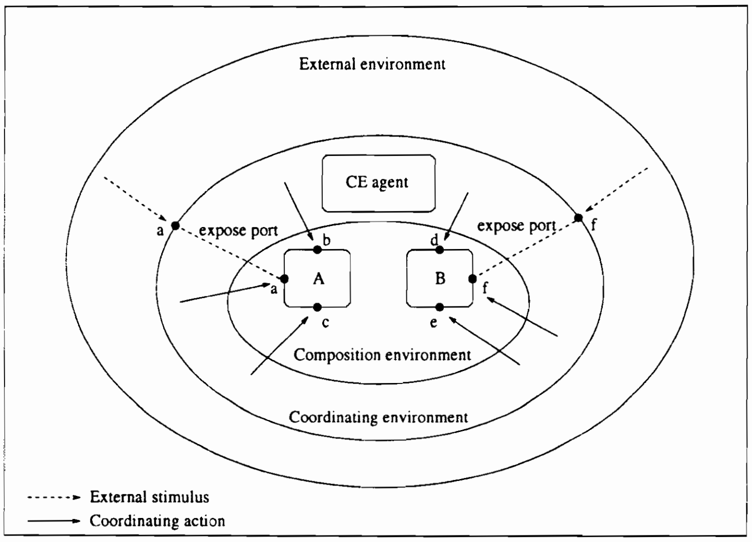

Figure 4.5: The introduction of a coordinating environment between a composition environment and the external environment.

composition environment which may result in inconsistent states of the group. Instead, the CE agent, depending on the collective state of the agents in the composition environment, selectively exposes ports of the CCS process-agents to the external environment. Such selective exposure of ports is achieved through **observation**: **A port is exposed to the external environment only if the CE agent observes that port**. In Figure 4.5, only ports a and f are exposed to the external environment in the current state of the CE agent (which reflects the current state of the group). The exposed ports may change in a subsequent state of the group. A CE agent may take a *coordinating action* by communicating with any of the agents at any of their ports. These coordinating actions are internal communications which generate τ actions but the τ actions **do not escape the boundary of the coordinating environment**. Moreover, since all coordinating actions are managed centrally by the CE agent and the CE agent **does not observe its own coordinating actions**, there is no problem associated with controlling the order of internal actions. The coordinating actions serve two important purposes: they enable a CE agent to force component agents to change their states and they enable the replacement of direct communication among component agents by communications among the CE agent and the components.

4.3.1 The Operational Semantics of CCE

The operational semantics of CE agents is described by a labeled transition system (LTS). Let $\mathcal{A}$ be a set of names ($\alpha_0, \alpha_1 \ldots$ range over $\mathcal{A}$) and $\bar{\mathcal{A}}$ be the corresponding set of *co-names* ($\bar{\alpha}_0, \bar{\alpha}_1, \ldots$ range over $\bar{\mathcal{A}}$), as in CCS. Let $\mathcal{L} = \mathcal{A} \cup \bar{\mathcal{A}}$ be the set of labels and $Act_{CCS} = \mathcal{L} \cup \{\tau\}$ be the set of actions, as in CCS. Let $\mathcal{C}$ be the set containing all CE expressions ($C, D, \ldots$ range over $\mathcal{C}$), $\mathcal{X}_{\mathcal{CE}}$ be the set of CE variables (X, Y, ... range over $\mathcal{X}$), $\mathcal{K}_{\mathcal{CE}}$ be the set of CE constants ($C_0, C_1, \ldots$ range over $\mathcal{K}_{\mathcal{CE}}$), let $ObsEv_{CE} = \mathcal{L}$, and let $CoordAct_{CE} = ObsEv_{CE} \cup \{\bigcirc\}$, where $\bigcirc \notin Act_{CCS}$ is a distinguished *no-action* symbol. Then, the operational semantics of CE agents is defined by $(\mathcal{C}, ObsEv_{CE} \times CoordAct^*_{CE}, \longmapsto)$ where the *transition relation* $\longmapsto \subseteq (\mathcal{C}, ObsEv_{CE} \times CoordAct^*_{CE}, \mathcal{C})$. For $(C, (\alpha_0, \bigcirc), C') \in \longmapsto$, we write, $C \overset{(\alpha_0,\bigcirc)}{\longmapsto} C'$ and for $(C, (\alpha_0, \alpha_1 \ldots \alpha_n), C') \in \longmapsto$, we write, $C \overset{(\alpha_0,\alpha_1 \ldots \alpha_n)}{\longmapsto} C'$.

The operational semantics described above highlights several important properties of both CE agents and the coordinated CCS process-agents. First, the two-tuple $(\alpha_0, (\alpha_1 \ldots \alpha_n))$ partitions an action of a CE agent into two distinct steps: *observation* followed by a *coordinating action sequence.* In $(\alpha_0, (\alpha_1 \ldots \alpha_n))$, the CE agent *observes* an action at port α_0 of a component process-agent and takes actions at its ports α_1 through α_n, in that sequence, which are directed to one or more component agents. When n = 0, the coordinating action sequence is empty and is represented by the $\bigcirc$ symbol. Thus, **a CE agent must observe an event** to progress with its computation.

The second important property that must be noted is that there is a **tight synchronization of the computation steps of a CE agent and those of the coordinated CCS process-agents**. The synchronization ensures that when a CE agent takes a sequence of coordinating actions, the component agents are ready to engage in such communications with the CE agent. In the absence of such a synchronization, the CE agent would not be able to progress thereby deadlocking the group.

The third important property that must be noted is that a CE agent **does not observe a τ action**. Since the communication among any pair of complementary ports is represented by a τ, a CE agent cannot distinguish between τ actions. As a result, a CE agent **coordinates only those CCS process-agents which do not communicate among themselves**.

4.3.1.1 CE Agent Expressions

CE agent expressions are formed using the following grammar:

$$C ::= \mathbf{0} \mid X \mid (\alpha_0, p) \triangleright C \mid C + C \mid C\{\Phi_1, \Phi_2\} \mid fix(X = C)$$

where $\mathbf{0}$ is the no-action CE agent, $p \in CoordAct^*_{CE}$, $\triangleright$ is like the '.' (prefix) combinator of CCS, '+' is the non-deterministic choice combinator of CCS, $\Phi_1 : ObsEv_{CE} \rightarrow ObsEv_{CE}$ and $\Phi_2 : CoordAct_{CE} \rightarrow CoordAct_{CE}$ are two renaming functions, '{ }' is the renaming combinator like '[ ]' in CCS, and the recursion expression (denoted by *fix* which must be read as "the CE agent X such that $X \stackrel{def}{=} C$") allows recursive definition of CE agents. Thus, CE agent expressions are obtained using the familiar CCS combinators. The only difference is that a CE agent is not like a CCS process-agent and hence cannot execute actions until it is composed, in a special, way with CCS process-agents.

4.3.1.2 The Transition Rules

The transition rules for the CCE combinators are given below. The names *Act, Choice*, *Rel*, and *Rec* imply that the rules are associated with $\triangleright, +$, { }, and *fix*, respectively. Also, $p \in CoordAct^*_{CE}$.

Act $$\frac{}{(\alpha_0, p) \triangleright C \xmapsto{(\alpha_0, p)} C}$$

Choice $$\frac{C_j \xmapsto{(\alpha_0, p)} C'_j}{\sum_{i \in I} C_i \xmapsto{(\alpha_0, p)} C'_j} \quad j \in I$$

Rel $$\frac{C \xmapsto{(\alpha_0, p)} C'}{C\{\Phi_1, \Phi_2\} \xmapsto{(\Phi_1(\alpha_0), \Phi_2(p))} C'\{\Phi_1, \Phi_2\}} \qquad \Phi_2(\bigcirc) = \bigcirc$$

Rec $$\frac{C\{fix(X = C)/X\} \xmapsto{(\alpha_0, p)} C'}{fix(X = C) \xmapsto{(\alpha_0, p)} C'}$$

Note that $\Phi_2(\alpha_1 \ldots \alpha_n) = \Phi_2(\alpha_1) \ldots \Phi_2(\alpha_n)$. Also note that the relabeling operator enables the separate relabeling of both the ports at which observations are made and the ports at which coordinating actions are taken so that a CE agent may adjust its observations and actions to match any relabeling done to the ports of component agents.

4.3.1.3 The New Composition Combinator

The interaction between CE agents and CCS process-agents is captured by introducing a new composition combinator in CCS which augments CCS process expressions as follows:

$$P ::= \mathbf{0} \mid X \mid \alpha_0.P \mid \overline{\alpha_0}.P \mid \tau.P \mid P + P \mid P|P \mid P\backslash L \mid P\{f\} \mid \mathit{fix}(X = E) \mid C[P_1]$$

$$P_1 ::= \mathbf{0} \mid X \mid \alpha_0.P_1 \mid \overline{\alpha_0}.P_1 \mid P_1 + P_1 \mid P_1|P_1 \mid P_1\{f\} \mid \mathit{fix}(X = E) \mid C[P_1]$$

---

where $f : \mathit{Act}_{CCS} \longrightarrow \mathit{Act}_{CCS}$ is a renaming function ($f(\tau) = \tau$), C is a CE agent expression and [ ] is the new composition combinator (note that the usual renaming combinator, [ ], of CCS has been replaced by { }) which composes a CE agent expression with a CCS process-expression to yield a CCS process-expression.

The two transition rules for [ ] which capture the interaction among CE agents and CCS process-agents are given below.

Comp1 $$\frac{P \xrightarrow{\alpha_0} P_1 \qquad C \overset{(\alpha_0,\bigcirc)}{\longmapsto} D}{C[P] \xrightarrow{\alpha_0} D[P_1]}$$

Comp2 $$\frac{P \xrightarrow{\alpha_0} P_1 \xrightarrow{\overline{\alpha_1}} \ldots \xrightarrow{\overline{\alpha_n}} P_{n+1} \qquad C \overset{(\alpha_0,\alpha_1 \ldots\, \alpha_n)}{\longmapsto} D}{C[P] \xrightarrow{\alpha_0} D[P_{n+1}]}$$

Rules *Comp1* and *Comp2* capture a step of computation by the composition of a CE agent and a CCS process-agent. In **Comp1**, the CE agent C does not take any coordinating action on the process agent P. Instead, the CE agent allows α_0 to escape its boundary and be available for interaction with the external environment. Note that in the latter process, P changes its state to P_1 and C changes its state to D.

Rule **Comp2** captures the **n-step coordinating action** property of CE agents. In an atomic step, a CE agent may observe an action at port α_0 and interact with the process agents P_1 through P_n at ports α_1 through α_n. The interactions with the process agents P_1 through P_n take place at complementary ports (for example, α_1 and $\bar{\alpha}_1$) but the resulting τ actions are consumed by the CE agent and not allowed to escape its boundary. In the process agent resulting from the n-step coordinating action, $D[P_{n+1}]$, the CE agent has progressed by one step whereas the process agent P has progressed by (n+1) steps. The CE agent deliberately suppresses n states, P_1 through P_n, of the coordinated process. The latter suppression of states prunes the actions at the nodes P_1 through P_n in the derivation tree of P and selects only the path from P to P_{n+1} that is relevant for the correct coordination of the group.

In the following, an Expansion Law for CCE is provided.

Let $P \equiv (P_1 \mid \ldots \mid P_n)$ and $C \equiv C_1 + C_2 + \ldots + C_n$. Then,

$$C[P] = \sum\{\alpha_0.C_i'[(P_1 \mid \ldots \mid P_j' \mid \ldots \mid P_n)] : P_j \xrightarrow{\alpha_0} P_j', C_i \overset{(\alpha_0, \circ)}{\longmapsto} C_i'\}$$

$$+ \sum\{\alpha_0.C_i'[(P_1 \mid \ldots \mid P_j^{N+1} \mid \ldots \mid P_n)] : P_j \xrightarrow{\alpha_0} P_j^1 \xrightarrow{\overline{\alpha_1}} P_j^2 \ldots \xrightarrow{\overline{\alpha_N}} P_j^{N+1},$$
$$C_i \overset{(\alpha_0, \alpha_1 \ldots \alpha_N)}{\longmapsto} C_i'\}$$

Note that the normal form of the process coordinated by a CE agent is an unrestricted composition of CCS process agents. The hiding of ports from the external environment is achieved by not observing events at certain ports of the coordinated process agent in specific states of the CE agent. Unlike the permanent form of restriction achieved by the restriction combinator of CCS, CE agents allow a dynamic form of port restriction based on selective observation.

4.4 The Radio Buttons in CCE

Consider the two radio button agents defined below.

$B1 \overset{def}{=} depress1.B1'$; $B1' \overset{def}{=} undepress1.B1$

$B2 \overset{def}{=} depress2.B2'$; $B2' \overset{def}{=} undepress2.B2$

The button agents do not communicate either among themselves or with any group coordinator agent. Thus, they never engage in any internal communication in the composition environment. The same set of buttons were coordinated by a group coordinator in section 4.2.2. This section shows how a non-intrusive group coordinator coordinates these two button agents.

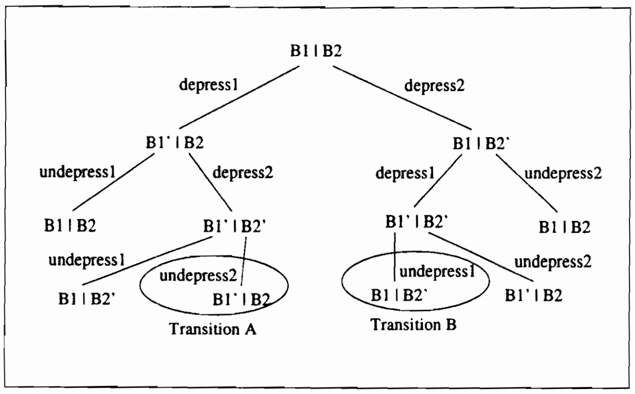

Figure 4.6: A partial derivation tree for the composition of the button agents.

Figure 4.6 shows a partial derivation tree for the composition of the two radio button agents defined above. In the tree, the leaf nodes are not expanded further because they already appear at other (expanded) internal nodes of the tree. The composed agent $B1 \mid B2$ may evolve by engaging in input actions either at port *depress1* or at port *depress2*. Agent $B1' \mid B2$ may evolve by engaging in input actions either at port *undepress1* or at port *depress2*. Agent $B1 \mid B2'$ may evolve by engaging in input actions either at port *depress1* or at port *undepress2*. Agent $B1' \mid B2'$ (in both the left and right subtrees of $(B1 \mid B2)$) may evolve by engaging in input actions either at port *undepress1* or at port *undepress2*. Note that the state $B1' \mid B2'$ represents an incorrect state of the group in which both the buttons are depressed. Thus, the task of a non-intrusive group coordinator agent will be to supress this state from being perceived by the external environment. Another important task of the non-intrusive group coordinator agent will be to block the transitions marked A and B. The path from the root of the tree to the leaf node marked Transition A in Figure 4.6, represents a behavior of the composition in which button two, being depressed after button one, is undepressed. Similarly, the path from the root of the tree to the leaf node marked Transition B, represents a behavior of the composition in which button one, being depressed after button two, is undepressed. According to the behavioral constraint imposed on the group, the two latter paths lead to inconsistent states of the group and hence, must be blocked.

Consider the non-intrusive group coordinator agent defined below.

$$\begin{aligned}
&\mathit{noneDepressed} \overset{def}{=} (\mathit{depress}1, \bigcirc) \triangleright \mathit{button}1\mathit{Depressed} \\
&\quad +(\mathit{depress}2, \bigcirc) \triangleright \mathit{button}2\mathit{Depressed} \\
&\mathit{button}1\mathit{Depressed} \overset{def}{=} (\mathit{depress}2, \overline{\mathit{undepress}1}) \triangleright \mathit{button}2\mathit{Depressed} \\
&\quad +(\mathit{undepress}1, \bigcirc) \triangleright \mathit{noneDepressed} \\
&\mathit{button}2\mathit{Depressed} \overset{def}{=} (\mathit{depress}1, \overline{\mathit{undepress}2}) \triangleright \mathit{button}1\mathit{Depressed} \\
&\quad +(\mathit{undepress}2, \bigcirc) \triangleright \mathit{noneDepressed}
\end{aligned}$$

Figure 4.7 shows a partial derivation tree for the group coordinator agent. In the tree, the leaf nodes are not expanded further because they already appear at other (expanded) internal nodes of the tree. The states of the group coordinator model the different consistent states of the button agents it coordinates. The coordinator starts in the state *noneDepressed* in which none of the buttons are depressed. From that state, it has the option of observing an action either at the *depress1* port or at the *depress2* port. This observation amounts to the selective exposing of those two ports so that the external environment may communicate with them. No coordinating actions are associated with the latter observations. Thus, in the derivation tree, the two two-tuples, $(\mathit{depress}1, \bigcirc)$ and $(\mathit{depress}2, \bigcirc)$, mark the transitions to level 1. The state *button1Depressed* captures the relative configuration of the buttons in which button one is depressed and button two is undepressed. From this state, the coordinator either observes an action at the *undepress1* port or observes an action at the *depress2* port. In the former case, the coordinator makes a transition to the *noneDepressed* state without taking any any coordinating action and in the latter case, it makes a transition to the *button2Depressed* state after taking an output action at the port $\overline{undepress1}$. The behavior from the state *button2Depressed* is analogous to the behavior of the coordinator from the state *button1Depressed.*

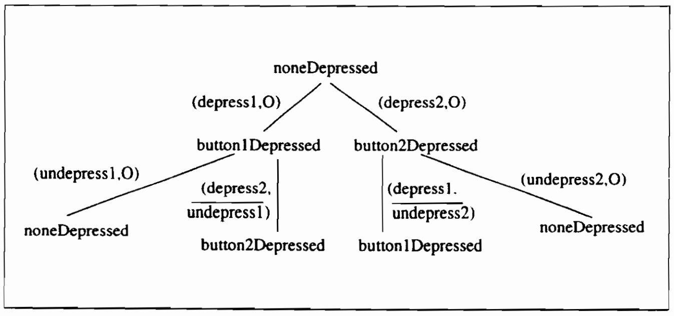

Figure 4.7: A partial derivation tree for the group coordinator agent.

The behavior of the composition of the buttons with the CE agent is shown below.

$$\begin{aligned}
&noneDepressed[B1 \mid B2] \\
&= noneDepressed[depress1.B1' \mid depress2.B2'] \\
&= depress1.button1Depressed[B1' \mid depress2.B2'] \\
&\quad + depress2.button2Depressed[depress1.B1' \mid B2'] \\
&= depress1.button1Depressed[undepress1.B1 \mid depress2.B2'] \\
&\quad + depress2.button2Depressed[depress1.B1' \mid undepress2.B2] \\
&= depress1.undepress1.noneDepressed[B1 \mid B2] \\
&\quad + depress1.depress2.button2Depressed[B1 \mid B2'] \\
&\quad + depress2.undepress2.noneDepressed[B1 \mid B2] \\
&\quad + depress2.depress1.button1Depressed[B1' \mid B2]
\end{aligned}$$

Note that unlike the previous problem of incorrect coordination, in the above expansion, when button two is depressed after button one, button one is undepressed, button two remains depressed, and the CE agent is in a state in which it may either observe the undepressing of button two or the depressing of button one.

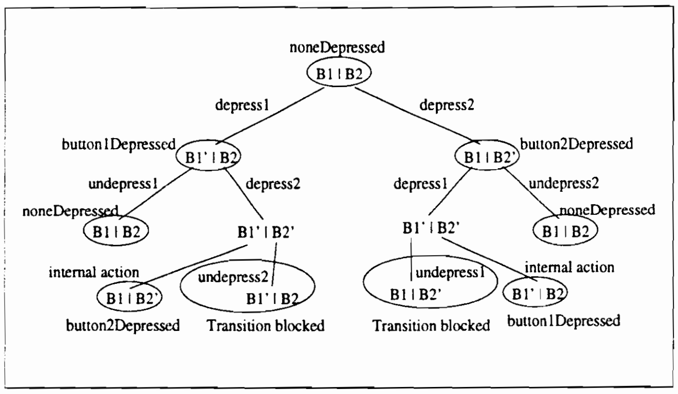

Figure 4.8: The partial derivation tree for the composition of the non-intrusive group coordinator agent and the two button agents. The above tree is the result of the superposition of the partial derivation trees in Figure 4.6 and Figure 4.7.

The composed behavior of the non-intrusive group coordinator agent and the two button agents may be better understood by superposing their derivation trees, as shown in Figure 4.8. The state $B1 \mid B2$ of the buttons corresponds to the state *noneDepressed* of the group coordinator. From the latter state, the composition may evolve by either an input action at port *depress1* or an input action at port *depress2*. The state $B1' \mid B2$ of the buttons corresponds to the state *button1Depressed* of the group coordinator. From the latter state, the composition may evolve by either an input action at port *undepress1* or an input action at port *depress2* followed by an internal action (that results due to a communication between the complementary ports *undepress1* and $\overline{undepress1}$). The latter, one-step coordinating action takes the group directly to the state $B1 \mid B2'$ that corresponds to the state *button2Depressed* of the group coordinator. Three important observations must be made: the one-step coordinating action following the action at port *depress2* appears as atomic to the external environment, the state $B1' \mid B2'$ of the buttons is rendered a transient, internal state which is not perceived by the external environment, and the inconsistent transition from the state $B1' \mid B2'$ through an input action at port *undepress2* is blocked by the group coordinator. A similar explanation applies to the transitions in the right subtree of the root node in Figure 4.8.

4.5 Specifying a Vending Machine in CCE

Consider the two slot agents, *S1* and *S2*, and a coin acceptor agent, *CA*, defined below.

$$S1 \overset{def}{=} open1.amount1.(\overline{dispense1}.S1 + \overline{fail1}.S1)$$
$$S2 \overset{def}{=} open2.amount2.(\overline{dispense2}.S2 + \overline{fail2}.S2)$$
$$CA \overset{def}{=} insert.CA'$$
$$CA' \overset{def}{=} insert.CA' + refund.CA + \overline{insertedAmount}.CA' + refundExcess.CA$$

Slot *S1* starts with an input action at port *open1*. Next, it engages in another input action at port *amount1* (which models the transfer of an amount value to the slot). After the latter communication, the slot may perform an output action at port $\overline{dispense1}$ (which models the successful extraction of an item) or may perform an output action at port $\overline{fail1}$ (which models the failure of a slot to dispense an item). Slot *S2* behaves analogously. The coin acceptor agent *CA* starts with an input action at port *insert* and makes a transition to state *CA'*. In the latter state, it may engage in one of four actions. First, it may engage in an input action at port *insert* and remain in state *CA'* (which models the insertion of multiple coins). Second, it may engage in an input action at port *refund* and transit to state *CA* (which models the extraction of all inserted coins). Third, it may engage in an output action at port $\overline{insertedAmount}$ and remain in state *CA'* (which models reading the inserted amount from the coin acceptor). Fourth, it may engage in an input action at port *refundExcess* and transit to state *CA* (which models the refunding of the excess amount inserted).

Figure 4.9 shows a partial derivation tree for the composition of the two slot agents and the coin acceptor agent defined above. The tree is similar to the derivation trees used in the last section except that several paths of the evolution of the behavior of the composition has not been pursued. Such paths are shown as empty subtrees (the empty ovals in the figure) and the actions which lead to the unexpanded paths are listed on the arcs. The main reasons for not expanding those paths are to keep the tree manageable in size and because those paths, after composing the above agents with a group coordinator, will be blocked.

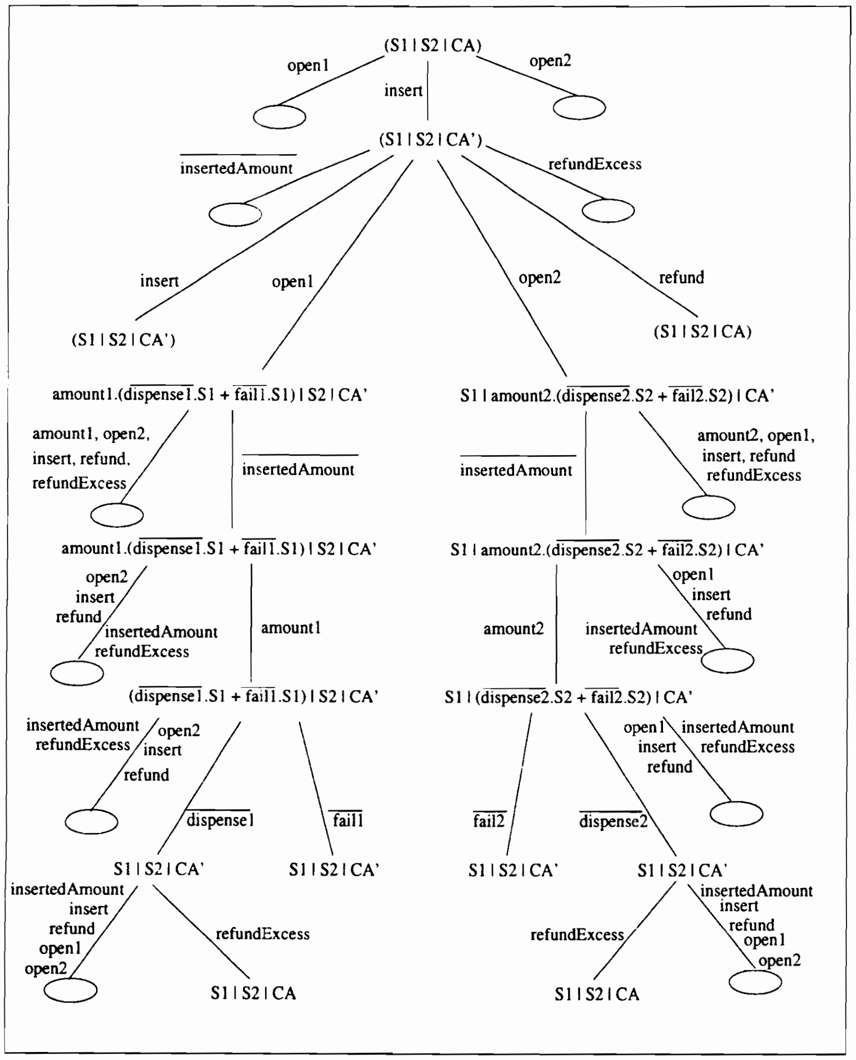
(S1 | S2 | CA)
open1
insert
open2
(S1 | S2 | CA')
insertedAmount
refundExcess
insert
open1
open2
refund
(S1 | S2 | CA')
(S1 | S2 | CA)
amount1.(dispense1.S1 + fail1.S1) | S2 | CA'
S1 | amount2.(dispense2.S2 + fail2.S2) | CA'
amount1, open2, insert, refund, refundExcess
insertedAmount
insertedAmount
amount2, open1, insert, refund refundExcess
amount1.(dispense1.S1 + fail1.S1) | S2 | CA'
S1 | amount2.(dispense2.S2 + fail2.S2) | CA'
open2 insert refund
insertedAmount refundExcess
amount1
amount2
insertedAmount refundExcess
open1 insert refund
(dispense1.S1 + fail1.S1) | S2 | CA'
S1 | (dispense2.S2 + fail2.S2) | CA'
insertedAmount refundExcess
open2 insert refund
dispense1
fail1
fail2
dispense2
open1 insert refund
insertedAmount refundExcess
S1 | S2 | CA'
S1 | S2 | CA'
S1 | S2 | CA'
S1 | S2 | CA'
insertedAmount insert refund open1 open2
refundExcess
refundExcess
insertedAmount insert refund open1 open2
S1 | S2 | CA
S1 | S2 | CA

Figure 4.9: A partial derivation tree for the composition of the two slot agents and the coin acceptor agent.

The role of a non-intrusive group coordinator in coordinating the composition of the two slot agents and the coin acceptor agent is to elicit from them the behavior of a vending machine. A quiescent vending machine is triggered by inserting one or more coins after which either a request to refund all the coins may be made or a request to open one of the slots may be made. If a request to open a slot is made, then the group coordinator must obtain the inserted amount from the coin acceptor and transfer it to that slot. This transfer illustrates how the group coordinator relieves components from engaging in explicit internal communications. Then, if the coordinator observes the dispensing of an item by the slot, it refunds the excess amount inserted and returns to its initial state. Otherwise, if it observes the failure to dispense an item by the slot, it returns to a state in which it allows requests for either further coin insertions, or refunding the inserted amount, or opening a slot.

The non-intrusive, vending machine coordinator agent is defined below. Note, to increase readability, a semicolon separates two consecutive actions in a sequence of coordinating actions of the CE agent.

$$
\begin{aligned}
&waitForCoins \overset{def}{=} (insert, \bigcirc) \triangleright processRequests \\
&processRequests \overset{def}{=} (insert, \bigcirc) \triangleright processRequests \\
&\quad +(refund, \bigcirc) \triangleright waitForCoins \\
&\quad +(open1, insertedAmount; \overline{amount1}) \triangleright slot1Request \\
&\quad +(open2, insertedAmount; \overline{amount2}) \triangleright slot2Request \\
&slot1Request \overset{def}{=} (\overline{dispense1}, \overline{refundExcess}) \triangleright waitForCoins \\
&\quad +(\overline{fail1}, \bigcirc) \triangleright processRequests \\
&slot2Request \overset{def}{=} (\overline{dispense2}, \overline{refundExcess}) \triangleright waitForCoins \\
&\quad +(\overline{fail2}, \bigcirc) \triangleright processRequests
\end{aligned}
$$

Figure 4.10 shows a partial derivation tree for the group coordinator agent. The states of the group coordinator model the different valid states of the components of the vending machine. The coordinator starts in the state *waitForCoins* in which it has the option of observing an action at (and hence exposing) the port *insert* and making a transition to the state *processRequests* without taking any coordinating action. In the state *processRequests*, the coordinator has the option of observing an event at any of four different ports. First, it may observe an input action at port *insert* and revert back to *processRequests* without taking any coordinating action. Second, it may observe an input action at port *open1* and transit to the state *slot1Request* after taking a sequence of two coordinating actions: an input action at port *insertedAmount* followed by an output action at port $\overline{amount1}$. Third, it may observe an input action at port *open2* and transit to the state *slot2Request* after taking a sequence of two coordinating actions: an input action at port *insertedAmount* followed by an output action at port $\overline{amount2}$. Fourth, it may observe an input action at port *refund* and transit to the initial state *waitForCoins* without taking any coordinating action. In the state *slot1Request*, the coordinator may observe either an output action at port $\overline{dispense1}$ or an output action at port $\overline{fail1}$. In the former case, it takes an output coordinating action at the port $\overline{refundExcess}$ and makes a transition to state *waitForCoins* and in the latter case, it takes no coordinating action makes a transition to state *processRequests*. The behavior of the coordinator from the state *slot2Request* is analogous to its behavior from the state *slot1Request*.

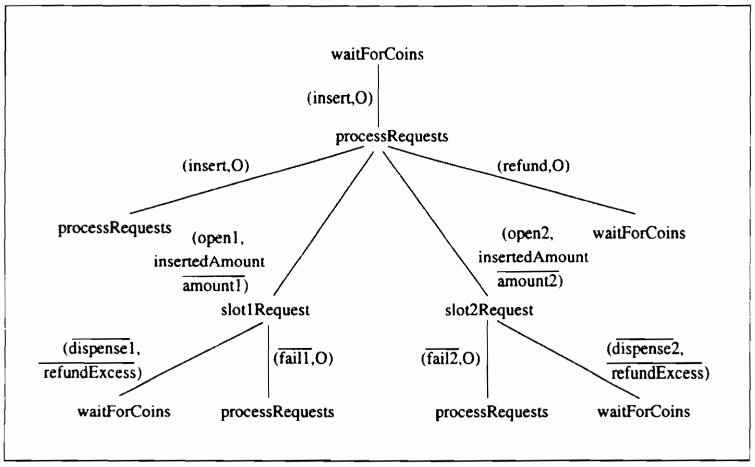

Figure 4.10: A partial derivation tree for the group coordinator agent.

The behavior of the composition of the slot and coin acceptor agents with the non-intrusive CE agent using the new composition combinator and the new transition rules is shown below.

$$
\begin{aligned}
& waitForCoins[S1 \mid S2 \mid CA] \\
& = insert.processRequests[S1 \mid S2 \mid CA'] \\
& = insert.insert.processRequests[S1 \mid S2 \mid CA'] \\
& \quad + insert.refund.waitForCoins[S1 \mid S2 \mid CA] \\
& \quad + insert.open1.slot1Request[(\overline{dispense1}.S1 + \overline{fail1}.S1) \mid S2 \mid CA'] \\
& \quad + insert.open2.slot2Request[S1 \mid (\overline{dispense2}.S2 + \overline{fail2}.S2) \mid CA'] \\
& = insert.insert.processRequests[S1 \mid S2 \mid CA'] \\
& \quad + insert.refund.waitForCoins[S1 \mid S2 \mid CA]
\end{aligned}
$$

$+insert.open1.\overline{dispense1}.waitForCoins[S1 \mid S2 \mid CA]$

$+insert.open1.\overline{fail1}.processRequests[S1 \mid S2 \mid CA']$

$+insert.open2.\overline{dispense2}.waitForCoins[S1 \mid S2 \mid CA]$

$+insert.open2.\overline{fail2}.processRequests[S1 \mid S2 \mid CA']$

The composed behavior of the non-intrusive group coordinator agent, the two slot agents, and the coin acceptor agent may be better understood by superposing their derivation trees, as shown in Figure 4.11. The state $(S1 \mid S2 \mid CA)$ corresponds to the state *waitForCoins* of the group coordinator. From the latter state, the composition may evolve by only an input action at port *insert*. The state $(S1 \mid S2 \mid CA')$ corresponds to the state *processRequests* of the group coordinator. From the latter state, the composition may evolve by any one of four input actions at the ports *insert*, *open1*, *open2*, and *refund*. The actions at ports *insert* and *refund* takes the composition to the already expanded states $(S1 \mid S2 \mid CA')$ and $(S1 \mid S2 \mid CA)$, respectively. The actions at ports *open1* and *open2* result in a sequence of two atomic, unobservable, internal coordinating actions before resulting in the states $(\overline{dispense1}.S1 + \overline{fail1}.S1) \mid S2 \mid CA'$ and $S1 \mid (\overline{dispense2}.S2 + \overline{fail2}.S2) \mid CA'$, respectively. The former state corresponds to the state *slot1Request* of the group coordinator and the latter state corresponds to the state *slot2Request* of the group coordinator. From $(\overline{dispense1}.S1 + \overline{fail1}.S1) \mid S2 \mid CA'$, the composition may evolve by either an output action at port $\overline{dispense1}$ or an output action at port $\overline{fail1}$. The former event causes an unobservable, internal coordinating action before resulting in the already expanded state $S1 \mid S2 \mid CA$ and the latter event results in the already expanded state $S1 \mid S2 \mid CA'$. The behavior from $S1 \mid (\overline{dispense2}.S2 + \overline{fail2}.S2) \mid CA'$ is analogous to that from $(\overline{dispense1}.S1 + \overline{fail1}.S1) \mid S2 \mid CA'$. As with the derivation tree in Figure 4.8, the observation-action sequences appear as atomic to the external environment and they make the composition transit through several transient, internal states which are not perceived by the external environment, and several inconsistent transitions of the composition are blocked by the group coordinator.

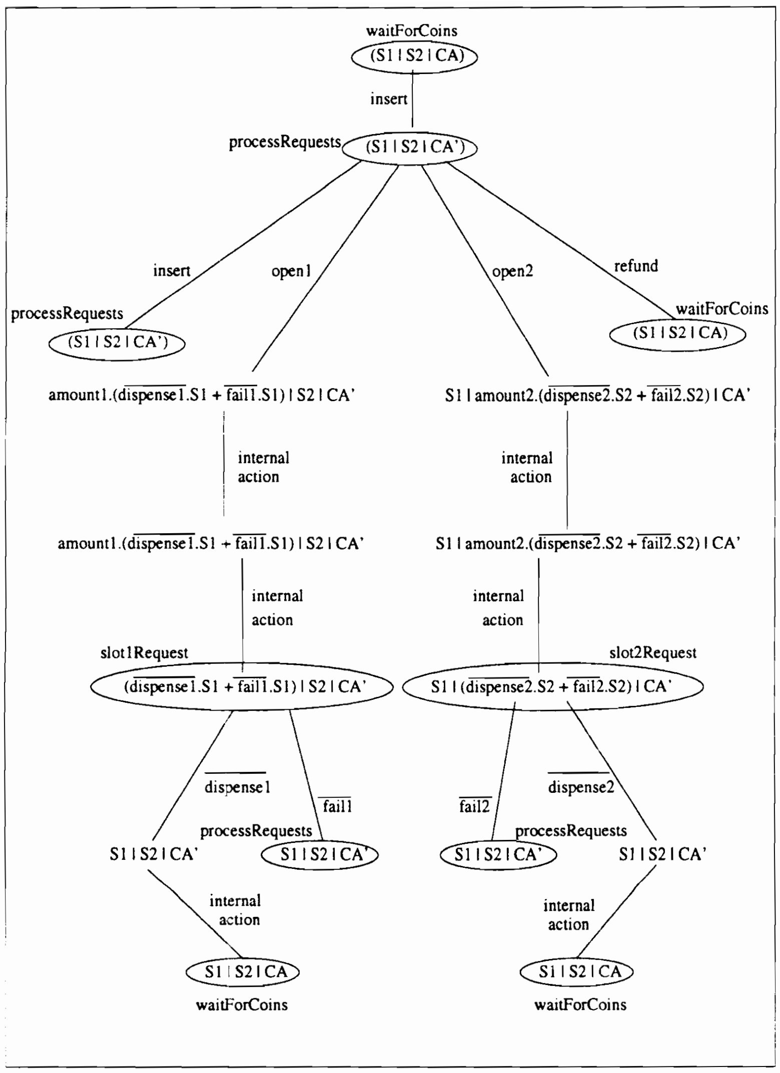

Figure 4.11: The partial derivation tree for the composition of the non-intrusive group coordinator agent, the two slot agents, and the coin acceptor agent. The above tree is the result of the superposition of the partial derivation trees in Figure 4.9 and Figure 4.10.

Chapter 5

A Detailed Design of the Coordinating Environments Model

5.1 Introduction

In this chapter, a detailed design of the Coordinating Environment (CEs) model is provided to show that the objects which play the most significant roles in realizing the model can be implemented. The underlying execution environment is assumed to support light-weight *threads.* For the details of how such a thread-based system may be utilized to realize autonomous objects, refer to [31].

The chapter is organized as follows. Section 5.2 discusses the realization of the elementary and the composite event objects. Section 5.3 provides lifecycle diagrams which show how event objects must be used and how their states change during use. Section 5.4 discusses the realization of Coordinating Environment (CE) objects and elaborates on the realization of the *Observe* operation which implements "event observation", the fundamental notion of the CEs model.

5.2 The Design of the Event Objects

The behavior of event objects is realized by three classes: *Event*, *ElementaryEvent*, and *CompositeEvent.* A partial realization of each of the latter classes and the implementation of some of their most important methods are discussed in the following sections.

5.2.1 The Event Class

The *Event* class captures the common features of both elementary and composite event objects. A partial definition of the class is shown below.

```
class Event : public Object {
   protected:
           ObjectList* constituents;
           int        numOfConstituents;
           Object*     currentConstituent;
           void        AssignCurrentConstituent(Object*);
                ...
   public:
           Event();
           virtual void AssignConstituents(Object*, ...);
           virtual GroupComponent* WhichComponent();
           virtual void Schedule();
           virtual void Block();
           virtual void Unblock();
           virtual void AwaitTermination(CoordinatingEnvironment*);
                ...
}
```

Every event object consists of one or more constituent objects. The constituents are stored in a list called *constituents.* The nature of these constituents differs with the type of the event object, whether elementary or composite. In the case of an elementary event object (EEO), *constituents* stores a list of pointers to instantiations of the subclasses of the *GroupComponent* class. In the case of a composite event object (CEO), *constituents* stores a list of pointers to instantiations of the *ElementaryEvent* class. The length of this list is stored in *numOfConstituents.* The constituent which participated in the most recently observed event is stored in *currentConstituent.* Note that depending on the type of the event object, *currentConstituent* stores either a pointer to an actual component or a pointer to an EEO. The constructor initializes these instance variables as follows:

```
Event::Event() {
   constituents = Null;
   numOfConstituents = 0;
   currentConstituent = Null; };
```

The *AssignConstituents* method is used to assign the list of constituents. For elementary event objects, instantiations of the subclasses of the *GroupComponent* class must be supplied as arguments to the latter method whereas for composite event objects, instantiations of the *ElementaryEvent* class must be supplied. The *WhichComponent* method is used to determine which of the recorded components posted the last event observed by the event object. The *AssignCurrentConstituent* method, as shown below, is used to assign *currentConstituent* after an event is observed.

```
void Event::AssignCurrentConstituent(Object* o) {
   currentConstituent = o; };
```

The remaining public methods in the *Event* class represent **four coordinating actions**. The *Schedule* method is used to schedule for execution the method requested in an observed acceptance event. The *Block* method is used to block the acceptance of specific request messages. The *Unblock* method is used to unblock blocked methods. The *AwaitTermination* method schedules a method for execution and waits for its termination.

5.2.2 The ElementaryEvent Class

The *ElementaryEvent* class is used to instantiate elementary event objects. A partial definition of the class is shown below.

```
class ElementaryEvent : public Event {
   protected:
           GroupComponent* observeInComponent;
           methodId        requestedMethod;
           eeoUsage        useToObserve;
           EventMessage*   eventMessage;
           int             blockMethod;
           int IsEventPosted() { if (eventMessage) return 1 else return 0; };
           void Initialize();
                  ...
   public:
           ElementaryEvent(methodId, GroupComponent*, ...);
           ElementaryEvent(GroupComponent*, ...);
           ElementaryEvent(methodId);
```

```
            ElementaryEvent();
            void AssignConstituents(Object*, ...);
            Event* EventUsage(eeoUsage);
            Event* NextEventIn(GroupComponent*);
            GroupComponent* WhichComponent();
            void AssignMethod(methodId);
            void Block();
            void Unblock();
            void AwaitTermination(CoordinatingEnvironment*);
            void Schedule();
            void AddArguments(int, ArgumentList*);
                ...
    }
```

There are several ways in which an EEO may be constructed. One may provide a method reference and pointers to all the components in which events associated with that method must be observed. Otherwise, one may provide either a method reference, or pointers to components, or nothing at all. If no method reference is provided, the *AssignMethod* method must be used to assign one before the event object may be used to observe events. If no component pointers are provided, the *AssignConstituents* method must be used to provide them before the event object may be used to observe events. The constructors and the *AssignConstituents* method are defined below.

```
ElementaryEvent::
   ElementaryEvent(methodId m, GroupComponent* a, ...) {
   Initialize();
   requestedMethod = m;
   // Extract pointers to components from stack,
   // form list, and store in "constituents";
   numOfConstituents = constituents→Length();
   if (numOfConstituents == 1)
      observeInComponent = a; };


ElementaryEvent::
   ElementaryEvent(GroupComponent* a, ...) {
   Initialize();
   // Extract pointers to components from stack,
   // form list, and store in "constituents";
   numOfConstituents = constituents→Length();
```

```
    if (numOfConstituents == 1)
        observeInComponent = a; };


ElementaryEvent::ElementaryEvent(methodId m) {
    Initialize();
    requestedMethod = m; };


ElementaryEvent::ElementaryEvent() {
    Initialize(); };


void ElementaryEvent::
    AssignConstituents(Object* a, ...) {
        // Extract pointers to components from stack,
        // form list, and store in "constituents";
        numOfConstituents = constituents→Length();
        if (numOfConstituents == 1)
            observeInComponent = a; };
```

Every constructor invokes the *Initialize* method to assign the default values to the instance variables of the class. The first constructor reassigns *requestedMethod*, extracts the component pointers and initializes *constituents*, and then initializes *numOfConstituents*, *observeInComponent* (conditionally), and *currentConstituent.* The second constructor does everything the first constructor does except for assigning *requestedMethod.* The third constructor invokes the *Initialize* method before reassigning *requestedMethod.* The fourth constructor only invokes *Initialize.* The *AssignConstituents* method does everything done by the first constructor, except for reassigning *requestedMethod* and invoking *Initialize.*

```
void ElementaryEvent::Initialize() {
    requestedMethod = Null;
    observeInComponent = Null;
    useToObserve = accept;
    eventMessage = Null;
    blockMethod = 0; };
```

The *Initialize* method, shown above, stores the default values in the instance variables. The variable *requestedMethod* stores a reference to the method whose acceptance or termination is observed using an EEO. The *observeInComponent* variable is used to record a constituent in which an event must be observed using the EEO. If *observeInComponent* is set, the EEO is used to observe an event in that component. Otherwise, an event in any one of the components recorded in *constituents* is observed. This instance variable is reset to *Null* after a successful observation is made using the event object. The default usage type of an event object, *accept* (for message-acceptance) is stored in *useToObserve.* The *eventMessage* instance variable, which records a pointer to an event message after a successful observation is made using the event object, is set to *Null* indicating that no event posting has been observed. The *blockMethod* instance variable is used to record whether the event object must block acceptance of the requested method in all the components. It is set to zero to indicate that the event object must not block acceptance events.

5.2.2.1 The EventUsage, NextEventIn, WhichComponent, and AssignMethod Operations

```
Event* ElementaryEvent::EventUsage(eeoUsage t) {
    useToObserve = t; return (Event*) this; };
```

The *EventUsage* method, shown above, may be used to modify the usage type of an EEO.

```
Event* ElementaryEvent::NextEventIn(GroupComponent* a) {
    observeInComponent = a; return (Event*) this;};
```

If there are multiple components recorded in an EEO, then using the *NextEventIn* method, as shown above, one may specify the particular component in which the next event must be observed.

```
GroupComponent* ElementaryEvent::WhichComponent() {
   return (GroupComponent*) currentConstituent; };
```

The *WhichComponent* method, shown above, is used to retrieve a pointer to the component in which the most recent event observation was made using an EEO. This method is particularly useful for a multi-constituent EEO.

```
void ElementaryEvent::AssignMethod(methodId m) {
   requestedMethod = m; };
```

The *AssignMethod* method, shown above, may be used to assign *requestedMethod* instance variable.

5.2.2.2 The Block, Unblock, and AwaitTermination Operations

```
void ElementaryEvent::Block() {
   if ((useToObserve == accept) && !(requestedMethod == Null)) {
      blockMethod = 1;
      if (observeInComponent)
        ((GroupComponent*) observeInComponent)→BlockMethod(requestedMethod);
      else
         for each component, "C", in the constituent list:
            ((GroupComponent*) C)→BlockMethod(requestedMethod);
   };
   else ERROR; };


void ElementaryEvent::Unblock() {
   if (blockMethod) {
      blockMethod == 0;
      if (observeInComponent)
     ((GroupComponent*) observeInComponent)→UnblockMethod(requestedMethod);
      else
         for each component, "C", in the constituent list:
            ((GroupComponent*) C)→UnblockMethod(requestedMethod);
   };
   else ERROR; };
```

The realization of the *Block* and *Unblock* methods are shown above. The *Block* method may be invoked on an EEO only if its usage type is *accept* and the *requestedMethod* variable has been assigned. The method sets *blockMethod* to 1 indicating that the EEO may not be used to observe any events. The actual blocking of a method is done using the *BlockMethod* method defined in the *GroupComponent* class. If *observeInComponent* is set in the EEO then the latter method is invoked using it. Otherwise, *BlockMethod* is invoked in every component stored in the constituents list.

The *Unblock* method may be invoked on an EEO only if *blockMethod* has been previously set to 1. The method resets *blockMethod* to zero indicating that the EEO may be used to observe acceptance events. The actual unblocking of a method is done using the *UnblockMethod* method defined in the *GroupComponent* class. If *observeInComponent* is set in the EEO then the *UnblockMethod* method is invoked on it. Otherwise, the *UnblockMethod* is invoked in every component stored in the constituents list.

```
void ElementaryEvent::AwaitTermination(CoordinatingEnvironment* ce) {
        Schedule( );
        ce→Observe(this→EventUsage(terminate)); };
```

The realization of the *AwaitTermination* method is shown above. When this method is invoked, a pointer to the CE object to which the CB method belongs must be supplied as an argument. The *AwaitTermination* method schedules for execution the request message observed using the event object by invoking the *Schedule* method. Then, it awaits the termination of the scheduled method by marking the event object to observe termination (using *EventUsage(terminate)*) and then invoking the *Observe* operation on the CE object that was supplied as an argument.

5.2.2.3 The Schedule and AddArguments Operations

The *Schedule* method of the *ElementaryEvent* class is used to schedule for execution a requested method. Prior to discussing the realization of the *Schedule* method, the structure of an event message is discussed. Event messages are instantiated from the *EventMessage* class, a partial declaration of which is shown below.

```
class EventMessage : public Object {
   protected:
           Message* theRequestMessage;
           Cbox* informComponent;
                ...
   public:
           methodId requestedMethod;
           GroupComponent* messageFrom;
           int scheduleMethod;
           int methodTerminated;
           EventMessage(GroupComponent*, Message*, Cbox*);
           GroupComponent* WhichComponent();
           void Schedule();
           void Terminated();
           void AddArguments(int, ArgumentList*);
                ...
};
```

An event message is constructed by the request handler in a component whenever an acceptance event must be posted to the coordinating CE object. The constructor is defined as follows:

```
EventMessage::EventMessage(GroupComponent* c, Message* m, Cbox* cbox) {
   requestedMethod = m→Method();
   messageFrom = c;
   theRequestMessage = m;
   scheduleMethod = 0;
   methodTerminated = 0;
   informComponent = cbox;
};
```

The constructor expects three arguments: a pointer to the component which is posting the event, a pointer to the request message whose acceptance is reported by the component, and a pointer to a Cbox. The *requestedMethod* instance variable is assigned a pointer to the requested method in the request message (obtained by invoking the *Method* operation on the request message). The pointer to the component posting the event message is stored in the *messageFrom* instance variable. The pointer to the request message is assigned to the *theRequestMessage* instance variable. Both the *scheduleMethod* and the *methodTerminated* instance variables are assigned zeros. The pointer to the Cbox is assigned to the *informComponent* instance variable. The latter Cbox is used by the request handler to block for a reply from the CE object. When a scheduling decision is made by the CE object, this Cbox is assigned an arbitrary value in order to awaken the blocked request handler in the component.

```
GroupComponent* EventMessage::WhichComponent() {
    return messageFrom; };
```

The *WhichComponent* method, defined above, is used to return a pointer to the component which posted the event message.

```
void EventMessage::Schedule() {
    scheduleMethod = 1;
    informComponent→Send(1);
};
```

The *Schedule* method, defined above, is used to record the scheduling decision of the CE object in an event message and to unblock the request handler in the component. As a result, the *scheduleMethod* instance variable is assigned a value of 1 indicating that the CE object has allowed the request message to be scheduled for execution. Next, the *Send* method is invoked on the Cbox stored in *informComponent* in order to unblock the request handler. Note that the value (in this case, 1) sent to the Cbox plays no significant role.

```
void EventMessage::Terminated() {
    methodTerminated = 1;
};
```

The *Terminated* method, defined above, is used by the request handler in a component to indicate the fact that the requested method has terminated. As a result, the method assigns the *methodTerminated* instance variable a value of 1.

```
void ElementaryEvent::Schedule()
   { eventMessage->Schedule(); };
```

Utilizing the operations of an event message, the *Schedule* method of the *ElementaryEvent* class is implemented as shown above. The method invokes the *Schedule* method on the *eventMessage* instance variable which stores a pointer to the event message which has been observed most recently by the event object.

```
void EventMessage::AddArguments(int numOfArguments, ArgumentList* argumentList)
{
   int i;
   for (i = 0; i < numOfArguments; i++)
      theRequestMessage->AddArgument(argumentList[i]);
};
```

The *AddArguments* method of the *EventMessage* class, defined above, expects two arguments: the number of arguments to be added to the request message and a pointer to an array containing the argument values. Using the two arguments, it repeatedly invokes the *AddArgument* method on the request message stored in the *theRequestMessage* instance variable.

```
void ElementaryEvent::
   AddArguments(int numOfArguments, ArgumentList* argumentList) {
      eventMessage→AddArguments(numOfArguments, argumentList); };
```

Utilizing the *AddArguments* method of an event message, the *AddArguments* method of the *ElementaryEvent* class is implemented as shown above. It invokes the *AddArguments* method on the event message stored in the instance variable *eventMessage* and supplies *argumentList* and *numOfArguments* as parameters to the method.

5.2.3 The CompositeEvent Class

CEOs are instantiated from the *CompositeEvent* class. A partial definition of the class is shown below.

```
enum ObservedEventType {acceptance, termination, noevent};
class CompositeEvent : public Event {
   protected:
            ceoUsage useToObserve;
            ObservedEventType eventObserved;
            int blockComponents;
                  ...
   public:
            CompositeEvent(ElementaryEvent*, ...);
            CompositeEvent();
            void AssignConstituents(Object*, ...);
            GroupComponent* WhichComponent();
            Event* EventUsage(ceoUsage);
            int IsAccept();
            int IsTerminate();
            void Block();
            void Unblock();
            void AwaitTermination(CoordinatingEnvironment*);
            void Schedule();
            void AddArguments(int, ArgumentList*);
                  ...
}
```

Like elementary events, composite events may be constructed with or without its constituents, as shown below.

```
CompositeEvent:: CompositeEvent(ElementaryEvent* e, ...) {
    // Extract all pointers to EEOs from stack and form "eeoList";
    for each EEO, "eeo" in "eeoList":
        constituents→AddToList(eeo);
    numOfConstituents = constituents→Length();
    useToObserve = noevent;
    eventObserved = noevent;
    blockComponents = 0; };


CompositeEvent::CompositeEvent() {
    useToObserve = noevent;
    eventObserved = noevent;
    blockComponents = 0; };


void CompositeEvent:: AssignConstituents(Object* a, ...) {
    // Extract all pointers to EEOs from stack and form "eeoList";
    for each EEO, "eeo" in "eeoList":
        constituents→AddToList(eeo);
    numOfConstituents = constituents→Length(); };
```

If a list of EEOs is provided to the constructor, that list is assigned to *constituents*. If no arguments are provided to the constructor, then the *AssignConstituents* method must be invoked to initialize *constituents* before the CEO may be used to observe events.

Depending on the states of the constituent EEOs, a CEO may be used to observe only an acceptance event, or only a termination event, or either an acceptance or a termination event. The intended use of a CEO must be explicitly specified before each use of the object.

5.2.3.1 The WhichComponent, EventUsage, IsAccept, and IsTerminate Operations

```
GroupComponent* CompositeEvent::WhichComponent() {
   return (GroupComponent*)
      (((ElementaryEvent*)currentConstituent)→WhichComponent()); };
```

The *WhichComponent* method, shown above, is used to retrieve a pointer to the component in which the most recent event observation was made by a CEO. Since the actual observation is made using an EEO, the method invokes the *WhichComponent* method on that EEO (recorded in *currentConstituent*) to retrieve the desired component.

```
Event* CompositeEvent::EventUsage(ceoUsage t) {
   if (t == accept)
      { useToObserve = t;
        return (Event*) this; };
   else if ((t == terminate) || (t == acceptTerminate)) {
      useToObserve = t;
      constituents→Reset();
      ElementaryEvent* eeo = ((ElementaryEvent*)constituents)→UseNextItem();
      for (; eeo; eeo = ((ElementaryEvent*)constituents)→UseNextItem())
         if (eeo→IsEventPosted())
            eeo→EventUsage(terminate);
      return (Event*) this; };
   else ERROR; };
```

Using the *EventUsage* method, *useToObserve* may be set to any one of these three values: *accept* (to observe a message-acceptance event), *terminate* (to observe a method-termination event), or *acceptTerminate* (to observe either a message-acceptance or a method-termination event). When *useToObserve* is set to *accept*, the CEO is used to observe an acceptance event using an EEO which has not been used to observe an acceptance event. When *useToObserve* is set to *terminate*, the CEO is used to observe a termination event using an EEO using which an acceptance event has been observed. When *useToObserve* is set to *acceptTerminate*, the CEO is used to observe either an acceptance event or a termination event. When *useToObserve* is set either to *terminate* or *acceptTerminate*, the usage type of the EEOs using which acceptance events have already been observed is set to *terminate.* Note that after every successful event observation that is made using a CEO, *useToObserve* is set to *noevent.*

When *useToObserve* is set to *acceptTerminate* and a successful observation is made using the CEO, the event that was observed (either acceptance or termination) is recorded in the *eventObserved* instance variable. The methods *IsAccept* and *IsTerminate* may be used to interrogate the status of *eventObserved* and return the appropriate boolean answer, as shown below.

```
int CompositeEvent::IsAccept() {
    if (eventObserved == acceptance) {
        eventObserved = noevent;
        return 1; };
    else return 0; };


int CompositeEvent::IsTerminate() {
    if (eventObserved == termination) {
        eventObserved = noevent;
        return 1; };
    else return 0; };
```

Once the status has been determined, *eventObserved* is set to *noevent* in both the above methods,

5.2.3.2 The Block, Unblock, and AwaitTermination Operations

```
void CompositeEvent::Block() {
    blockComponents = 1;
    constituents→Reset();
    ElementaryEvent* eeo = (ElementaryEvent*) constituents→UseNextItem();
    for (; eeo; eeo = (ElementaryEvent*) constituents→UseNextItem())
        eeo→Block();
};
```

```
void CompositeEvent::Unblock() {
   blockComponents = 0;
   constituents→Reset();
   ElementaryEvent* eeo = (ElementaryEvent*) constituents→UseNextItem();
   for (; eeo; eeo = (ElementaryEvent*) constituents→UseNextItem())
      eeo→Unblock();
};
```

The realization of the *Block* and the *Unblock* methods are shown above. The CEO invokes the *Block* method in every constituent EEO to block the method whose acceptance is observed by the EEO. Similarly, it invokes the *Unblock* method in every constituent EEO to unblock the blocked method.

```
void CompositeEvent::AwaitTermination(CoordinatingEnvironment* ce) {
   Schedule( );
   ce→Observe(((ElementaryEvent*)currentConstituent)→EventUsage(terminate)); };
```

The realization of the *AwaitTermination* method is shown above. Unlike the similar method in the *ElementaryEvent* class, the above method uses the *currentConstituent* variable to mark the EEO using which the termination event will be observed.

5.2.3.3 The Schedule and AddArguments Operations

```
void CompositeEvent::Schedule()
   { ((ElementaryEvent*)currentConstituent)→Schedule(); };
```

The realization of the *Schedule* method is shown above. The *currentConstituent* instance variable, which points to the constituent EEO using which the last acceptance event was observed, is used to invoke the *Schedule* method. Note that a CEO must be used to schedule the most recent acceptance event before the next event is observed using it, since the value of *currentConstituent* changes after every acceptance-event observation by a CEO.

```
void CompositeEvent::
   AddArguments(int numOfArguments, ArgumentList* argumentList) {
      ((ElementaryEvent*)currentConstituent)→
         AddArguments(numOfArguments, argumentList); };
```

Utilizing the *AddArguments* method of an EEO, the *AddArguments* method of the *CompositeEvent* class is implemented as shown above. It invokes the *AddArguments* method on the EEO stored in the instance variable *currentConstituent* and supplies a list of arguments and the number of arguments as parameters.

5.3 Using the Event Objects

The event objects of either type must be used in predefined ways to satisfy the constraints of the CEs model and the constraints of the current realization. In the following sections, the lifecycle diagrams of single-constituent and multi-constituent elementary event objects and that of composite event objects are shown. Although similar to the lifecycle diagrams described in Chapter Two, these diagrams show how the values of the instance variables change as operations are applied on the event objects. These diagrams are also utilized to guide the realization of the *Observe* operation described in Section 5.4.2.

5.3.1 Lifecycle of a Single-Constituent Elementary Event Object

Figure 5.1 shows the lifecycle of a single-constituent elementary event object instantiated from the *ElementaryEvent* class. The letters *A* through *H* mark the private instance variables of the *ElementaryEvent* class. The different states in the lifecycle are shown as boxes. The initial state of the lifecycle diagram is the box marked 1. Labeled arcs emanating from a box mark the state transitions. The labels can be one of either the public operations of the *ElementaryEvent* class, or the *Observe* operation, or $\overline{\textit{Observe}}$ which indicates the failure of the *Observe* operation to observe an event using the event object. For all objects instantiated from the *ElementaryEvent* class, it will be assumed that the methods *AssignConstituents* and *AssignMethod* have already been applied in order to reach the initial state of the diagram. In states other than the initial state, only those instance variables whose values change due to the execution of an operation are shown.

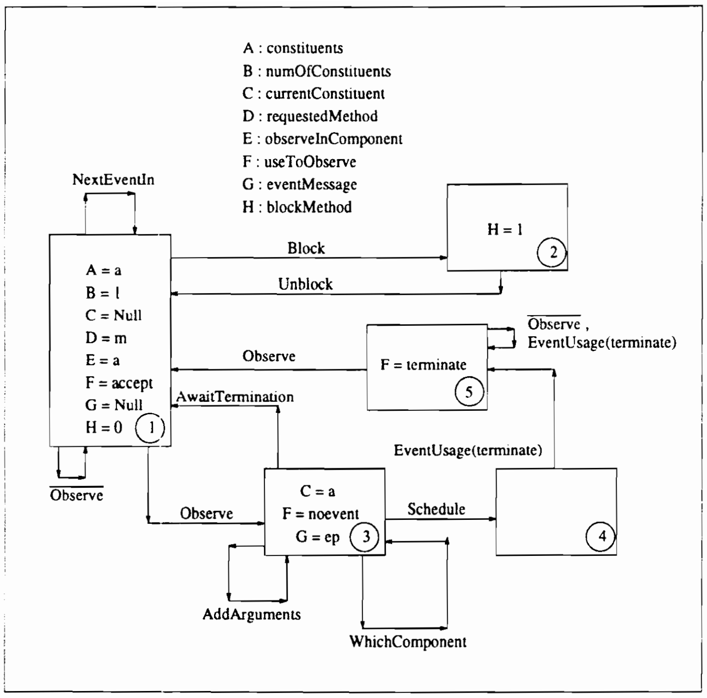

Figure 5.1: Lifecycle of a single-constituent elementary event object.

The assignments of the instance variables in the initial state has the following interpretation. There is only one constituent marked a and stored in A. Number of constituents, B, is 1 and the current constituent, C, is *Null.* The method, m, whose acceptance and termination must be observed by the event object is stored in D. The component in which events must be observed (a, since there is only one component) is stored in E. The default usage mode of the event object, *accept*, is stored in F. No posted event is recorded in G indicating that the event object is ready to be used for observation. H is assigned the value zero indicating that the event object may not be used to block acceptance events. Out of the eight instance variables, only the values of four variables, C, F, G, and H, may change due to state transitions.

An event object in its initial state may be used in one of three ways. First, the *NextEventIn* operation may be invoked on it with a pointer to the component recorded in the EEO as an argument. Note that for single-constituent EEOs, this operation plays no significant role since the *observeInComponent* permanently records the only component. Second, it may be used to block event postings by invoking the *Block* method which causes a transition to state 2 in which H is set to 1. When the *Unblock* method is invoked in state 2, the event object returns to the initial state and the value of H is reset to zero (box 1). The third way of using an event object which is in its initial state is to use it in an *Observe* operation. If an event is observed using the event object, then the *Observe* arc causes a transition to state 3. If the event object is not used to observe an event, then the event object remains in the initial state when *Observe* terminates (specified by the $\overline{Observe}$ transition).

In state 3, C, F, and G assume new values. G records a pointer to the event posting (ep), C records the component, a, in which the event took place, and F is set to *noevent* indicating that the usage type of the event object must be set once again before its next use. Also, in this state, the *AddArguments* and *WhichComponent* methods may be used to append argument values to the request message whose acceptance was observed and to retrieve a pointer to the event posting component, respectively. Both these methods may be applied since G records a pointer to the posted event.

After observing an acceptance event, from state 3, either the *AwaitTermination* method may be invoked on the object to both schedule the request message for execution and await the termination of the method or the *Schedule* method may be invoked to only schedule the request message for execution. In the former case, when the scheduled method terminates, *AwaitTermination* terminates and the event object reverts back to the initial state and, in the latter case, when *Schedule* terminates, the event object transits to state 4 in which none of the instance variables change values (shown by an empty box). From state 4, the only possible operation on the event object is to use the *EventUsage* method to set the usage type of the object to *terminate* (to observe termination) and transit to state 5 in which F records the new usage type.

In state 5, two operations are available. First, the *EventUsage* method may be invoked to set the usage type of the event to *terminate* without causing any state transition. Second, the *Observe* operation may be invoked to observe a termination event using the event object. If a termination event is observed successfully using the event object then the event object transits to the initial state. If not, the event object remains in state 5 (specified by the $\overline{Observe}$ transition) and waits to be used in a subsequent *Observe* operation.

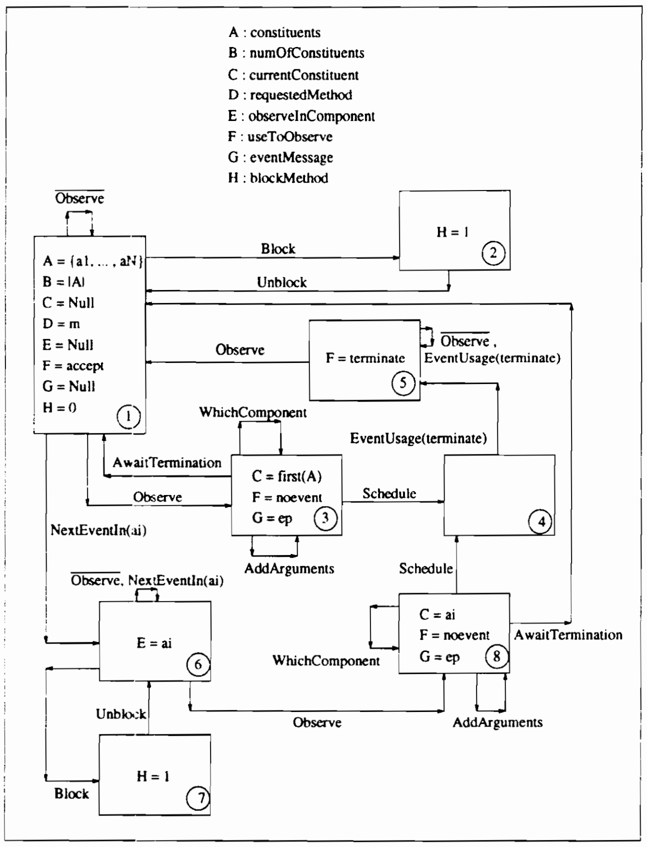

Figure 5.2: Lifecycle of a multi-constituent elementary event object.

5.3.2 Lifecycle of a Multi-Constituent Elementary Event Object

Figure 5.2 shows the lifecycle diagram of a multi-constituent elementary event object instantiated from the *ElementaryEvent* class. The interpretation of A through H remains the same as in Figure 5.1 but in the initial state (box 1), A is initialized to a list of component pointers (shown in the set notation), B records the length of this list, and E is set to *Null.*

The *Block-Unblock* loop from the initial state is similar to the one in Figure 5.1, except that event postings from all the components recorded in A are blocked. The *Observe-AwaitTermination* loop (box 1 - box 3 - box 1) from the initial state has the same interpretation as in Figure 5.1 except that C records the pointer to one of the constituents in A in which an event was observed (denoted by *first(A)*). Since, in the initial state, no component is specified in which an event must be observed, the *Observe* operation tries to observe an event in any one of the components recorded in A and the first component in which an event can be observed is recorded in C. The $\overline{\textit{Observe}}$ transition in the initial state, as before, specifies the state of the object when no event is observed using it. Finally, the *Observe-Schedule-EventUsage(terminate)-Observe* loop (box 1 - box 3 - box 4 - box 5 - box 1) from the initial state has the same interpretation as in Figure 5.1.

A new transition which is possible from the initial state is by using the *NextEventIn* method to transit to state 6. The latter method is used to specify the component in which the next acceptance event must be observed. The component is recorded in E in state 6. From state 6, the *Block-Unblock* loop (box 6 - box 7 - box 6) may be initiated which blocks and unblocks event postings from only the component recorded in E. Also, the *NextEventIn* method may be invoked in state 6 without causing any state transition.

Among the other possible transitions which may be initiated from state 6 are *Observe-AwaitTermination* (box 6 - box 8 - box 1) and *Observe-Schedule-EventUsage(terminate)-Observe* (box 6 - box 8 - box 4 - box 5 - box 1). In the first case, an acceptance event in the component recorded in E is observed, the request message is scheduled for execution, and the termination of the scheduled method is observed. In the second case, an acceptance event in the component recorded in E is observed, the request message is scheduled for execution, then the event object is marked so as to observe a termination event, and finally it is used in an *Observe* operation to observe the termination of the scheduled method. If the *Observe* operation from state 6 does not use the event object to observe an event, then the object remains in state 6 and waits to be used in a subsequent *Observe* operation.

5.3.3 Lifecycle of a Composite Event Object

Figure 5.3 shows the lifecycle of an event object instantiated from the *CompositeEvent* class. The letters A through F mark the private instance variables of the class. A through C have the same interpretation as in Figure 5.2 except that A is a list of elementary event objects. D records whether the event object must observe either an acceptance, or a termination, or either of an acceptance or a termination event. In the case of the latter usage, E records which of the two possible events was actually observed. In the initial state (box 1), D is set to *noevent* indicating that the usage type of a CEO must be set before observing an event using it. F, when set to 1, indicates that the CEO is being used to block acceptance events in components. The history variable X records the number of acceptance events which have been observed using a CEO and the observation capacity variable Y records the maximum

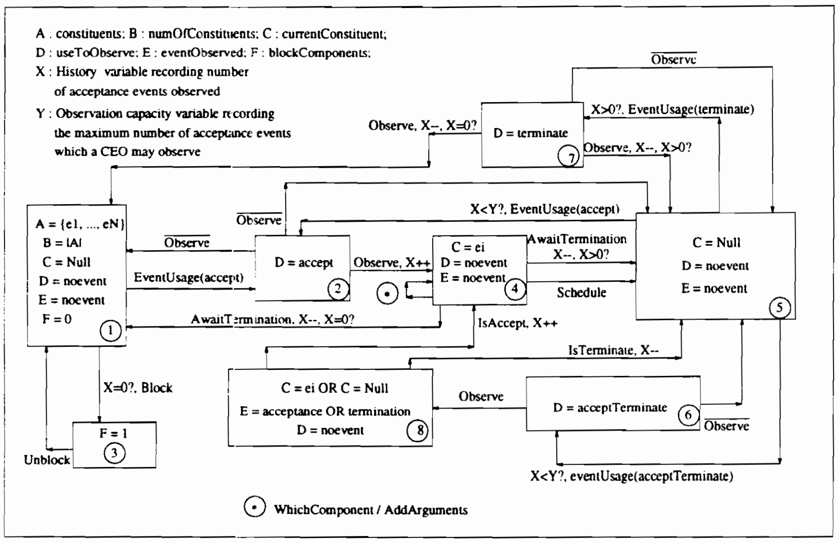
A : constituents; B : numOfConstituents; C : currentConstituent;
D : useToObserve; E : eventObserved; F : blockComponents;
X : History variable recording number of acceptance events observed
Y : Observation capacity variable recording the maximum number of acceptance events which a CEO may observe
$\overline{\text{Observe}}$
X>0?, EventUsage(terminate)
Observe, X--, X=0?
D = terminate
7
Observe, X--, X>0?
X<Y?, EventUsage(accept)
A = {e1, ..., eN}
B = |A|
C = Null
D = noevent
E = noevent
F = 0
1
$\overline{\text{Observe}}$
$\overline{\text{Observe}}$
EventUsage(accept)
D = accept
2
Observe, X++
C = ei
D = noevent
E = noevent
4
AwaitTermination X--, X>0?
Schedule
C = Null
D = noevent
E = noevent
5
AwaitTermination, X--, X=0?
IsAccept, X++
IsTerminate, X--
X=0?, Block
F = 1
3
Unblock
C = ei OR C = Null
E = acceptance OR termination
D = noevent
8
Observe
D = acceptTerminate
6
$\overline{\text{Observe}}$
X<Y?, eventUsage(acceptTerminate)
WhichComponent / AddArguments

Figure 5.3: Lifecycle of a composite event object.

number of acceptance events which may be observed using a CEO. Note that these are not instance variables of a CEO and must be maintained externally. The different values of X and Y are used as pre-conditions, post-conditions and post-actions to the public operations of the *CompositeEvent* class and to the *Observe* operation. For all objects instantiated from the *CompositeEvent* class, it will be assumed that the method *AssignConstituents* has already been applied in order to reach the initial state of the diagram (box 1).

A possible transition from state 1 is to invoke the *Block* method and transit to state 3. This transition can be made only if X has a value of zero (that is, no event postings have been observed using the event object). Hence, $X=0$ followed by a "?" appears as the pre-condition for applying the *Block* method. In state 3, F, set to 1, records the new usage type of the object. The only transition possible from state 3 is to invoke the *Unblock* method and transit to state 1.

The other transition possible from the initial state is to use the *EventUsage* method to set the usage type of the object to *accept* (to observe an acceptance event). This invocation causes a transition to state 2 in which D records the new usage type.

From state 2, the only transition possible is by using the event object in an *Observe* operation. If an acceptance event is observed using the composite event object, X is incremented as a post-action and the state of the object transits to state 4. If the event object is not used to observe an event, then it transits to state 1. In state 4, C stores the pointer to the EEO *ei* using which the observation was made and D is set to *Null* indicating that the next usage type of the event object must be set.

The first transition possible from state 4 is through the invocation of the *AwaitTermination* method to schedule the request message observed by the EEO *ei* and wait for the termination of the corresponding method. As a post-action, X is decremented by one and depending on whether X is zero or non-zero, the state of the object either transits to state 1 or to state 5, respectively.

The second transition possible from state 4 is through the invocation of the *Schedule* method which schedules the request message observed by the EEO *ei*. Note that on scheduling the request message, the event object transits to state 5 in which C is reset to *Null*. This frees up the event object to be used to observe the subsequent events in any of its constituents and not in the component in which the last acceptance was observed.

Among the other transitions possible from state 4 are through the invocations of the *WhichComponent* and *AddArguments* methods. Both the latter methods use the EEO recorded in C to either return a pointer to the component that posted an event or to append arguments to the request message recorded in *ei*. The object remains in state 4 as a result of invoking these two methods.

There are three possible transitions from state 5. The first transition, to state 7, marks the usage type of the object to *terminate* using the *EventUsage* method if $X > 0$ (that is, if there is any termination event to be observed). The second transition, to state 2, marks the event object to observe an acceptance event using *EventUsage* if all the EEOs have not been used up in observing acceptance events (that is $X < Y$ is true). Note that, in this case, if the CEO is not used to observe an event in state 2, then the state of the CEO transits back to state 5. The third transition, to state 6, marks the event object to observe either an acceptance or a termination event only if there is at least one EEO available which can observe an acceptance (that is, $X < Y$ is true) (otherwise, one may try to observe more acceptance events than there are EEOs in a CEO).

There are two possible transitions from state 7. An event object in this state may be used in an *Observe* operation in order to observe a termination event. If the event object is used to observe such an event, then as a post-action, X is decremented. Then depending on whether the new value of X is zero or greater, the event object transits to either state 1 or state 5, respectively. If the event object is not used to observe an event, then its state reverts back to state 5.

The only transition possible from state 6 is to state 8 through the successful observation of either an acceptance or a termination event by an *Observe* operation. Note that X is not incremented as a post-action when *Observe* terminates because which of the two possible events was observed cannot be inferred. As a result, C, in state 8, may either store a pointer to an EEO (indicating that an acceptance event was observed) or store *Null* (indicating a termination event was observed). E, in state 8, stores either the value *termination* (indicating method termination) or the value *acceptance* (indicating message acceptance). If the event object is not used to observe an event, then its state reverts back to state 5.

The two possible transitions from state 8 inquire whether an acceptance or a termination event was observed using the event object. If acceptance was observed, the method *IsAccept* returns true, X is incremented as a post-action, and a transition is made to state 4. If termination was observed, the method *IsTerminate* returns true, X is decremented as a post-action, and a transition is made to state 5.

5.4 The Design of CE Objects

The behavior of CE objects is captured by the *CoordinatingEnvironment* class. Since CE objects and event objects work very closely to realize non-intrusive coordination, the operations of the *CoordinatingEnvironment* class relies heavily upon several operations of the event objects. In the following, first, a partial definition of the *CoordinatingEnvironment* class is introduced and a few of its operations described. Next, how the *ElementaryEvent*, the *CompositeEvent*, and the *CoordinatingEnvironment* classes must be extended to realize "observation" is described. Finally, the *Observe* operation which realizes "observation" is described.

5.4.1 The CoordinatingEnvironment Class

A partial definition of the *CoordinatingEnvironment* class is shown below.

```
class CoordinatingEnvironment : public Object {
                    ...
  public:
      void Ignore(methodId, GroupComponent*, ...);
      void AddArguments(Event*, ...);
      void Become(methodId, ...);
      void Observe(Event*, ...);
                    ...
}
```

It defines the methods which realize three coordinating actions: *Ignore*, *AddArguments*, and *Become*, and the *Observe* operation, which implements the search algorithm that matches event objects with event postings.

```
void CoordinatingEnvironment::Ignore(methodId m, GroupComponent* c1, ...) {
  // Extract all pointers to components from stack and form "componentList";
  for each component, "comp", in "componentList":
    comp→InhibitMethod(m);
};
```

The *Ignore* method, shown above, is used to inhibit the posting of the acceptance event associated with a specific method by one or more components. The method associated with which events must be inhibited and the components which are affected are supplied as parameters. If there is more than one component involved, the method extracts the pointers from the stack. The actual inhibition is done by invoking the method *InhibitMethod* in each component and passing the method identifier as an argument.

```
void CoordinatingEnvironment::AddArguments(Event* eventObject, ...) {
  // Extract all argument values from stack and form "argumentList";
  // Store number of arguments extracted in "numOfArguments";
  eventObject→AddArguments(numOfArguments, argumentList); };
```

The *AddArguments* method, shown above, is used to add arguments to the argument list of a request message. The method extracts all the arguments to be added to a request message from the stack and stores them in the list *argumentList*, and also stores the number of arguments in the variable *numOfArguments*. It then invokes the *AddArguments* method on the event object supplied in the argument *eventObject*.

```
void CoordinatingEnvironment::Become(methodId* cbMethod, ...) {
  // Extract all argument values from stack and form "argumentList";
  // Store number of arguments extracted in "numOfArguments";
 // Instantiate a Callstate object using cbMethod, argumentList, and numOfArguments;
  // Start a thread in the RequestHandler method
  //    of the CoordinatingEnvironment class;
};
```

The *Become* method, shown above, is used to execute the initial or a replacement CB method. It accepts as arguments a method reference corresponding to a CB method and any arguments that are required by the CB method to execute. *Become* extracts the reference of the method and the arguments, if any, from the stack, and schedules it for execution.

The *Observe* method, described in section 5.4.2.4, accepts an arbitrary number of event-object pointers as arguments and applies a predefined algorithm to extract an event posting which matches with one of the event objects. It is a blocking operation which terminates only after an observation has been made. It returns a pointer to the event object using which an observation was made.

5.4.2 Realizing Event Observation

In this section, an implementation of the *Observe* operation is sketched to show the feasibility of its realization. In order to do so, first, the *ElementaryEvent*, the *CompositeEvent*, and the *CoordinatingEnvironment* classes are augmented as shown in the following sections. Note that the *CoordinatingEnvironment* class is declared as a *friend* class of the *ElementaryEvent* and the *CompositeEvent* classes which allows the *Observe* operation to use the protected methods of these two classes.

5.4.2.1 Augmenting the ElementaryEvent Class

The *ElementaryEvent* class is extended by adding a few protected methods. A partial definition of the class showing only the additional methods appears below.

```
class ElementaryEvent : public Event {
  protected:
          int ObservedTermination();
          int IsElementaryEvent();
          void BindEventMessage(EventMessage*);
          void SeverEventMessage();
          int IsMatch(EventMessage*);
                  ...
};
```

The *ObservedTermination* method determines whether an EEO has observed a termination event after it has been used to observe an event by the *Observe* operation. If so, the method returns a non-zero value, as shown below.

```
int ElementaryEvent::ObservedTermination() {
  if (useToObserve == terminate)
    return 1;
  else return 0; };
```

The *IsElementaryEvent* method determines whether an event object is an elementary or a composite event object. The method defined in the *ElementaryEvent* class returns a non-zero value, as shown below.

```
int ElementaryEvent::IsElementaryEvent() {
  return 1; };
```

The *BindEventMessage* method is used to bind an event message to an event object which has been observed using that event object. This binding enables a CE object to take coordinating actions on the coordinated components through the event object. The method is implemented as follows:

```
void ElementaryEvent::BindEventMessage(EventMessage* postedEvent) {
    eventMessage = postedEvent;
    currentConstituent = (Object*)(eventMessage→WhichComponent());
    useToObserve = noevent; };
```

The above method realizes the new assignment of values to the private instance variables of an EEO shown in box 3, in Figure 5.1, and in boxes 3 and 8, in Figure 5.2. The posted event message is recorded in *eventMessage*, the component posting the event message is recorded in *currentConstituent* by extracting a pointer to the component from the event message using the *WhichComponent* method, and the usage type of the EEO is reset to *noevent*.

The *SeverEventMessage* method is used to sever the binding between an event message and an event object. The binding is severed after the termination of the requested method is observed by a CE object and is done so that the event object may be reused to observe other events.

```
void ElementaryEvent::SeverEventMessage() {
    eventMessage = Null;
    currentConstituent = Null;
    useToObserve = accept;
    if (numOfConstituents > 1)
        observeInComponent = Null; };
```

The above method realizes the new assignment of values to the private instance variables of an EEO shown in box 1 (after the transitions from box 5 and box 3), in Figure 5.1, and in box 1 (after the transitions from box 5 and box 3), in Figure 5.2. The link to the posted event message is removed from *eventMessage*, the link to the component which posted the event message is removed from *currentConstituent*, and the usage type of the EEO is set to *accept*. If the EEO is a multi-constituent one, then *observeInComponent* is set to null.

The *IsMatch* method is used to determine whether the event message supplied as an argument to the method "matches" with the EEO. The exact matching criterion is described below. If there is a match, the method returns a non-zero value, otherwise it returns a zero.

```
int ElementaryEvent::IsMatch(EventMessage* eventMessage) {
    if (requestedMethod == eventMessage→requestedMethod)
        if (observeInComponent)
            if (observeInComponent == eventMessage→messageFrom)
                if (useToObserve == terminate)
                    if (eventMessage→methodTerminated)
                        return 1;
                    else return 0;
                else return 1;
            else return 0;
        else
        { constituents→Reset();
       GroupComponent* component = (GroupComponent*) constituents→UseNextItem();
       for (; component; component = (GroupComponent*) constituents→UseNextItem())
                if (component == eventMessage→messageFrom)
                    if (useToObserve == terminate)
                        if (eventMessage→methodTerminated)
                            return 1;
                        else return 0;
                    else return 1;
            return 0;
        };
};
```

To declare that an event message "matches" the EEO, it must satisfy at least two and at most most three criteria. The two criteria which must be satisfied are, first, the method pointer stored in the EEO must match the method pointer stored in the event message, and, second, the component pointer stored in the event message must match with one of the component pointers stored in the EEO. Note that in case of a single-constituent EEO or if a specific component has been marked for observation in a multi-constituent EEO, then the *ObserveInComponent* variable will store the pointer to the component. Otherwise, the method has to search through all the component pointers stored in the *constituents* list for a match. A third matching criterion is applied only when the EEO is marked to observe a termination event. In that case, if the event message has the *methodTerminated* instance variable set to true, there is a match. Otherwise there is a mismatch.

5.4.2.2 Augmenting the CompositeEvent Class

The *CompositeEvent* class is extended by adding a few protected methods. A partial definition of the class showing only the additional methods appears below.

```
class CompositeEvent : public Event {
   protected:
            int ObservedTermination();
            int IsElementaryEvent();
            void BindEventMessage(EventMessage*);
            void SeverEventMessage();
            void ResetEventUsage();
            int IsMatch(EventMessage*);
                    ...
};
```

The *ObservedTermination* method determines whether a CEO has observed a termination event after it has been used to observe an event by the *Observe* operation. In order to determine that, it invokes the *ObservedTermination* method in the EEO stored in the *currentConstituent* instance variable. If termination has been observed, the method returns a non-zero value, as shown below.

```
int CompositeEvent::ObservedTermination() {
   if (((ElementaryEvent*)currentConstituent)→ObservedTermination())
      return 1;
   else return 0; };
```

The *IsElementaryEvent* method determines whether an event object is an elementary or a composite event object. The method defined in the *CompositeEvent* class returns a zero, as shown below.

```
int CompositeEvent::IsElementaryEvent() {
   return 0; };
```

The *BindEventMessage* method is similar to the method with the same name in the *ElementaryEvent* class, except that the event message must be bound to the EEO (in the CEO) which has observed the most recent acceptance event. The method is implemented as follows:

```
void CompositeEvent::BindEventMessage(EventMessage* postedEvent) {
   if (useToObserve == acceptTerminate) {
      useToObserve = noevent;
      eventObserved = acceptance; };
   else {
      useToObserve = noevent;
      eventObserved = noevent; };
   ((ElementaryEvent*)currentConstituent)→BindEventMessage(postedEvent); };
```

The above method realizes the new assignment of values to the private instance variables of a CEO shown in box 8 and box 4 (after the transition from box 2), in Figure 5.3. If the usage type of the CEO is *acceptTerminate*, then *eventObserved* is set to *acceptance* indicating that an acceptance event was observed (event messages are bound only when acceptance events occur) and *useToObserve* is reset to *noevent*. If the usage type of the CEO is not *acceptTerminate*, then *eventObserved* is set to *noevent* and *useToObserve* is reset to *noevent*. The posted event message is recorded in the EEO which observed the event by invoking the *BindEventMessage* method on the *currentConstituent* instance variable. Note that it is assumed that the *currentConstituent* variable points to the relevant EEO when the above method is invoked.

The *SeverEventMessage* method is similar to the method with the same name in the *ElementaryEvent* class, except that the event message must be severed from the EEO (in the CEO) which has observed the most recent method termination event. The method is implemented as follows:

```
void CompositeEvent::SeverEventMessage() {
   if (useToObserve == acceptTerminate) {
      useToObserve = noevent;
```

```
        eventObserved = termination; };
    else {
        useToObserve = noevent;
        eventObserved = noevent; };
    ((ElementaryEvent*)currentConstituent)→SeverEventMessage(); };
    currentConstituent = Null; };
```

The above method realizes the new assignment of values to the private instance variables of a CEO shown in box 8, in box 1 (after the transition from box 7), and in box 5 (after the transition from box 7). If the usage type of the CEO is *acceptTerminate*, then *eventObserved* is set to *termination* indicating that a termination event was observed (event messages are severed only when termination events occur) and *useToObserve* is reset to *noevent*. If the usage type of the CEO is not *acceptTerminate*, then *eventObserved* is set to *noevent* and *useToObserve* is reset to *noevent*. The link to the posted event message is severed in the EEO which observed the termination event by invoking the *SeverEventMessage* method on the *currentConstituent* instance variable. After the latter invocation terminates, *currentConstituent* is reset to *Null*.

The *ResetEventUsage* method sets the usage type of a CEO to *noevent*, as shown below.

```
int CompositeEvent::ResetEventUsage() {
    useToObserve = noevent; };
```

The above method is used to reset the usage type of a CEO which was used in an *Observe* operation but was not successful in observing an event. The setting of the *useToObserve* variable corresponds to the transitions to box 5 (from box 2, box 7 and box 8) and to box 1 (from box 2).

The *IsMatch* method is used to determine whether the event message passed as an argument to the method "matches" with the CEO. The exact matching criterion is described below. If there is a match, the method returns a non-zero value, otherwise it returns a zero.

```
int CompositeEvent::IsMatch(EventMessage* eventMessage) {
    ElementaryEvent* eeo;
    constituents→Reset();
    if (useToObserve == accept) {
        eeo = (ElementaryEvent*) constituents→UseNextItem();
        for (; eeo; eeo = (ElementaryEvent*) constituents→UseNextItem())
            if !(eeo→ObservedTermination())
                if (eeo→IsMatch(eventMessage)) {
                    currentConstituent = eeo;
                    return 1;
                };
    };
    else if (useToObserve == terminate) {
        eeo = (ElementaryEvent*) constituents→UseNextItem();
        for (; eeo; eeo = (ElementaryEvent*) constituents→UseNextItem())
            if (eeo→ObservedTermination())
                if (eeo→IsMatch(eventMessage)) {
                    currentConstituent = eeo;
                    return 1;
                };
    };
    else if (useToObserve == acceptTerminate) {
        eeo = (ElementaryEvent*) constituents→UseNextItem();
        for (; eeo; eeo = (ElementaryEvent*) constituents→UseNextItem())
            if (eeo→IsMatch(eventMessage)) {
                currentConstituent = eeo;
                return 1;
            };
    };
```

An event message matches a CEO only if the CEO has an EEO which matches with the event message. The matching criteria remain the same as described for an EEO. The main function of the CEO is to determine which EEOs should be picked to participate in the matching process. The latter issue is resolved by considering the current usage mode of the CEO and the usage modes of its EEOs. If the CEO is set to observe an acceptance event, then only those EEOs which are ready to observe acceptance events participate in the search. The *IsMatch* method is invoked in every such EEO until a match is found. Once a match is found, the *currentConstituent* variable in the CEO is set to point to the matching EEO and the method returns a non-zero value. The same processing ensues when the CEO is set to observe a termination event and also when the CEO is set to observe either an acceptance or a termination event. The only differences are that, in the former case, only those EEOs which are ready to observe termination participate in the search, and, in the latter case, every EEO participates in the search.

5.4.2.3 Augmenting the CoordinatingEnvironment Class

The *CoordinatingEnvironment* class is extended by adding one protected instance variable and one protected method. A partial definition of the class showing only the additional items appears below.

```
class CoordinatingEnvironment : public Object {
   protected:
      Queue* eventMessages;
      Event* MatchEventMessage(EventMessage*, EventObjectList*);
                  ...
}
```

The *eventMessages* instance variable is a pointer to the queue in a CE object which buffers the event messages posted by components. The *MatchEventMessages* method implements the algorithm using which an event object is matched with an event message. It is implemented as shown below.

```
Event* CoordinatingEnvironment::
   MatchEventMessage(EventMessage* eventMessage, EventObjectList* eventObjects)
   {
      Event* event = (Event*) eventObjects→UseNextItem();
      for (; event; event = (Event*) eventObjects→UseNextItem()) {
         if (event→IsElementaryEvent())
            event= (ElementaryEvent*) event;
         else
            event= (CompositeEvent*) event;
         if (event→IsMatch(eventMessage))
            return (Event*) event;
      };
      return (Event*) 0;
   };
```

The above method expects an event message and a list of event objects as arguments. It tries to locate an event object that matches with the event message by invoking the *IsMatch* method on every event object. If there is a match, the method returns a pointer to the event object. If none of the event objects match with the event message, the method returns a null pointer.

5.4.2.4 The Observe Operation

The *Observe* operation is defined below.

```
Event* CoordinatingEnvironment::Observe(Event* eo1, ...) {
  int i;
  EventMessage* eventMessage;
  Event* eventObject = 0;
  EventObjectList* eventObjects;
  int numOfEventMessages = eventMessages→Length();
  //Extract pointers to event objects from the stack and form the list "eventObjects"
  for ( i = 1; i <= numOfEventMessages; i++) {
    eventMessage = eventMessages→ExtractItem(i);
    if (eventObject = MatchEventMessage(eventMessage, eventObjects)) {
      if (eventObject→IsElementaryEvent())
        eventObject = (ElementaryEvent*) eventObject;
      else
        eventObject = (CompositeEvent*) eventObject;
      if !(eventObject→ObservedTermination()) {
        eventMessages→Remove(eventMessage);
        eventObject→BindEventMessage(eventMessage); };
      else if (eventObject→ObservedTermination()) {
        eventMessages→Remove(eventMessage);
        eventObject→SeverEventMessage(); };
    }; \\ end-if
  }; \\ end-for
  while !(eventObject) {
    int newNumOfEventMessages =
      eventMessages→WaitForNewMessages(numOfEventMessages);
    for (i = numOfEventMessages + 1; i <= newNumOfEventMessages; i++) {
      eventMessage = eventMessages→ExtractItem(i);
      if (eventObject = MatchEventMessage(eventMessage, eventObjects)) {
        if (eventObject→IsElementaryEvent())
          eventObject = (ElementaryEvent*) eventObject;
```

```
            else
               eventObject = (CompositeEvent*) eventObject;
            if !(eventObject→ObservedTermination()) {
               eventMessages→Remove(eventMessage);
               eventObject→BindEventMessage(eventMessage); };
            else if (eventObject→ObservedTermination()) {
               eventMessages→Remove(eventMessage);
               eventObject→SeverEventMessage(); };
         }; \\ end-if
      }; \\ end-for
      numOfEventMessages = newNumOfEventMessages;
   }; \\ end-while
   for each event object "event" in the list "eventObjects":
      if (!(event→IsElementaryEvent()) && !(event == eventObject))
         event→ResetEventUsage();
   return eventObject;
};
```

The very first action in *Observe* is to record the length of the *eventMessages* queue in *numOfEventMessages.* This value is used to determine how many event messages must be processed to detect a matching event posting and also plays a role in determining "new" event messages. Next, a list is formed out of the pointers to the event objects sent as arguments to the *Observe* method and they are stored in the list *eventObjects.* Then, each of the *numOfEventMessages* event message, starting at the head of the *eventMessages* queue, is processed in the following way until a match is found. The *MatchEventMessage* method is invoked and the event message and the list of event objects, *eventObjects*, are sent as arguments to it. If the latter method finds a matching event object in the *eventObjects* list, then it returns a pointer to that event object. Next, the *ObservedTermination* method is invoked on the returned event object. If the latter method returns false, then an acceptance event has been observed, and if the method returns true, then a termination event has been observed. If an acceptance event is observed, the event message, after being removed from the *eventMessages* queue, is bound to the event object which was returned by *MatchEventMessage.* If a termination event is observed, the event message is removed from the *eventMessages* queue and the binding between the event message and the event object which was returned by *MatchEventMessage* is severed.

Since there is no guarantee that the expected event message would be available in the *eventMessages* queue when the *Observe* operation is invoked, the method must wait until a satisfactory observation can be made. Once all the *numOfEventMessages* number of event messages have been searched unsuccessfully for a match, the method enters a loop which is terminated only when a successful observation is made. In each iteration of the loop the following takes place. The *WaitForNewMessages* method is invoked on the *eventMessages* queue and the number of event messages already searched, recorded in *numOfEventMessages*, is supplied as an argument to it. The latter method determines whether the queue size has grown beyond the value supplied as argument. If so, it returns the new size of the queue. Otherwise, it blocks until a new item is enqueued after which it returns the new size. After determining the new size of the queue, each new event message is processed, as before, until a match is found. If a match is found, the *eventObject* variable records a pointer to the event object that matched and that terminates the loop. If no match is found in the newly posted event messages, *numOfEventMessages* is updated to record the total number of event messages processed so far and the loop returns to wait for new event postings.

After a successful observation has been made and before returning the event object, the *Observe* method resets the usage mode of every CEO which participated in the operation but was unsuccessful in observing an event. The latter is done by invoking the *ResetUsageMode* method on each CEO and ensures that the CEO may be used in one of its three usage modes in the next *Observe* operation.

Chapter 6

Synchronizers, ACT, and the CEs Models: A Final Look

6.1 Introduction

The CEs model was motivated by a desire to improve certain aspects of the synchronizers [10] and the *Abstract Communication Types* (ACT) [41] models of coordination. The goal was to design a more flexible hybrid coordination model for object-group coordination. Now that the CEs model has been introduced and its ability to solve non-trivial coordination problems illustrated, a detailed comparison of the features of the CEs model with those of the two other models in its category is made. Such a comparison helps highlight the relative merits of the models and also helps understand the major differences between them. Section 6.2 compares the CEs model with synchronizers and section 6.3 does the same with the ACT model.

6.2 Synchronizers and The CEs Model: A Comparison

In this section, the synchronizers model is contrasted with the CEs model in order to gain a deeper insight into the fundamental issues which differentiate the two models. In the following sections, the three example synchronizers appearing in [10] are reproduced, each followed by a set of observations and a detailed comparison with a solution to the same problem in the CEs model.

6.2.1 Example One : The Vending Machine Synchronizer

Consider the vending machine synchronizer in [10]. The vending machine is considered to be the collection of a coin acceptor object and two slot objects, one containing apples and the other containing bananas. On extracting a fruit, the coin acceptor is cleared (the issue of refunding the excess amount is ignored for simplicity). Pushing a special button on the coin acceptor refunds the inserted amount.

```
VendingMachine(acceptor, apples, bananas, apple_price, banana_price) {
   init amount := 0
(1)      amount < apple_price disables apples.open,
(2)      amount < banana_price disables bananas.open,
(3)      acceptor.insert(v) updates amount := amount + v,
(4)      (acceptor.refund or apples.open or bananas.open)
             updates amount := 0
}
```

Note that the constraints in the above synchronizer are numbered to refer to them. The numbering is not part of the actual syntax and does not specify any ordering among the constraints. A pointer to a coin acceptor *actor* is stored in *acceptor*, and *apples* and *bananas* store pointers to the two slot *actors*. The price of an apple and a banana is stored in *apple_price* and *banana_price*, respectively. The coin acceptor *actor* has two methods, *insert* and *refund*, and each slot *actor* has one method, *open*. The amount inserted in the coin acceptor *actor* is remembered in the private variable *amount*.

The four declarative, *unordered* constraints in the above synchronizer coordinate the components of a vending machine as follows. The constraints which are applicable after an *actor* accepts a message are determined by matching the pattern *actor-pointer.method-name* in all the constraints. Constraint 3 updates the *amount* variable after every acceptance of the *insert* message by the coin acceptor *actor*. The updation of the amount is done using the *updates* operator and by using the actual argument value *v*, that is recorded in the *insert* message. Constraints 1 and 2 determine whether an *open* message must be allowed to proceed by checking whether the amount inserted is greater than the price of the desired item. If the amount is insufficient, the *disables* operator delays the invocation of the *open* method in the slot. In the Actor model, the *disables* operator causes the accepted *open* message to return to the mail queue of the slot *actor*. Constraint 4 assigns zero to the *amount* variable whenever an *open* message is accepted by a slot *actor* or a *refund* message is accepted by the coin acceptor *actor*.

In the following, several observations are made about the above synchronizer and it is compared with the vending machine CE class in Chapter Three.

6.2.1.1 Replication of Instance Variables in Components

The vending machine synchronizer stores the amount inserted by a customer and the price of each item in itself whereas the CE object does not. The CE object allows the amount and the prices to be stored in the respective components. Whenever necessary, the CE object extracts a value from one component and makes it available to another. Although this transfer of data involves extra method invocations, maintaining state that is critical for coordination in a decentralized way has the following advantages. First, there is no replication of data in both a component and the synchronizer. For example, the inserted amount is recorded both in the coin acceptor actor and the synchronizer. The CE class, however, does not record and update the inserted amount as coins are inserted. Second, the privacy of the data sent by a client in a request message is retained. Since the above synchronizer must update the *amount* variable, it uses the argument value in the *insert* message to do that. In the CEs model solution, the argument value in the *insert* message is not accessed by the CE object. Third, the CE class allows a better separation of concerns by allowing components to make their own decisions. For example, whether the amount inserted is sufficient for extracting an item should be decided by a slot and not by the coordinator. The role of the coordinator should be to provide the necessary information to the slot so that it can make its decision. This independence from the details of local decision making has the potential of increasing the reusability of CE classes. Fourth, the issue of maintaining the consistency of replicated data items does not arise in the CEs model solution since instance variables are not replicated. For example, when *amount* is set to zero in the above synchronizer, the amount in the coin acceptor also must be set to zero. It is not clear from the above specification how that is achieved. The vending machine CE object, on the contrary, sends explicit messages to reset the accumulated amount in the coin acceptor.

6.2.1.2 Buffering of Disabled Messages

When an accepted message is disabled by the *disables* operator in a synchronizer, the message is put back in the mail queue for subsequent processing. Some questions which have not been addressed in [10] are: When and how is a disabled message "enabled" (especially when there is no *enables* operator)? Does a synchronizer guarantee the processing of a disabled message? What if a disabled message is from an unsynchronized client?

The CEs model provides concrete solutions to these problems. Message blocking is achieved by the *block* operation and message unblocking is achieved by the *unblock* operation. The CEs model is designed to handle both synchronized and unsynchronized clients by not buffering request messages and making clients responsible for their message dispatching behavior.

6.2.1.3 Unstructured Constraints

The constraints in a synchronizer have no syntactic structuring. As a result, when the number of constraints increase, a synchronizer will be difficult to understand and maintain. Consider a vending machine with 20 slots. There will be 20 constraints similar to constraint 1 and 2 and constraint 4 will contain 21 disjuncts. Moreover, due to the linear specification of the constraints, which observations apply in which state of the group cannot be inferred easily. For example, it is not apparent from the vending machine synchronizer that a successful interaction sequence must start with the observation of an *insert* message. Due to the same lack of structuring, every constraint in a synchronizer is tried even if some constraints are not applicable in a specific state of the group.

CE classes, on the other hand, are structured as a collection of CB methods. That structuring makes CE classes easier to understand and maintain and also allows incremental modification of CB methods using inheritance. Increasing the number of components has no effect on the syntactic complexity of a CE class and each CB method explicitly specifies the applicable event observations using the *Observe* operation. Moreover, only those observations and coordinating actions which are applicable in the current state of the group are recorded in a CB method.

6.2.1.4 Interaction of the Disables and Updates Operations

The *apples.open* pattern appears in both a *disables* and an *update* constraint. The resolution of the order of application of the constraints when a pattern matches both a *disables* and an *updates* constraints is: apply *disable* before *update.* Whether an *update* constraint will always be applied after a *disable* constraint is not explicitly stated in [10]. For example, in the vending machine synchronizer, the *update* constraint must not be applied after a *disable* constraint is applied to *apples.open.* That is because the *update* constraint would incorrectly assign a zero to *amount.* The CEs model does not suffer from such ambiguities because of its imperative nature.

6.2.2 Example Two : The Resource Administrator Synchronizer

Consider the following synchronizer.

```
AllocationPolicy(adm1, adm2, max) {
   init prev := 0
(1)   prev >= max disables (adm1.request or adm2.request),
(2)   (adm1.request or adm2.request) updates prev := prev + 1,
(3)   (adm1.release or adm2.release) updates prev := prev - 1
}
```

The two variables *adm1* and *adm2* store pointers to two resource administrators which allocate two types of resources, say, printers and disks, respectively. There is a limit to how many resources each administrator can allocate (thereby limiting the number of consecutive *request* messages it can accept before accepting a *release* message). In addition to the local constraint (which is managed by an administrator), there is a constraint on the group formed by the two administrators: the combined number of resources allocated by the group cannot exceed a certain value stored in *max.* The instance variable *prev* stores the number of resources which have been allocated by the group.

6.2.2.1 Method Termination Not Observed

A problem with the above synchronizer is that it does not observe method terminations in components. When *max* resources have been allocated and a *release* message is observed, the *prev* variable is decremented by one. That enables the successful observation of an accepted *request* message. If the latter *request* message executes before the former *release* message, then the maximum-resource restriction is violated. Thus, observing termination of methods is an important issue in maintaining the consistency of an object group in an asynchronous and concurrent setting. The CEs model does not suffer from this problem since method-termination events are observable.

6.2.2.2 Constraint Application Ordering Difficult to Understand

Another shortcoming of the above synchronizer is that it is not evident from the specification that a disabled *request* message is enabled once a *release* message is observed. That happens because there is no explicit ordering among constraints 1 and 3. The ordering is implicit and must be inferred by noticing that both the constraints use the variable *prev* and by using ones knowledge about the problem. Also, while *request* messages are blocked, it is not apparent that constraints 1 and 2 are inapplicable. In a CE-based solution, the CB method corresponding to the maximum-allocation state will observe only a *release* message, and on making a successful observation, it will explicitly unblock *request* messages. Thus, CE classes, through their structured approach, increases the ease of understanding of coordination specifications.

6.2.3 Example Three : The Dining Philosophers Synchronizer

Consider the following synchronizer for the dining philosophers problem.

```
PickUpConstraint(c1, c2, phil)
{
(1)     atomic( (c1.pick(sender) where sender = phil),
                (c2.pick(sender) where sender = phil) ),
(2)    (c1.pick where sender = phil) stops
}
```

The above synchronizer is instantiated from the *eat* method of a philosopher *actor*. The *actor* supplies a pointer to itself (which is recorded in the variable *phil*) and pointers to the two fork *actors* it wishes to acquire (which are recorded in *c1* and *c2*, respectively). Thus, for a group of five philosophers, say, who are all trying to acquire forks simultaneously, there will be five active synchronizers coordinating the access to the forks. The *atomic* operator either matches all the patterns in its argument list (in which case, the matching messages are executed) or it fails to match (in which case, the matching messages are delayed). The two patterns in the argument list of the *atomic* operation in constraint 1 determine whether the two forks *c1* and *c2* have accepted *pick* messages sent from the *eat* method of the same philosopher, *phil.* If so, the *pick* messages are executed. Otherwise, they are delayed. Constraint 2 ensures that once the acceptance of a *pick* message is observed in one of the two fork actors desired by a philosopher *actor*, the synchronizer object deletes itself using the *stops* operation.

6.2.3.1 Concurrent Method Invocation and the Atomic Operation

An assumption in [10] is that "the *eat* method **concurrently** invokes *pick* on each of the needed chopsticks". In a concurrent, message passing environment, *concurrent* invocation implies acquiring exclusive rights to send *pick* messages to the *fork* actors so that at any point in time, at least a pair of contiguous forks may process requests from the same philosopher (thereby satisfying the *atomic* operation). In order to gain such exclusive access rights, an *eat* method must enter a critical section in order to send the two *pick* messages to the fork *actors* (see, for example, the solution to the dining philosophers problem using ACT++ [31]). But such a serialization, by itself, guarantees that two forks will be picked up simultaneously by a philosopher *actor* and the role of the synchronizer in forcing the coordination is not apparent in that case. The solution in the CEs model, on the other hand, does not rely on the atomicity of message observations. The CE object coordinates a group of philosopher objects and does not consider fork objects as part of the group.

6.2.3.2 Miscellaneous Observations

Several other points may be noted about the above synchronizer. First, it is not apparent that the constraint which terminates the synchronizer must be executed after the *atomic* constraint has been satisfied. Such constraint execution orderings are implicit and could hinder understanding and maintaining synchronizers. Second, initializing a *PickUpConstraint* synchronizer from the *eat* method of a philosopher object might reduce the reusability of the *philosopher* class. Consider the case when the same philosopher wants to participate in a meal where availability of forks is not an issue; this will cause a redefinition of the *eat* method. Lastly, the encapsulation of the *pick* message is broken because the synchronizer reads the value of the actual argument sent in the request message.

6.3 The ACT Model and The CEs Model: A Comparison

The ACT model is based on the *composition-filters* object model (CFOM). The fundamental feature of the CFOM model which differentiates it from the object model on which the CEs model is based, termed the CE object model (CEOM), is the use of filters. A filter is an object with a message queue and is instantiated from a *filter class*. A filter object, termed an *input filter*, determines whether incoming messages to an object are accepted or rejected and a filter object, termed an *output filter*, is used to affect outgoing messages from an object. Several input and output filters may be associated with a single object. A filter object is associated with *filtering conditions*. A filtering condition is a sequence of *filter elements* where each element consists of three parts: a *condition*, a *matching part*, and a *substituting part*. The condition part must be satisfied for the continuation of evaluation of a filter element. The matching part is matched against a message to determine whether the filter element applies to the message. The substituting part replaces, if required, portions of the message.

Unlike the CFOM, the CEOM does not use separate objects to determine whether to accept or reject a message. Instead, such actions are coded in a special method of an object, termed the *replacement interface handler* (RIH). A message in the CFOM is processed sequentially by each input filter until it is dispatched for execution. Thus, the order in which the filters are specified plays a significant role in determining the message processing behavior of an object. A slight change in the order of two filters may change the behavior of an object drastically. Among the several operations which a filter can perform on a message object in the CFOM, one is to change argument values in a request message. In contrast, a CE object may only add argument values to a message object. This property of the CEOM retains the encapsulation of messages exchanged by clients and servers. Moreover, unlike the CFOM, an object in the CEOM does not affect any aspect of an outgoing message.

The CFOM uses *delegation-based inheritance* in which a subclass must declare a private instance of the superclass in order to inherit from the class. Then, using input filters, messages requesting the execution of superclass methods are delegated to this private instance object. If it is determined from the conditions in the input filters that the superclass does not support the requested method, the message is executed by the subclass object. Since input filters are evaluated in a specific order, the order of the filter elements must be explicitly manipulated by a designer of an object to properly use the inheritance mechanism. Moreover, if the instance object corresponding to the superclass must execute a method that has been redefined in the subclass, then a special *pseudo-variable* termed *server* must be used to obtain access to the redefined method. The CEOM, on the contrary, uses both a language construct and the underlying run-time mechanisms of C++ for inheritance. As a result, in CEOM, one does not have to use special pseudo-variables like *server* to access subclass methods. Instead, the *virtual* method declaration mechanism yields a transparent access mechanism to subclass methods. The design method proposed in this dissertation to develop reusable coordination specifications relies heavily on the syntactic and run-time support for inheritance. Using a *delegation-based inheritance* mechanism as the basis of such a method would have burdened a designer with dealing with low-level, message-scheduling issues.

The ACT model relies on an ACT object to abstract communication details and to enforce coordination constraints. An ACT object is like any other CFOM object except that it operates on first-class representations of messages received and sent by the coordinated objects. The act of transforming a message to this first-class representation is termed *message reification.* A message is reified by using a special filter object instantiated from the class *Meta.* Consider the class *ReferencePoint* defined below that uses an ACT object.

```
class ReferencePoint interface
   externals
     figure: OneWayConstraint;
   internals
     myPoint: Point;
   methods
     display returns Nil;
   inputfilters
     {
        constraint: Meta = True ↦[*.moveTo]figure.applyConstraint;
        disp: Dispatch = True ↦ myPoint.*, True =¿ inner.*;
```

```
    }
end;
```

The above class is a subclass of the class *Point.* Hence a private instance variable *myPoint* is declared of type *Point* in the *internals* section of the interface declaration. The instance variable *figure*, declared in the *externals* section, stores a reference to an instance of the ACT class *OneWayConstraint.* Note that a component in the ACT model stores an explicit reference to the group coordinator by its type (in this case *OneWayConstraint*). In the CEs model, a component need only store a pointer to the *CoordinatingEnvironment* class and need not know the names of abstract or concrete CE classes. That independence has the potential of increasing the reusability of components.

A component class in the ACT model declares explicit filters which forward messages received by the component to an ACT object. In the *ReferencePoint* class above, the filter object *constraint*, instantiated from class *Meta*, is used to forward a message to the ACT object stored in *figure.* The specific message that is forwarded is the one which requests the execution of the *moveTo* method defined in the *Point* class. Through message reification, the request message is passed as an argument to the *applyConstraint* method in the *OneWayConstraint* class. On the contrary, in the CEs model, posting of events by components is done transparently and no explicit specification need be included in the definition of a component.

Consider the *OneWayConstraint* class defined below from which the ACT object is instantiated.

```
class OneWayConstraint interface
    methods
        applyConstraint(Message) returns Nil;
        putDependants(OrderedCollection(Any)) returns Nil;
        size returns Integer;
        putConstraints(OrderedCollection(Block)) returns Nil;
        getConstraints returns OrderedCollection(Block));
    inputfilters
        disp: Dispatch = True ↦ inner.*;
end;
```

Unlike a CE object, the constraint enforcement action of an ACT object is explicitly initiated by executing a public method on its interface, the *applyConstraint* method in the above example. In a CE object, a component just posts an event and, depending on the behavior of the CB method that is active, an appropriate coordinating action is taken. Moreover, unlike a CE object, an ACT object does not capture the states of the group in explicit syntactic entities. A CE object captures the state of a group in a CB method which helps understand and reason about the behavior of the group.

Refining the definition of ACT classes through delegation-based inheritance has the same shortcomings as before: delegating messages to superclass methods needs explicit ordering of input filters and the execution of subclass methods from a superclass method requires explicit manipulation of a pseudo variable.

Finally, the notion of observing termination of methods in component objects is not present in the ACT model. In the CEs model, method terminations are observed because it helps a CE object to synchronize with the activities of the coordinated components and to be aware of the current states of all the components.

Chapter 7

Summary and Contributions

7.1 The Coordinating Environments Model

The existing paradigms for coordinating a group of objects which work together to achieve a common task/goal have been classified into three categories: centralized, decentralized, and hybrid. In the decentralized paradigm, characterized by its use of explicit communication, coordination issues are entangled with the inherent functionality of an object thereby complicating its design and maintenance. The centralized paradigm avoids this entanglement by severing explicit communication links among components and by managing coordination centrally. The centralized paradigm has the side-effect of hiding components of a group and projecting a consolidated view to the clients. Such an approach hides internal complexity from clients and has been successfully applied in the design of large, hierarchically structured, procedural software systems. But, such central control conflicts with the basic premise of the concurrent object-oriented programming paradigm: point-to-point, direct communication among software counterparts of real-world entities. For example, when using the buttons in a panel, one interacts directly with a button; when using a vending machine, one interacts directly with the coin acceptor and the slots; when using an elevator system, one interacts directly with an elevator car. Although hiding the complexity of internal communications must be achieved, the natural expectation of clients of how such systems project their views to the external world and how clients interact with them must not be sacrificed. The hybrid paradigm addresses the drawbacks of the decentralized and the centralized paradigms by managing coordination centrally and yet exposing components to clients. Thus, coordinators in this paradigm strike a balance between separate encapsulation of coordination and rigid, centralized control.

This dissertation proposes a new hybrid coordination model termed the *Coordinating Environments* model (CEs model). Coordination in the CEs model is enforced by *Coordinating Environment objects* (CE objects) which are instantiated from *Coordinating Environment classes* (CE classes). The objects coordinated by CE objects are termed *autonomous objects*. Autonomous objects apply selective message processing criteria before executing messages and have the capability of executing in parallel with clients. Autonomous objects do not buffer request messages that cannot be processed immediately because of local constraints. Instead, clients are made to withdraw such requests. This relieves CE objects from maintaining the causal ordering between client requests. CE objects use special methods, called *Coordinating Behavior* methods (CB methods), which implement the coordinating actions using procedural techniques.

Coordinating actions of a CE object are triggered by the occurrence of events in component objects. Event occurrences are transparently "observed" by a CE object. As a result, the designer of a CE object does not have to include code for event notifications. CE objects realize observation through a matching process which uses references to components and their methods but does not use arguments used by the methods. This feature helps maintain the encapsulation of the data exchanged in request messages by clients and servers. A CE object observes both the acceptance of a request message and the termination of a method that was scheduled by the CE object. The observation of termination is necessary for correct coordination in a concurrent, asynchronous, message-passing, object-oriented environment. On observing an event, a CE object may take one of the following actions apart from updating its local variables: schedule the accepted message for execution and either continue immediately or wait for the termination of the method, block and unblock request messages, add default argument values to the argument list in the request message, synchronously and asynchronously invoke methods in components, or specify a replacement CB method. Another action which may be taken by a CE object is to mark one or more acceptance events as unobserved. This action enables a CE object to ignore those acceptance events which do not play any role in its coordinating activities.

A method for designing reusable CE classes in C++ which utilizes the features of a special runtime system is described. The salient feature of the design method is that it defines *abstract CE classes* which do not make use of either the names of the classes from which components are instantiated or the names of the methods of components. An abstract CE class is extended using the inheritance mechanism to define *concrete CE classes.* A concrete CE class maps the coordination pattern defined by the abstract CE class into concrete coordination solutions. Using the design method, several coordination problems have been solved ranging from the coordination between a pair of buttons in a panel to the coordination of a multi-car elevator system. The level of complexity that can be handled by the primitives of the CEs model is an encouraging indication of its suitability to solve a wide variety of coordination problems. A detailed design of the classes to implement CE objects and the event objects (using which a CE object observes events) have also been provided.

7.2 The Calculus of Coordinating Environments

The Calculus of Coordinating Environments (CCE) is motivated by the desire to apply the concept of hybrid, non-intrusive group coordination to coordinate agents specified in the Calculus of Communicating Systems (CCS). Using CCS directly to specify coordination has two weaknesses. First, coordination is modeled only at a very low level by making agents engage in explicit communications. Such low-level specifications are very poor candidates for specifying designs of software components which must satisfy software engineering criteria like separation of concerns, comprehensibility, modifiability, and reusability. Also, such low-level specifications are difficult to construct and difficult to understand. Second, when the computation steps of the composition of agents are determined using the Expansion Law of CCS, many terms are generated which represent incorrect coordination sequences among the agents.

CCE extends CCS by providing a new composition combinator and by introducing a special type of agent called Coordinating Environments agents (CE agents). A CE agent coordinates the composition of multiple CCS process-agents by observing actions at specific ports of the coordinated agents and taking coordinating actions. Two new composition rules and a new Expansion Law are proposed which enable the composition of a CE agent with CCS process-agents to yield execution steps which are consistent with the coordination constraints present in a group. CCE is considered to be the first step towards a much more expressive calculus which will provide an integrated approach for specifying concurrency, communication, and coordination.

7.3 Contributions

There are two major contributions of this dissertation: a new hybrid model of coordination for coordinating concurrent objects, termed the Coordinating Environments model (CEs model), and a new formalism for specifying and reasoning about coordinating agents, termed the Calculus of Coordinating Environments (CCE).

The CEs model is an improvement over two other similar hybrid coordination models, namely, synchronizers and ACT. Compared to synchronizers, the CEs model achieves a better separation of concerns by preventing the replication and the management of component data in group coordinators. Compared to the ACT model, it is easier to reuse coordination specifications using inheritance in the CEs model. The use of a delegation-based inheritance mechanism in ACT complicates the subclassing mechanism thereby burdening the designer with low-level details unrelated to the management of coordination. Also, designer of components in the ACT model must include explicit code to implement the interaction among the components and a group coordinator. Such interactions are implicit in the CEs model. Compared to both synchronizers and ACT, the CEs model achieves a structured representation of coordinating actions based on the valid states of a group. Such a structured representation makes group coordinators easier to understand and maintain. In synchronizers coordinating actions are specified as a list of declarative condition-action rules that are difficult to understand and maintain as the complexity of the coordination problem increases. In ACT, coordinating actions are defined as operations of the coordinating agent that do not represent the valid states of a group.

The CCE allows the formal specification of non-intrusive coordinating agents called Coordinating Environment agents (CE agents). CE agents allow the formal specification of component agents that do not engage in explicit communication either with other coordinated components in a group or with the group coordinator and that do not synchronize with a group coordinator to intercept coordinating actions. CE agents are difficult to specify in the Calculus of Communicating Systems (CCS) since there is only one way of representing interaction among agents in CCS: explicit communication. CCE partitions the action of a CE agent into an observation action followed by a coordinating action. Due to the special semantics of the CE agents, the observation action occurs without engaging in explicit communication with a coordinated agent thereby relieving the coordinated agent from engaging in explicit communication. CE agents are also difficult to specify using the notion of membranes in the Chemical Abstract Machine (CHAM) formalism because the behavior of a membrane cannot be specified independently. The CCE also enables the verification of the coordination ability of a CE agent by proposing special composition rules and a new expansion law.

7.4 Future Work

The CEs model is not the panacea for the complex problems which arise in the specification and the enforcement of multi-object coordination in concurrent object-oriented languages. Rather, it is a step towards mitigating some of the known shortcomings of extant coordination paradigms and is expected to lead to better and refined coordination models. Some possible avenues of future research are discussed below.

In its current form, the CEs model does not support sharing of a component between two or more CE objects so that the shared component can post events to more than one CE object. Such sharing can be easily implemented by storing multiple CE-object pointers in a component and by informing a component which event must be posted to which CE object. But that increases the complexity of the specification and the burden on the designer of CE objects. The feasibility of such sharing must be studied by studying actual coordination problems which require component sharing and possible ways of optimizing the specification and the implementation of such a scheme must be devised.

The feasibility of hierarchical composition of CE objects must be studied. The horizontal composition of CE objects has been illustrated by the ripple-carry adder coordination problem in Chapter Three. In the solution to that problem, a CE object facilitates the horizontal composition of two other CE objects by coordinating two components, one from each of the groups coordinated by the latter CE objects. In a hierarchical (or vertical) composition, the composing CE object will store pointers to one or more composed CE objects (and, optionally, other components). In that case, the most important question which must be answered is: What events must trigger the actions of the composing CE object? There are mainly two alternatives: events in components coordinated by the composed CE objects or events in the composed CE objects. In the case of the first alternative, an event in a component must percolate through the hierarchy of CE objects and possibly be observed at each level. Although not impossible to implement, such a scheme would delay a component until each level has made a decision about the event. Also, the issue of resolving conflicting decisions made by different CE objects must be addressed. In the case of the second alternative, the primary issue is: What events in composed CE objects may be observed by the composing CE object? The only alternative appears to be: Initiation and termination of CB methods. But, since several events may be observed in a single CB method, observing the initiation and termination of a CB method amounts to observing all the events which the CB method observed. Whether such collective event observations may be useful in coordinating object groups must be investigated.

Some other possible avenues of work are as follows. First, the object model considered in this dissertation realizes asynchronous method invocation using method pointers. As a result, overloaded methods cannot be invoked using request messages. An alternative way must be devised so that method overloading is possible. Another aspect of the object model that needs further investigation is the inability of an object to send a request message to itself. According to the model, an object may not accept a request message while it is executing a method. So if an object sends a message to itself, it will not be able to decide whether to process it until the current method terminates. Yet the current method cannot terminate until the object decides whether to process the message. This circular waiting leads to deadlock. The coordination problems solved in the dissertation did not require objects to send messages to themselves. The need for such a capability must be ascertained and possible extensions to the object model must be investigated. Second, type checking of arguments supplied in request messages must be realized in order to provide a type-safe programming environment. Third, possible ways of refining the design method described in Chapter Three for defining abstract and concrete CE classes must be investigated. Fourth, the CEs model must be implemented and the performance of the proposed coordination primitives must be studied and compared to those of other coordination models. Fifth, using the experience gained in defining and realizing the CEs model, the possibility of devising a methodology for designing reusable software components must be investigated. Especially, the issue of *externalization*, the events and data which must be exposed by a component to facilitate coordination, must be carefully investigated.

With respect to the Calculus of Coordinating Environments (CCE), some possible avenues of work are as follows. First, the severity of the current restriction that coordinated agents may not communicate among themselves must be determined. Such internal communications generate a special action in CCS, termed the τ action. Since the ports involved in generating the τ action cannot be determined once the τ action occurs, a CE agent cannot distinguish between two τ actions. The inability to distinguish between τ actions affects the coordination capability of a CE agent because the CE agent cannot synchronize its coordinating actions with the occurrence of events in the components. Hence, the restriction is imposed on coordinated agents. Although none of the coordination problems solved required communication among coordinated components, the flexibility that such internal communications may introduce must be determined and and the composition rules of CCE must be appropriately extended. Second, the *pi-calculus* [26] may be considered for modeling coordination among agents whose interconnection topology is dynamic. Third, the Asynchronous CCS [32, 33] may be considered to model coordination among agents that engage in asynchronous communication using messages. Such a calculus would capture more faithfully the communication among concurrently executing autonomous objects which are the cornerstone of the concurrent object-oriented paradigm.

Bibliography

[1] R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.

[2] A. R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In OOPSLA'90, in Special Issue of SIGPLAN Notices, pages 169-180, Ottawa, 1990. ACM Press. Joint conference ECOOP/OOPSLA.

[3] I. M. Holland. Specifying Reusable Components Using Contracts. In O. Lehrmann Madsen, editor, *Proceedings ECOOP'92*, LNCS 615, pages 287-308, Utrecht, The Netherlands, July, 1992. Springer-Verlag.

[4] C. Arapis. Specifying Object Interactions. In D. Tsichritzis, editor, *Object Composition.* University of Geneva, 1991.

[5] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990.

[6] G. Agha and C. J. Callsen. ActorSpace: An Open Distributed Programming Paradigm. In *Principles and Practice of Parallel Programming Conference Proceedings,* 1993. To appear.

[7] A. Yonezawa, editor. *ABCL — an object-oriented concurrent system.* MIT Press, 1990.

[8] S. Gibbs. Composite Multimedia and Active Objects. In *OOPSLA'91 Conference Proceedings*, pages 97-112.

[9] M. H. Olsen, E. Oskiewicz, J. P. Warne. A Model for Interface Groups. In *Proceedings IEEE 10th Symposium on Reliable Distributed Systems*, pages 98-107, 1991.

[10] S. Frolund and G. Agha. A Language Framework for Multi-Object Coordination. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 346-359, Germany, July, 1993. Springer-Verlag.

[11] D. C. Luckham et al. Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems. *Journal of Systems Software.* Vol. 21, June 1993, pages 253-265.

[12] M. Bourgois, J-M Andreoli, and R. Pareschi. Concurrency and Communication: Choices in Implementing the Coordination Language LO. *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, R. Guerraoui, O. Nierstrasz, and M. Rivelli editors, LNCS 791, pages 73-92, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[13] ISA Project Core Team. ANSA: Assumptions, Principles, and Structure. In J. P. Warne, editor, *Conference Proceedings of Software Engineering Environments 1991*, University College of Wales, Aberystwyth, March, 1991.

[14] M. Aksit et al. Abstracting Object Interactions Using Composition Filters. *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, R. Guerraoui, O. Nierstrasz, and M. Rivelli editors, LNCS 791, pages 152-184, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[15] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[16] N. Carriero and D. Gelernter. Linda In Context. *Communications of the ACM*, April, 1989, Vol. 32, Number 4.

[17] D. Gelernter. Multiple tuple spaces in Linda. In E. Odjik, M. Rem, and J.-C. Syre, editors, *PARLE '89, Vol. 2*, LNCS 366, pages 20-27, June 1989. Springer-Verlag.

[18] N. Francez, B. Hailpern, and G. Taubenfeld. Script: A Communication Abstraction Mechanism and its Verification. em Science of Computer Programming, 6, 1986, pages 35 - 88, North-Holland.

[19] A. W. Holt. Diplans: A New Language for the Study and Implementation of Coordination. *ACM Transactions on Office Information Systems*, Vol. 6, Number 2, pages 109-125, April 1988.

[20] P. Ciancarini. Coordinating Rule-Based Software Processes with ESP. *ACM Transactions on Software Engineering and Methodology*, Vol. 2, Number 3, pages 203-227, July 1993.

[21] F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, February 1993, Vol 1, Number 1.

[22] A. Corradi and L. Leonardi. PO Constraints as Tools to Synchronize Active Objects. *Journal of Object-Oriented Programming,* pages 41-53, Oct. 1991.

[23] C. Atkinson, S. Goldsack, A. D. Maio, and R. Bayan. Object-Oriented Concurrency and Distribution in DRAGOON. *Journal of Object-Oriented Programming*, March/April 1991.

[24] Mario Tokoro. Computational Field Model: Toward a New Computational Model/Methodology for Open Distributed Environment. In *Proceedings of the 2nd IEEE Workshop on Future Trends in Distributed Computing Systems*, Sept. 1990, Cairo, Egypt.

[25] J. Van Den Bos and C. Laffra. PROCOL: A Concurrent Object-Oriented Language with Protocols, Delegation, and Constraints. *Acta Informatica*, Vol. 28, Number 6, 1991.

[26] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. Research Report. Laboratory for Foundations of Computer Science, University of Edinburgh, June 1989.

[27] Oscar Nierstrasz. Towards an Object Calculus. *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing,* M. Tokoro, O. Nierstrasz, P. Wegner, A. Yonezawa editors, LNCS 612, Springer-Verlag, Geneva, Switzerland, July 15-16, 1991.

[28] G. Berry and G. Boudol. The Chemical Abstract Machine. *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages,* San Francisco, California, January 17-19, 1990.

[29] J-P Banatre and D. L. Metayer. A New Computational Model and Its Discipline of Programming. Technical Report INRIA Report 566, 1986.

[30] K. G. Larsen. *Context-Dependent Bisimulation Between Processes.* Doctoral Dissertation. University of Edinburgh, 1986.

[31] M. Mukherji. *The Implementation of ACT++ on a Shared Memory Multiprocessor.* M.S. Project Report. Department of Computer Science, Virginia Tech, Feb. 1992.

[32] K. Honda and M. Tokoro. On Asynchronous Communication Semantics. In M. Tokoro, O. Nierstrasz, and P. Wener, editors, *Proceedings of ECOOP '91 Workshop on Object-Based Concurrent Computing*, pages 21-51, Geneva, Switzerland, July 1991. Springer-Verlag.

[33] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 133-147, Geneva, Switzerland, July 1991. Springer-Verlag.

Additional Readings

[34] D. Kafura, M. Mukherji, and G. Lavender. ACT++: A Class Library for Concurrent Programming in C++ using Actors. *Journal of Object-Oriented Programming*, October, 1993.

[35] M. Papathomas. A Unifying Framework for Process Calculus Semantics of Concurrent Object-based Languages. In M. Tokoro, O. Nierstrasz, and P. Wener, editors, *Proceedings of ECOOP '91 Workshop on Object-Based Concurrent Computing*, pages 53-79, Geneva, Switzerland, July 1991. Springer-Verlag.

[36] S. Matsuoka, K. Wakita, and A. Yonezawa. Analysis of Inheritance Anomaly in Concurrent Object-Oriented Languages. *ECOOP/OOPSLA'90 Workshop on Object-Based Concurrent Systems*, August 1990.

[37] C. Tomlinson and V. Singh. Inheritance and Synchronization with Enabled Sets. In *OOPSLA'89 Conference Proceedings,* pages 103-112, October 1989.

[38] G. Lavender and D. Kafura. Specifying and Inheriting Concurrent Behavior in an Actor-Based Object-Oriented Language. Technical Report TR 90-56. Virginia Tech, 1990.

[39] C. Neusius. Synchronizing Actions. In *ECOOP'91 European Conference on Object-Oriented Programming.* Springer-Verlag, 1991.

[40] S. Frolund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In O. Lehrmann Madsen, editor, *Proceedings ECOOP'92*, LNCS 615, pages 185-196, Utrecht, The Netherlands, July, 1992. Springer-Verlag.

[41] M. Aksit and L. Bergmans. Obstacles in Object-Oriented Software Development. In *Proceedings OOPSLA '92.* ACM, Oct. 1992.

[42] B. N. Freeman-Benson and A. Borning. Integrating Constraints with an Object-Oriented Language. In O. Lehrmann Madsen, editor, *Proceedings ECOOP'92*, LNCS 615, pages 268-286, Utrecht, The Netherlands, July, 1992. Springer-Verlag.

[43] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. In *Conference Record of the 9th Annual ACM Symposium on POPL,* pages 7-19, Albuquerque, New Mexico, Jan. 1982. ACM.

[44] O. Nierstrasz. Active Objects in Hybrid. In *Proceedings OOPSLA '87,* pages 243-253, Dec. 1987. Published as ACM SIGPLAN Notices, Vol. 22, No. 12.

[45] S. C. Reghizzi, G. G. de Paratesi, and S. Genolini. Definition of Reusable Concurrent Software Components. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 148-166, Geneva, Switzerland, July 1991. Springer-Verlag.

[46] M. Karaorman and J. Bruno. Introducing Concurrency to a Sequential Language. *Communications of the ACM,* pages 103-116. Sept. 1993, Vol.36, No. 9.

[47] M. Muhlhauser, W. Gerteis, and L. Heuser. DOCASE: A Methodic Approach To Distributed Programming. *Communications of the ACM,* pages 127-138. Sept. 1993, Vol.36, No. 9.

[48] A. Yonezawa and M. Tokoro (Editors). *Object-Oriented Concurrent Programming.* Computer Systems Series. MIT Press, 1987.

[49] P. America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In *ECOOP '87 Conference Proceedings,* pages 234-242. Springer-Verlag, 1987.

[50] E. P. Andersen and T. Reenskaug. System Design by Composing Structures of Interacting Objects. In O. Lehrmann Madsen, editor, *Proceed-*

ings ECOOP'92, LNCS 615, pages 133-152, Utrecht, The Netherlands, July, 1992. Springer-Verlag.

[51] K. M. Chandy and C. Kesselman. Compositional C++: Compositional Parallel Programming. California Institute of Technology.

[52] B. Thomsen. A Calculus of Higher Order Communicating Systems. In *Conference Record of the 16th Annual Symposium on POPL*, pages 143-154, Austin, Texas, Jan. 1989. ACM.

[53] A. Obaid and L. Logrippo. An Atomic Calculus of Communicating Systems. In H. Rudin and C. H. West, editors, *Proceedings of the IFIP 7th international Conference on Protocol Specification, testing, and Verification*, pages 91-104, Zurich, Switzerland, May, 1987. Elsevier Science Publishers.

[54] R. Milner. Functions as Processes. In *Proceedings of the 17th International Colloquium on Automata, Languages, and Programming*, pages 167-180, Warwick University, England, July, 1990. LNCS 443. Springer-Verlag.

[55] K. V. S. Prasad. *Combinators and Bisimulation Proofs for Restartable Systems.* Doctoral Dissertation. University of Edinburgh, 1987.

[56] D. Walker. Pi-Calculus Semantics of Object-Oriented Programming Languages. Research Report, University of Technology, Sydney. Sept. 1990.

[57] K. G. Larsen. *Context-Dependent Bisimulation Between Processes.* Doctoral Dissertation. University of Edinburgh, 1986.

[58] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or How to use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, pages 9-18, Vol. 18, No. 1, Jan. 1992.

[59] A. Corradi and L. Leonardi. Concurrency Within Objects: Layered Approach. *Information and Software Technology,* pages 403-411, Vol. 33, No. 6, July/August 1991.

[60] F. Hayes and D. Coleman. Coherent Models for Object-Oriented Analysis. In *Proceedings OOPSLA '91*, pages 171-183, SIGPLAN Notices, Vol. 26, N0. 11, Nov. 1991.

[61] N. Carriero and D. Gelernter. Coordination Languages and their Significance. Technical Report YALEU/DCS/RR-716, Yale University, July, 1989.

Vita

Manibrata Mukherji was born on June 18, 1963 in Calcutta, India. He obtained the Bachelor of Computer Science and Engineering degree from Jadavpur University, Calcutta, India, in June 1986. He obtained the Master of Science degree and completed a Doctorate in Computer Science and Applications from Virginia Tech, Blacksburg, Virginia, in February 1992 and July 1995, respectively.

[illegible]