Concurrent Object-Oriented Languages and the Inheritance Anomaly

Dennis G. Kafura[1] and R. Greg Lavender[2]

[1] Dept. of Computer Science, Virginia Tech, Blacksburg VA 24061, USA
[2] MCC, 3500 W. Balcones Center Dr., Austin TX 78759, USA

Abstract. A survey of concurrent object-oriented languages is presented. The survey is organized around three models: the Animation Model that describes a variety of relationships between threads and objects, an Interaction Model that classifies the possible semantics of invocations and returns between a client object and a server object, and a Synchronization Model that shows different ways in which concurrent invocations can be managed by a server. A number of representative languages are briefly presented. The problem of synchronization in concurrent object-oriented languages is considered in detail including a discussion of the *inheritance anomaly*. A synchronization mechanism, called a *behavior set*, is shown to avoid this anomaly in certain cases. The implementation of behavior sets in ACT++, an actor-based concurrent programming framework implemented in C++, is described.

1 Introduction

This survey of concurrent object-oriented programming languages is organized along the following lines:

1. a review of historical and technical factors motivating the synthesis of concurrency and object-orientation.
2. a description of the most significant problems confronted by designers of concurrent object-oriented languages, an overview of the solutions to these problems and a summary of what appear to be commonly accepted solutions.
3. a brief overview of each of several languages illustrating attempts to integrate the individual design solutions into a coherent language.
4. an in-depth presentation of the inheritance anomaly, and a solution to this problem, called behavior sets, which has been developed as part of the ACT++ project.

The remainder of this introductory section consider the first point. Subsequent sections consider each of the other points in order.

This paper relies heavily on a previous survey by Wegner [29] and a taxonomy presented by Papathomas [23]. While differing in terminology and in some conclusions, the thorough work of Papathomas was an especially important information source for the survey undertaken here.

As a final preparatory comment, the terms "language", "object-oriented" and "parallel" are given generous interpretations in this survey. The term language is taken to encompass:

Fig. 1. Evolution of concurrent object-oriented programming

The most basic memory abstraction is that of a storage location (machine byte, word or register). This is the only abstraction in low-level assembly languages. The earliest high-level languages introduced a fixed set of built-in types, providing greater ease in organizing an application's data and greater safety in manipulating this data. The provision of user-defined types extended to the programmer the ability to define application specific types that had the same standing as the previous built-in types. The development of abstract data types recognized the importance of the separation of the type's interface from its possibly many differing implementations. Object-oriented programming adds to abstract data types the important ability to incrementally extend an existing type.

Abstractions for a processor can also be traced by considering how programming languages have treated the notion of control. The primitive manipulation of control, by directly and explicitly affecting machine registers, was abstracted in higher level languages through the statement structure, some of which implicitly managed control (expressions) and some of which explicitly managed control (branches, loops). The invention of the closed subroutine introduced a major control abstraction. Each subroutine could be envisioned as an extension of the instruction set of the physical processor. Advances in architecture and operating systems encouraged application developers and language designers to depart from the sequential model resulting in languages with a process concept. Coroutines, parallel tasks and their associated coordination devices (e.g., monitors) provided an abstraction of a complete machine. More recent operating system research has reduced the previously high cost of multiprocess applications through the invention of lightweight processes or threads.

As shown in Fig. 1, concurrent object-oriented languages unify the previously separated abstractions for processor and memory. It is interesting to note that current research is following the pattern of natural evolution. A large number of experimental concurrent object-oriented languages are being developed; the technical and market forces that shape the environment of computing will select only a small number of survivors from among this rich variety.

As opposed to the historical factors just considered, the technical factors motivating concurrent object-oriented languages include at least the following four. First, conceptual economy is achieved by unifying the processor and memory abstractions. It is not necessary for the application developer to consider how to organize the application's control separately from the question of how to organize the application's objects. This is an especially important advantage in that it avoids presenting the developer with two semantic models—one model for control and a different model for data and functions. Second, modeling fidelity is enhanced in those applications where autonomous, real-world entities are pervasive. In this case the attributes of the real-world entity are more completely, overtly and simply reflected in concurrent objects than would be the case if the autonomy and the functionality of the object were realized disjointly. Third, simplicity is attained for the two reasons already present but also for two other reasons. An important goal in language design is to focus attention on important abstractions while suppressing from attention aspects that are irrelevant to this purpose. A concurrent object-oriented language suppresses substantial detail about control flow management, specifically the invocation and scheduling of object executions. By contract, sequential languages force the developer to explicitly construct a, perhaps elaborate, invocation and scheduling

Fig. 2. The Three Models

The selection of the three models is not intended to imply that this is the only,

Fig. 3. The Animation Model

The most global decision about animation is the extent to which the threads respect the boundary of an object. The first of two alternatives in the Animation Model may be termed the unrelated case. In the unrelated case, threads and objects are treated as independent concepts. An object is defined and created without regard for how any thread may animate that object. Similarly, a thread is free to cross an object's boundary and to have at a single point in time an execution history in which any number of object boundaries have been crossed. The unrelated approach has been used in the earliest concurrent object-oriented languages and in later low-level, general purpose, run-time frameworks.

The unrelated approach has several advantages. First, it is easy to develop such a language—combining any threads facility with an object-oriented language will suffice. Few requirements are imposed on either the language or the existing threads package. Second, this approach is attractive in application domains that are strongly "activity centered" as opposed to "object centered". The activity, being directly associated with the thread, has an explicit representation that captures the activity's properties and maintains the activity's state.

The disadvantages of the unrelated approach are more fundamental than its advantages. First, the presence of multiple threads within a single object recreates all of the synchronization problems that have been the bane of concurrent programming from its inception. Second, the modeling advantages offered by object-oriented programming are eroded by the unconstrained license given to threads in this approach. Part of the erosion results from the extraneous attributes added to objects to cope with thread synchronization. Additional erosion results if the threads are merely an elaborate control structure superimposed on the object structure. In this case a significant aspect of the application cannot be addressed within an object-oriented framework. Third, and most significantly, the unrelated approach fails to advance our understanding of either the object-oriented paradigm or concurrency; each aspect exists unchanged and, as implied by the name, unrelated to the other.

The second alternative in the Animation Model, the related approach, involves a deeper integration of concurrency and object-orientation. In this approach a thread exists only within an object; the thread's locus of control may not migrate beyond the encapsulating boundary of this object. As a thread's scope of effect is now limited, it is necessary to consider how a client thread in one object interacts with the thread in a server object. The nature of this interaction is significant and is considered in the Interaction Model presented below.

The related approach allows language and system researchers to probe, and application developers to benefit from, the answers to two key questions:

1. Can concurrency be transparently encapsulated within an object?
2. Can inheritance be used to organize and specialize synchronization policies in the same way that inheritance is used to organize and specialize an object's functionality?

The answer to the first question is remarkably straightforward. The second question is remarkably subtle and a complete answer remains elusive. Further consideration of the second question is deferred until Sect. 4 where a partial answer is presented. Only the first of these questions in considered further at this point.

Mapping threads to objects in the related approach involves two independent dimensions: uniformity and multiplicity. Uniformity refers to whether the object model invests every object with a thread (the homogeneous case) or whether some objects possess threads while other do not (the heterogeneous case). An object possessing a thread is often termed an *active* objects while a *passive* object is used to describe an object without a separate thread. Multiplicity refers to the number of threads that exist within an active object. While all concurrent object-oriented languages provide some form of concurrency among objects (inter-object concurrency), allowing more than one thread within an object creates, in addition, a finer level of concurrency (intra-object concurrency). The various forms of multiplicity are considered further below.

The first choice on the uniformity dimension, the homogeneous approach, offers a simple view of the relationship between threads and objects; only a single protocol of interaction among objects needs to be understood, implemented, optimized and employed. Papathomas [23] argues that homogeneous objects are more easily reused as all objects inherently possess the ability to be used in a concurrent environment without any synchronizing actions being required of the client. In contrast, the clients must themselves provide the synchronization required to access passive objects. Finally, in some languages the syntaxes or mechanisms for accessing active versus passive objects are different. Changing an object definition from one form to the other requires not only internal changes to the object itself but also requires changes to the clients of the object. This is generally viewed as contrary to the philosophy of object-oriented programming where changes in the implementation of the server should not affect the client.

The second choice on the uniformity dimension, the heterogeneous approach, is argued for on the basis of both modeling and implementation. From a modeling perspective, some objects are more usefully viewed as being passive in nature while others are active. For example in designing an application for a library simulation, the librarians, patrons and the wall clock might be taken to be active entities while the books, tables and chairs might be considered passive. While such decisions vary with changes in the modeling perspective, each perspective offers some important distinction between entities that can engage in activity and entities that are the subjects of this activity. Preserving this important modeling distinction in the object's implementation assists in traceability and validation. From an implementation viewpoint, having a thread in each object may not be feasible if there are a relatively large number of objects and the run-time environment does not efficiently support lightweight threads. The performance of the application in this case will not be acceptable.

A middle ground on the uniformity dimension is taken by languages that permit passive objects but which require that they be encapsulated within the implementation of an active object. Thus, homogeneity is preserved at the outer (active object) level while heterogeneity is permitted within an active object.

The final factor in the Animation Model is that of multiplicity. This factor determines the number of concurrent threads that can executed within an object at the same time. The first of the four alternatives is the sequential case in which an active object always has a single thread of control. Such objects are simple to reason about as there is no concurrency within the object boundary. Languages that take this approach divided the world into the outer, highly concurrent part (where concurrency is achieved by invocations among autonomous, active objects) and the inner, sequential environment within each active object. While programming within a sequential object is simpler, performance gain are jeopardized if the object size is overly large so that opportunities for parallel execution are unavailable. For example, encapsulating a large matrix in a single object makes parallel operations on this matrix impossible. The next choice for multiplicity is a monitor-like (also called quasi-concurrent) form. In this case, an object may have several logical threads executing within its boundaries, but only one of these may be active at a time. Primitive constructs (such as sleep/wakeup in monitors) or automatic mechanisms are used to determine which of the possible several schedulable threads will execute next. Reasoning about the internal operation of monitor-like objects is somewhat more difficult than reasoning about sequential ones, but there is added flexibility as well. For example, a server object that accepts requests from clients, delegates the requests to worker tasks, and monitors the reply to the client is difficult or impossible to implement using a sequential object while it is straightforward to construct such a server using the monitor-like approach. The third form of multiplicity is called the actor form. In this case, an object may have multiple threads executing within it subject to the requirement that only the most recently created one of them has the ability to change the "state" of the object. The other threads may interrogate the state of the object and may alter their own local execution state. This notion follows directly from the relationship among the several behaviors of a given actor that operate exactly as described. The fourth, and last, form of multiplicity is concurrent. This form allows multiple threads to execute freely within an object. The weakness of this approach is that additional synchronization needs to be added in order to insure that the state of the object remains consistent in the face of concurrent alterations. The advantage of this approach is the possible performance enhancements that can be achieved by having multiple threads operating concurrently on large amount of data encapsulation within an object.

2.2 The Interaction Model

The Interaction Model is concerned with the semantics of the interaction between a client object and a server object. The two aspects of this relationship are:

1. What are the semantics of the invocation as viewed by the client?
2. How is the result computed by the server returned to the client?

If not evident, the discussion below will show that these two questions are intimately related. It will also be seen that concurrent object-oriented language perspective contributes little to the first questions but provides a novel perspective on the second question. The elements of the Interaction Model are shown in Fig. 4.

An interaction between a client object and a server object is initiated by the client's invocation that may be either synchronous or asynchronous. These forms of interaction have been extensively studied in the operating system and distributed computing domains; their application in the context of concurrent object-oriented languages is unsurprising.

Synchronous and asynchronous invocations different in their effect on the client. A synchronous invocation causes the client to block from the time at which the invocation is made until the time at which the server's reply is returned. By contrast an asynchronous invocation allows the client to proceed concurrently with both the delivery of the request to the server, the server's computation that generates the reply and the delivery of the reply.

Neither form of interaction can claim to be inherently superior. Neither is more "powerful" as each can be used to implement the other and each possesses disadvantages absent in the other. Two disadvantages of a synchronous invocation are:

1. the client is unnecessarily blocked in those cases where the client simply informs the server of some event and expects no reply. This is a particular problem if the server is in the condition described in the second disadvantage.
2. great care must be taken in designing a server so that, in servicing one request, a synchronous invocation made by the server itself does not cause the server to

Fig. 4. The Interaction Model

become blocked for such a period of time that the server cannot expeditiously serve other clients' requests.

The asynchronous form of invocation is free from these problems. Two disadvantages of an asynchronous invocation are:

1. lacking a clear structural tie to the client, it is more difficult for the server to return the result to the client. Some additional responsibility must be borne by the language or the programmer to return the proper result. The lack of a clear tie to the client also makes exception handling more difficult. The worst case is when a server experiences an exception in servicing the request of a client that has already terminated because it did not expect a reply.
2. run-time buffering of requests (e.g., in mailboxes or ports) is required to support asynchronous invocations. This additional expense is not justified if the application requires only, or at least predominately, synchronous forms of interaction.

These disadvantages do not occur in the synchronous case.

The choice between synchronous and asynchronous invocation semantics is properly based on the needs of the application and for this purpose two kinds of applications can be distinguished: transactive and reactive. A transactive system is one in which the dominant issue is the sequence of actions performed across multiple objects; the key design and implementation decisions depend from this issue. For transactive systems a synchronous invocation is preferred because the progress of each transaction is represented directly by the sequence of synchronous invocations leading to the current locus of control. In reactive systems, however, the dominant issue is how each agent maintains its integrity in the face of requests over which it has little, if any, direct control. Reactive systems prefer an asynchronous semantics as it more directly reflect the autonomous nature of the agents comprising the application.

This discussion of the invocation semantics is concluded by noting two interesting variations. First, a proxy object may be interposed between the client and the real server for whom the proxy is only a representative. Proxy objects are found in distributed system when the client and the proxy are located on the same node and the real server is remotely located. In this case the invocation semantics between the client and the proxy may be synchronous while the invocation semantics between the proxy and the real server may be asynchronous. Second, a broadcast style of asynchronous message passing may be used instead of the point-to-point form more commonly associated with method invocation. Broadcasting may also be matched with a form of "coordinated termination" to insure that all messages have been received before the client resumes.

The second major aspect of the Interaction Model concerns how the server's result is returned to the client. The alternatives are shown in Fig. 4 above. The return semantics may be either implicit or explicit indicating whether the identity of the object to receive the result is implicitly or explicitly known to the server.

The implicit form of return implies that the knowledge of where to deliver the reply is carried by the structure of the invocation and cannot be altered by the application. The most obvious case of an implicit form is the return semantics commonly used for synchronous invocations. The invocation environment (typically a stack of activation records) implicitly contains the identify of the client to whom the result is returned. When the server returns, the environment is adjusted to deliver the result to the invoking client. The implicit form may also be used for asynchronous invocations. A message, for example, may automatically be stamped with the identity of the sender so that an implicit return can be performed by the server.

The explicit form of return allows greater flexibility because the application is permitted to control the destination to which the server's result will be delivered. The value of this flexibility is seen in the common Coordinator-Worker model. In this model a single Coordinator receives the clients' requests and forwards each request to an available Worker. Upon receiving the forwarded request, a Worker computes the required reply value and transmits this value directly to the originating client. The Worker informs the Coordinator that it is ready to service another request. Notice that the direct transmission of the result from the Worker to the client did not involve the Coordinator. This is not possible with implicit returns as the result would have to be passed from the Worker to the Coordinator and then from the Coordinator to the client. The explicit form of return allows for the direct transmission of this value as the Worker is aware explicitly of the identity of the originating client.

Explicit returns may take one of two forms. In one case, the by-value case, the mechanism is a conduit through which the returned values is delivered. The server may not know what object is awaiting the value at the receiving end of the conduit. Variations of by-value mechanisms give some means of explicitly manipulating their end of the conduit to either the client alone, the server alone, or possibly both. For example, the server may be able to pass its end of the conduit to another server as would be desired in programming the Coordinator in the Coordinator/Worker problem. Alternatively, the client may have some ability to pass its end of the conduit

Fig. 5. Future and Cbox usage

As shown in part (a) of Fig. 5, the future variable is created in conjunction with an invocation. The client proceeds with its execution until an evaluation of the future is attempted. The client will block only under the condition that the server has not already returned, thus supplying the value to the future. It should be observed that the future variable is not visible to the server. The case of Cboxes, shown in part (b) of Fig. 5, differs from futures in two ways. First, the Cbox is visible to the server as an explicit argument and, second, more than one Cbox can be associated with an invocation. Also, part (c) of Fig. 5, shows that the immediate server (function `g`) may delegate all or a portion of its Cboxes to other servers. In the example shown in part (c), function `g` delegates to function `h` the responsibility for providing a value for the Cbox `v2`.

Additional factors related to futures are whether the future is first class and whether it is typed. To be first class means that a future variable may be passed as an argument without forcing its evaluation. The undesired evaluation of a future can occur because the formal parameter is expecting a realized value (e.g., an integer) and not a potential value (e.g., a future integer). The forced evaluation of the future, unfortunately, limits the usability of the future because its scope of effect is limited to the method in which it is created. The other factor related to futures is whether the future is typed, that is whether the type of the potential value is included as part of the type of the future itself. For example, a future that will provide an integer value may be typed as a "future" integer or more weakly as an un-typed future.

The discussion of the Interaction Model is concluded by mentioning two other related issues. The first issue, naming, is straightforward. The question here is how the client and the server are identified to each other for the purpose of an invocation. The three alternatives are:

symmetric as in CSP where the client and the server must each name the other,
asymmetric the client must name the server but the server need not name the client, or
anonymous neither the client nor the server name each other explicitly.

The most common usage is asymmetric. The second issue is whether the language is statically typed. Type conformance in object-oriented languages is a deep issue. As concurrency adds little to this issue, it is simply noted here as a factor without further comment.

2.3 The Synchronization Model

The third element of the object model. the Synchronization Model, is the most intricate and the most interesting of the three models considered in this review. The framework of the Synchronization Model, shown in Fig. 6, closely parallels that of Papathomas [24], though some terminology and organization has been changed.

The Synchronization Model considers what, if any, controls are imposed over concurrent invocations impinging on a given object. Invocation control is considered a synchronization issue because it involves the management of concurrent operations so as to preserve some semantic property of the object being acted upon. As with other synchronization problems, those invocations that are attempted at inappropriate times are subject to delay. Typically, the internal "state" of the receiving object dictates which subset of invocations is permitted to occur at a given time. Postponed invocations are those that would invalidate desired properties of the internal state. The need for this form of synchronization should be clear: guarantees

Fig. 6. The Synchronization Model

of many semantic properties cannot be made in the absence of guarantees about the internal state of the object. Notice that only inter-object concurrency is involved; intra-object concurrency, if it is permitted, must be synchronized by some other means that are not discussed further in this paper.

The two basic alternatives in the Synchronization Model are between an unconditional interface and a conditional interface. An unconditional interface implies that invocations are free to occur without regard for the condition of the receiving object. Notice that the application developer cannot prevent an invocation from occurring, the developer can only attempt to control the progress of the invocation already begun. For this purpose, a locking mechanism or condition variables are usually provided. A monitor is an example of an unconditional interface. The unconditional interface approach inevitably extracts a price by increasing the internal complexity of the object. Furthermore, the code added to achieve the synchronization runs afoul of the inheritance anomaly (discussed in Sect. 4), which limits the use of inheritance to specialize the synchronization control. A conditional interface is one in which an invocation is subject to postponement until its execution is compatible with the state of the object. In this case the object boundary is exploited as a synchronizing barrier. Synchronization is achieved by manipulating the object's boundary. The interesting question, of course, is how such manipulation is carried out.

Two different approaches to creating a conditional interface can be distinguished: concrete and abstract. Each of these approaches creates a connection between a pending invocation and the internal state of the object. These approaches differ in how the state of the object is determined, the variety of information available about the object state and the invocation, and the mechanisms employed to achieve the synchronization.

In the concrete case the internal state of the object is determined by direct interrogation of the object's state variables. Tests made on the internal state determine which invocations to permit. The two identifiable forms of concrete state testing are: guarded accepts and guarded ports.

In the guarded accept approach a monolithic, multi-way, conditional, non-deterministic, fair selection is used. Examples of this approach are the guarded select in CSP and the select construct in Ada. Such constructs are monolithic because they are an integral whole—all of the alternatives are required to be expressed at the same place in the object. The construct is multi-way and conditional meaning that any number of alternatives may be expressed and each alternative may present a condition whose truth value determines whether this alternative is available for selection. The conditions are propositions over the object's state variables. The selection is non-deterministic to deal with the case where more than one alternative is possible at a given time. Finally, the construct is usually defined to be "fair" in its selections so that a pending and selectable invocation is eventually allowed to proceed.

The subtle disadvantage of a guarded accept, implied by its monolithic nature, is that it interferes with the inheritance mechanism in two ways. To illustrate the first form of interference, consider an object definition that is based on a guarded accept construct. Now consider the problem of creating a new object definition from the existing one by inheritance. Since the guarded accept is monolithic, the subclass must provide its own guarded accept for all methods, both its own new or redefined methods in additional all of the inherited methods. Such necessity is contrary to the intent of inheritance to provide an incremental definition. It is precisely this incremental definition of the concurrency control that is lost by the guarded accept. The second form of interference is a weakening of encapsulation. Since the inheriting definition must completely reestablish the guarded accept, it must have extensive knowledge of the existing definition's state variables in order to properly form the guarded accept's conditions.

In the guarded port approach an invocation is placed in a port associated with the method being invoked. Each such port may be opened or closed depending on tests made of the object's state variables. The statements to open or close any port may be placed in any method of the object. For example, consider a bounded buffer object with methods named `put` and `get`, which are associated with ports of the same name. At creation only the object's `put` port is opened. If the buffer contains at least one element and is not full both the `put` and `get` ports would be opened. Finally, if the buffer becomes full, the `put` port is closed.

The guarded ports approach is more flexible than the guarded accept approach because it is not monolithic. However, this does not completely remove the defects in inheritability and encapsulation cited earlier for guarded accepts. These defects persist because the code to manipulate the ports is intimately interwoven with the code to perform the manipulation of the object's state. There is no clear separation between these two distinct, but related, aspects of the object. As with the example of guarded accepts, an inheriting definition might need to have extensive knowledge of the existing definition's state variables and port names in order to add or redefine a method. Such knowledge would be required because the new or redefined method's actions of opening or closing ports has an obvious affect on the other inherited methods.

The second major kind of conditional interfaces is termed abstract to reflect the fact that some abstraction exists to separate the synchronization aspects of the object from its functional aspects. The separation thus achieved overcomes the difficulties seen above with inheritability and encapsulation. In one approach, termed "behavioral", the abstraction is based on a description of the object's behavior expressed as a subset of currently available methods. In the other approach, termed reflective, an abstraction of the object's underlying execution mechanism is exposed to explicit manipulation.

In the behavioral case an object is seen as projecting a time-varying interface to its potential clients. At different times, a client observes different interfaces being projected by the server depending on the server's current condition. The projected interface changes in accordance with changes in the internal state of the object. The internal state of the object and its projected interface are linked through state-testing predicates that are programmed or are available implicitly. Synchronization is achieved by an enforcement mechanism that guarantees that an invocation is suspended until the projected interface contains the method named in the invocation. Thus, no invocation can occur when the server is in a state that is incompatible with the invocation. The term "behavioral" is used because this view of an object is similar to the notion of observable behavior used in CCS [20] and other similar theories of concurrency. Each projected interface is a subset of the maximal statically defined interface of the server. In this way the usual static type-checking rules can be applied.

In the reflective approach, the execution of an object is open to inspection and alteration by a "meta-object". In the general case, the meta-object is itself an object that can be inspected and altered by a meta-meta-object, and so on. This general reflective mechanism can be used for object synchronization by placing the synchronizing aspect of an object in that object's meta-object. For example, whenever the object is the target of an invocation, the meta-object inspects the state of the receiving object and alters the execution of the object as necessary. If the meta-object's inspection indicates that the object is in an inappropriate state for the invocation to occur, the invocation may be preserved for later execution. When the object finishes the execution of a method, the meta-object again intervenes and determines if there are any postponed invocations that are now compatible with the state of the object. Since the meta-object has access to the structure of the object, the meta-object can perform the state testing necessary to determine whether an invocation is compatible with the object's state.

3 A Survey of Several Languages

In this section a variety of concurrent object-oriented languages are briefly described. The description of each language is related to the element of the Object Model presented in the previous section. In most cases, example programs are shown to illustrate significant language features. The authors readily admit that this set of example languages is limited in two ways. First, space limitation preclude the inclusion of all concurrent object-oriented languages. Languages have been included that are well known, known by the authors, or possess an interesting set of features or an interesting approach. Second, each language is only briefly described and justice is certainly not done to the language as a whole, particularly for rich or complex languages. This survey should be viewed as only highlighting certain features of each language.

3.1 ACT++

While the ACT++ framework for concurrent object-oriented programming and, in particular the concurrency control mechanism used in ACT++, will be presented in the next section, a brief description is given here for comparison purposes.

ACT++ is a class hierarchy developed in C++. This hierarchy provides the programmer with abstractions for concurrent programming based on the actor model of computation [1]. An application in the actor model is comprised of a number of actors, autonomous agents that execute concurrently and reactively in response to the arrival of messages. Message passing is one-way and asynchronous. An actor has a mail queue to hold messages that have been received but not yet processed. An actor has a distinguished current behavior that processes one of the queued mail messages. At some point in its execution the current behavior establishes a "replacement" behavior to processes another message from its queue. The replacement behavior is established by a "become" operation. It is possible to use the become operation so that the current and the replacement behavior execute concurrently, by executing the become operation before the current behavior finishes. Alternatively, by placing the become operation at the end of the execution of the current behavior, a serial execution is enforced. In addition to the become operation, an actor may send messages and create new actors. The ACT++ framework provides classes for the basic elements of this model: actor, behavior, message and their associated operations.

In terms of the language taxonomy presented in the previous section, ACT++ uses a related approach in the Animation Model, threads are only permitted to execute within the boundary of an actor. ACT++ was intended to be used to define homogeneous objects. However, there are two ways in which heterogeneous objects may be used. First, a behavior may internally implement some number of passive objects. Second, since ACT++ is based on C++ it is possible to define passive objects that are passed among actors or that are operated on concurrently by several actors. It should be noted, however, that this second case falls outside of the scope of the intended model of computation. The multiplicity employed by ACT++ is the actor form of concurrency. In terms of the Interaction Model, ACT++ uses asynchronous invocations and results are conveyed explicitly though Cboxes. Finally, in the Synchronization Model, ACT++ uses a behavioral form of an abstract, conditional interface through a mechanism called Behavior Sets.

While a more complete example is given in Sect. 4, a brief example of bounded buffer behavior is given in Fig. 7. In this example, only the skeleton of the synchronization code is shown. The Bounded Buffer class contains three behavior sets initialized by the constructor to contain (the address of) the `in` method, the `out` method and both of these methods. Two state testing functions named `empty` and `full` are defined to determine the buffer's condition. The key aspect of the synchronization technique is the `nextBehavior` method. The `nextBehavior` method uses the state testing functions to select among the three defined behavior sets. Using the `become` method (defined in the class `Actor`), the selected behavior set is used to control the selection of the next message accepted by the actor, namely the next

Fig. 7. A bounded buffer in ACT++

3.2 ABCL

ABCL is the name given to a family of languages. Only the features of ABCL/1 [30] are described here. Like ACT++, ABCL/1 is an actor-based language. All objects are homogeneous and each object may be described as serialized, meaning that its execution is sequential, or unserialized, meaning that it may be executed concurrently without restriction. In this way, ABCL/1 supports two forms of multiplicity both of which may be used in a single program. The underlying Interaction Model of ABCL/1 is asynchronous message passing with an explicit form of return. However, syntactic structuring allows a variety of others forms to be used. For example, while the basic form of interaction is asynchronous, the syntax of the language allows the programmer to discriminate three variations of invocations: past type, which is strictly asynchronous; now type, which is a synchronous send; future type, in which a future variable is passed as part of the invocation. Two variations of the explicit return are supported in the language. The first form is of the future variety. In this case the sender specifies a future variables as part of the invocation and this is pattern matched in the server as a vehicle for delivering the reply. The second form is of the destination variety. Thus, an invocation message bears a reply point indicating to what object the result should be delivered. The reply point is available to the server object as a pattern-matched variable. The language allows the programmer the flexibility of using an implicit form of return, in which the default reply destination (the reply destination specified by the sender) is used. The Synchronization Model used in the original ABCL/1 language was a guarded accept form. A select statement allowed queued messages to be pattern matched against the alternatives of the select. One of the messages satisfying a pattern matching would be selected for execution. It should also be noted that a later variant, named ABCL/R [28], use a reflective form of synchronization control. Two other minor observations about ABCL are the following. First, inheritance is not supported in the base language. Instead of form of task delegation is used, Second, messages may be sent in either normal mode or express mode. Express mode messages provide a means of interrupting the normal processing of a message by a server. This useful facility is lacking in most other concurrent object-oriented languages. An example of ABCL/1 is shown in Fig. 8. In this example a three element buffer is created and its use by a producer and consumer are illustrated. Note that an object consists of a state part (private data of the object) and a script part (the methods or operations defined by the object). In the put method the use of the select construct can be seen. When the buffer is full an arriving put method cannot be executed. Thus, the select construct is executed to await the arrival of a get method that will create empty space in the buffer. After such a get method has been executed the put method can be resumed to store the provided value in the buffer.

3.3 Hybrid

The Hybrid language and run-time system [21] employs homogeneous active objects that use a monitor-style form of multiplicity. Recall that this allows an object to have at most one active thread at a time from among several threads that may execute within the object at different points in time. A mechanism called a domain provides additional structuring of objects and threads. A domain is viewed as a "top-level" object that may encapsulate several lower-level objects. It is the domain that is subject to the monitor-style control of thread execution. By implication, this restriction extends to all of the objects within a domain. Hybrid uses a synchronous form of invocation with implicit return. However, to gain need flexibility the language also defines a "delegated" form of invocation. In a delegated invocation, the thread of the client is suspended until the server returns. While the client thread is suspended,

Fig. 8. A three element buffer in ABCL/1

another thread within the same domain as the client may be scheduled for execution. When the server returns the client thread is unblocked and will be scheduled to resume its execution at some time when its domain is idle. The Synchronization Model used by Hybrid is that of guarded ports, which are termed delay queues. Each operation of an object may be associated with a delay queue. An operation can only be invoked when its delay queue is open. The opening and closing of delay queues is done by code within the object's methods.

The synchronization aspects of Hybrid are illustrated in the example shown in Fig. 9. This example shows the familiar bounded buffer. The abstract part of the type description declares that invocations of `put` and `get` operations are controlled by a delay queue. The name of the delay queues is not relevant in the abstract type description. In the private part of the type definition two specific delay queues, `putDelay` and `getDelay` are created. The `init` method opens the `putDelay` queue (thus allowing the put method to be invoked) and closes the `getDelay` queue (thus forbidding a `get` method to be invoked as the first operation of the bounded buffer). The `put` method opens the `getDelay` queue (as a `get` method is now possible) and possibly closes the `putDelay` method when the buffer is full (to prevent the buffer from overflowing).

3.4 Procol

The Procol language [16, 27] incorporates a wide variety of interesting features including constraints and object persistence. The brief survey given here cannot do justice to the full scope of this language.

Fig. 9. Use of delay qeues in Hybrid

Objects in Procol are sequential and homogeneous. Two kinds of methods may be specified for an object: "actions" and "intactions". Actions are normal methods or operations. An intaction has the capability of interrupting the execution of an action. In this case the thread executing the action is suspended, the intaction is executed and the suspended action thread is then resumed. An intaction itself may not be interrupted by another intaction. Intactions are similar to express messages in ABCL/1.

Objects communicate through two message passing operations. A one-way asynchronous send operation is provided along with a synchronous request operation. The later operation is used when a return value is expected. To provide flexibility to the server, a form of delegated send is also included in the language. Similar to delegation in Hybrid and other languages, a delegated send allows the server to transfer to another object the responsibility for providing the return value needed by the client. One interesting feature of the send operation is that the target (the object to receive the message) may be indirectly named by specifying only the type of the target, not an instance of this type. In this case the message is delivered to any one instance of the specified type.

Return values are handled by an explicit form in which the server specifies the destination to which the result is to be sent. The common case is expressed as `send`

Fig. 10. A protocol declaration in Procol

The authors of Procol draw special attention to the clear separation between the specification of the protocol and the definition of the actions.

3.5 Pool-T

The Pool family of languages was developed specifically to support parallel programming. Philosophically these languages in this family are designed to be small, compact, cohesive languages with well understood interactions among its features.

In Pool-T [3] all entities are objects, all objects are sequential, and all objects communicate with each other via synchronous message passing. As is usually the case with synchronous invocations, the receiver of a server's return value is known implicitly by the server. One minor exception to the use of synchronous message passing allows an object to communicate with itself (i.e., invoke one of its own methods) through a simple function call. The exception is necessary to avoid self-deadlock when using the selective message acceptance mechanism described later.

Unlike many other concurrent object-oriented languages, Pool-T objects are "active" and not "reactive". That is, a Pool-T object may execute without the need to be enlivened by an arriving message. Each object has a "body" part that begins executing when the object is created. The body, at its own discretion, may choose to block in order to allow an invocation of one of its methods to occur. Since all objects are sequential, the body of an object and one of the object's methods cannot execute concurrently. When the body wishes to allow an invocation, the body executes an "answer" statement that names a list of methods. A single invocation of a method named in the answer's list is selected for execution. For additional flexibility, an Ada-like select construct also exists. This allows conditions to be associated with the acceptance of an invocation for a method. Thus, the Synchronization Model used in Pool-T is the guarded accept form of a concrete, conditional interface.

Two minor observations about Pool-T are the following. First, a server may "return", thereby unblocking the client, and yet continue in the same method to complete post-processing actions in parallel with the resumed client. Second, passive objects may be created by declaring an object with no body. In this case a default body is supplied. The default body simply accepts messages in sequential order.

3.6 ESP

The Extensible System Platform (ESP) [17] was develop at the Microelectronics and Computer Technology Corporation (MCC) as the software component of a research effort to develop flexible hardware and software components for building parallel systems. One of the major design goals of ESP was to retain as much as possible the syntax and semantics of C++ programming in ESP. Ideally, the parallelism of ESP would be transparent to the application developer. The ESP system is most appropriately used for distributed memory machines as it employs a medium-grain concurrency and fairly substantial kernel for object management and messaging.

In ESP all objects are homogeneous. However, as with several other concurrent extensions to C++ (e.g., ACT++), an ESP object may define as part of its local state information arbitrarily complex passive C++ entities. ESP objects are sequential. Concurrency in ESP, therefore, arises from the concurrent execution of different objects on different machines. By using the "placement syntax" of the C++ new operation, the programmer is given the ability to control or influence the placement of objects on the nodes comprising the execution environment. The programmer is able to specify the exact node on which the created object must run, specify that the object must (or must not) execute on the same node as another object, or specify that the object may execute on any node, thus allowing the system's load balancing strategy to decide on the object's placement.

The Interaction Model in ESP provides an asynchronous invocation mechanism matched with a future style of return by-value. Both of these promote higher levels of concurrent execution. The futures, however, are not first-class. An attempt to pass a future as an argument results in blocking until the value of the future is available. This form of future is more limited than that available in Eifell// or Mentat (both described later). However, ESP provides flexible ways of manipulating futures through a "future set". As implied by the name, the programmer may create a set of future variables associated with invocations that are in progress. It is then possible, for example, to await the availability of any one of the futures in the future set or await all of their availability.

The Synchronization Model in ESP is based on a technique called "method locking" that allows an object to lock or unlock its methods. When locked, a method cannot be invoked. Messages that would cause the invocation of a locked method are held pending in the object's mail queue until the method is unlocked. This form of synchronization falls in the category of a guarded ports style of concrete, conditional interface.

3.7 Ellie

A distinguishing aspect of the Ellie language [4] is the pervasive use of fine grain concurrency matched with a strategy for compile-time grain adaption. The aim of grain size adaption is to allow the compiler to aggregate several fine grain concurrent objects into one larger grain object in order to achieve more efficient execution on the target architecture. This approach confronts directly the phenomenon that different machine architectures support different grain sizes with varying efficiencies. For example, a massively parallel shared memory architecture will work better with fine grain objects than with large grain objects while the reverse is true for a distributed memory parallel machine or a distributed collection of workstations. The goal of Ellie is to allow the programmer to describe the solution to the problem while leaving the details of the grain size selection to the compiler. It remains to be seen how completely this goal can be achieved.

In Ellie, as in Pool-T, all entities are objects. Each object has a single associated process. Thus, Ellie is a homogeneous, sequential related object model. The language differentiates between functions, methods that have no side effects, and operations, methods that may have side effects. This distinction allows immutable objects to be recognized and used efficiently in a parallel environment as the immutable object may be freely copied.

Invocations in Ellie are synchronous. The value return mechanism distinguishes between "bounded processes"s that act like a synchronous invocation and 'unbounded processes" that act like a future. A bounded process is created to execute a method that is a regular function or method. Invocations of methods declared as "future functions" or "future operations" cause the creation of unbounded processes. Thus, the Interaction Model used in Ellie is synchronous and provides an explicit return mechanism through futures.

Delegation is also provided in Ellie through a mechanism called "interface inclusion". The following example is provided in [4]:

```
export local_1, (local_2), (local_3).(remote_1, remote_2)
```

When given as the export interface of object X, the above declaration implies that the interface of X is taken to be the union of:

Fig. 11. Controlling the accept interface in Ellie

An example of an accept interface is given in Fig. 11. A semaphore object is defined with an accept interface of `accept wait, signal;`. In each method, the `include` statement defines a new accept interface. (The `include` statment is misleading since it does not add the specified names to the accept interface; but rather its defines a new accept interface). Thus, when the integer `val` reaches zero, the `wait` operation is removed from the accept interface. When a method's name is not in the current accept interface, processes trying to invoke that operation are suspended. Therefore, when `val` is zero processes trying to execute a `wait` method will be delayed in accordance with the usual understanding of a semaphore. When a `signal` operation is performed, so that `val` has the value one, the `wait` operation is reinserted into the accept interface. A process suspended on the `wait` method would now find the method available in the interface and its invocation could then be initiated.

3.8 Sina

The Sina language [5, 26] makes straightforward choices in its Animation and Interaction Models. Invocations in Sina are synchronous with an implicit return. In the Animation Model, all objects are uniformly active. The language, however, is somewhat ambivalent in its position on multiplicity. By default the Sina preprocessor generates only sequential objects. However, the user may override this default, in which case all objects have unrestricted concurrency. The language thus offers both of the two extreme choices in this factor. Note, however, that these two alternatives cannot be combined in a single application.

The most interesting aspect of Sina is its Synchronization Model that uses a technique called "composition filters". The basic concept is that a message is subject to screening by any number of filters. The filters are applied in an ordered determined by the object's definition. An invocation occurs only when a message passes through all of the filters defined for the target object. A filter is created out of conditions, predicates over the state variables of the object. Thus, Sina synchronization falls into the behavioral class of an abstract, conditional interface.

The composition filters mechanism will be explained by reference to the example in Fig. 12, which shows the definition of a synchronized stack. First, note that the object definition is divided into an interface part and an implementation part as indicated by the keywords. In the interface specification, the keyword `internals` introduces private data members of the class. An instance of a predefined unsynchronized class named Stack is declared as a private member of the SyncStack class. The keyword `conditions` introduces named condition variables. The implementation of these conditions is given in the implementation part. In this example the `NonEmpty` condition is defined in terms of the size of the internal `inStack` member. The `Inputfilters` keyword introduces the filters to be applied to incoming messages. In this example there are two input filers. The first input filter is named `sync` and is of type `Wait`. This filter declares that the `pop` method can only be invoked when the `NonEmpty` condition is true; all other methods except `pop` can be invoked at any time. The second input filter is named `disp` and is of type `Dispatch`. This is a predefined type of filter that indicates that all of the methods of the `inStack` member may be invoked. A filter of type `Dispatch` is used to indicate that any message that succeeds in reaching this filter may be invoked.

Sina also allows synchronization counters to be included in the definition of filters. The standard object model defines a number of such counters including dispatched, completed and received that indicate how many messages are in each of these states.

3.9 Presto

Presto [6] is a C++-based framework for parallel programming of shared memory multiprocessors. The goal of Presto is to provide low-level mechanisms for developers to create specialized, domain-specific abstractions. One kind of "developer" are those experimenting with various concurrent programming models that are more highly

Fig. 12. Example of Sina composition-filters

structured than is Presto itself. That Presto has had some success in achieving this goal is evidenced by the ACT++ system described earlier, which is implemented using Presto. Thus, Presto is in the unrelated class in the Animation Model in keeping with its role as a low-level, neutral base.

In terms of the Interaction Model, both synchronous and asynchronous invocations are possible. The synchronous style of invocation is that of the base C++ language. The asynchronous form of invocation is provided via the threads management facilities. Fig. 13(a) shows the basic facility of asynchronous invocation. In this example a stack, `S`, is created. Also created is a thread, `t`, which will asynchronously execute the `push` method on object `S`. The single argument of the `push` operation is the integer value `43`. Notice that the argument list cannot be type checked against the definition of the `push` method. Notice also that the implementation of the stack is unaware of the fact that it is being invoked asynchronously. Figure 13(b) illustrates how multiple threads can be created to perform matrix multiplication. In this usage, the `Matrix::multiply` method creates concurrency that is transparent to the client. Also observe that the multiply method can be invoked either synchronously or asynchronously without any change to the `Matrix` class itself. Finally, return values are handled implicitly. For an asynchronous invocation, a thread `join` operation is provided by which the return value can be extracted from a thread previously started in some method.

The Presto Synchronization Model is an unconditional interface. To provide basic mechanisms for creating more structured interface semantics, Presto provides prim-

Fig. 13. Presto thread operations

itive locks and a monitor-style set of facilities. These basic synchronization tools allow low-level control of the invocation process to be created. This is possible since the threads facility is implemented as a C++ class that can be subclassed to create other forms of asynchronous invocations. Alternatively, a thread may be created only when an invocation has been accepted, rather than when the invocation is initiated. This later strategy is used in the ACT++ system.

3.10 Guide

Guide [15] is a language developed for the programming of distributed applications that manipulate permanent data. The language and an operating system (also called Guide) were developed at the University of Grenoble. Aside from the features explicitly considered below, Guide provides a separation between the type hierarchy and the class (implementation) hierarchy, distribution transparency and automatic persistence.

Like Presto, Guide falls in the unrelated class in the Animation Model. A distinction is drawn between passive, persistent objects and the execution vehicles, called "jobs" and "activities". A job represents the execution of an applications that consists of several activities. Activities are created by means of a simple `cobegin-coend` construct.

The Interaction Model provides synchronous invocations and implicit returns. The Guide programming paradigm is one in which jobs or activities operate directly on passive objects and, if communication or synchronization is desired, coordination is achieved through shared passive objects. The shared passive objects are provided with synchronization mechanisms to safeguard their internal consistency in the face of concurrent invocations.

The Synchronization Model of Guide is a behavioral form of an abstract conditional interface. A passive object may regulate the sequence of invocation that it accepts through the specification of a `control` clause. The control clause may name one or more methods defined in the object and, for each method, list an "activation condition". The activation condition is a predicate that must be true in order for the method with which it is associated to be invoked. The activation condition may refer to instance variables defined in the object, the actual parameters contained in the invocation and special "activation counters". The activation counters, similar to those defined in Sina, provide counts of the following for each method: the number of invocations of the method, the number of accepted invocations of the method and the number of completed executions of the method. Other conditions may defined in terms of these basic ones.

An example of the use of the Guide mechanisms is shown in Fig. 14. In this example the implementation of a typical bounded buffer is given. The interesting aspect of this example is the control clause at the end of the class definition. This clause specifies constraints on the invocations of both the `Put` and `Get` methods. The activation condition associated with the `Put` method states that the number of complete `Put` invocations (each adding an element to the buffer) cannot exceed by more than the size of the buffer the number of complete `Get` invocations (each removing an element from the buffer). Furthermore, the activation condition also insist that no `Put` methods are executing concurrently. The activation condition for the `Get` method requires that the buffer is not empty (i.e., that more `Put` invocations have completed than `Get` invocations).

3.11 Eiffel//

The primary goal of Eiffel// [8], a concurrent extension to the Eiffel language, is to achieve performance improvements with as few changes as possible to an application developed as a non-concurrent Eiffel system. This goal is similar to that of ESP, which retains the programming model of C++ but allows for transparent distributed execution. In Eiffel// class inheritance is used extensively to achieve its goals.

In terms of the Animation Model, Eiffel// uses a related approach that is homogeneous and sequential. The language is classified as homogeneous because, even though both active and passive objects are permitted, passive objects are required to be private objects of an single active object. This restriction is similar to that in ACT++. In Eiffel//, passive objects are passed among active objects only by value (i.e., by a deep copy). Each active object is a member of a class derived from the predefined base class `PROCESS`. A distinguished method, `Live`, is provided by the `PROCESS` class, but may be redefined in a subclass. The `Live` method, automatically

Fig. 14. Synchronization of a passive object in Guide

invoked when the object is created, defines the script or code for the active object. The default Live method accepts messages in FIFO order.

In terms of the Invocation Model, the language uses asynchronous invocations and returns values by a mechanism termed "wait by necessity". This return mechanism is an explicit return, by-value in the style of futures. As in other future mechanisms, an invocation immediately yields an "awaited object". An attempt to access the value of an awaited object causes the accessing process to block. A powerful aspect of Eiffel// is that the awaited object can be passed as an argument or assigned to another object without blocking. Only the explicit attempt to obtain the value of the awaited object causes blocking to occur.

The Synchronization Model provided by Eiffel// is an interesting one and is similar in many respects to the behavior set concept in ACT++, which is detailed in the next section of this paper. The technique employed in Eiffel// is that of a behavioral form of an abstract, conditional interface. The key concept in Eiffel// is that both "routines" (i.e., methods) and "requests" (e.g., invocations) are first class objects amenable to manipulation by the provided base classes in Eiffel// and by the application developer. To support the desired synchronization structure two base classes are provided, `ROUTINE` and `REQUEST` . The operator `&` applied to a routine yields an object of type `ROUTINE`. Mechanisms are also available to examine the set of `REQUEST` objects pending at a given object.

Synchronization in Eiffel// will be illustrated with reference to the example in Fig. 15. This figure shows the skeleton of the `ABSTRACT_PROCESS` class, derived from the `PROCESS` base class, and the `ABST_BUFFER` class derived from both the `FIXED_LIST` and `ABSTRACT_PROCESS` classes. The key elements in the `ABSTRACT_PROCESS` class are the routines `associate` and `synchronization`. The `synchronization` routine is

Fig. 15. Synchronization in Eiffel//

3.12 Mentat

Mentat [10] is both a language and run-time system. The Mentat Programming Language (MPL) is based on C++. The primary goal of Mentat is to allow the same application to be run across different parallel architectures with no changes in the application itself. This goal is achieved by compile time analysis of the program written in MPL. In this sense Mentat is similar to Ellie in its use of compiler technology to support parallel programming. However, Ellie uses fine-grain concurrency while Mentat uses a courser grain of concurrency. The Mentat compiler translates from MPL into a macro data flow representation that is then analyzed to discover inherent parallelism. The C++ code generated by the MPL compiler is executed in the Mentat virtual machine environment. Portability across different machine architectures is achieved by porting the Mentat run-time system. The authors of Mentat report that they regularly write Mentat programs that run unchanged across shared memory and distributed memory machines.

The Mentat language takes a homogenous and concurrent approach in the Animation Model. Each Mentat object is active and concurrency is achieved both among Mentat object (inter-object concurrency) and, possibly, within Mentat objects (intra-object concurrency). The Mentat compiler, of course, plays a key role in discovering and exploiting such forms of concurrency although inter-object concurrency is fairly apparent in the structure of the Mentat program itself.

The Interaction Model in Mentat uses asynchronous invocations and a value-based form of explicit returns, which is similar to a future. The difference between the Mentat return mechanism and a future is that the future in Mentat is implicit. The programmer simply declares the invocation and names the variable to receive the return value. Since the invocation is asynchronous, the value of the return variable is initially undefined. However, the Mentat compiler generates code that allows the continued execution of the sender until such time as the sender actually needs the return value. In addition, as in Eiffel//, the variable to receive the result value may be passed as an argument or used in an assignment without causing the execution to block. Mentat carefully records all objects that are dependent on the return value. When the invoked method finally returns, via the "return to future" (`rtf`) primitive, a result value is delivered to all objects awaiting the value.

An important measure of Mentat's success, and for other similar approaches as well, is the speed-up that can be obtained across different machine architectures. In [10] the following performance data is reported for Mentat. Two applications, matrix multiply and Gaussian elimination, were run on two different machine architectures, a network of eight Sun workstations and a 32-processor Intel iPSC/2. For the matrix multiply problem of size 400 a maximum speed-up of between 6 and 7 was obtained on the Sun workstations and a maximum speed-up of between 18 and 20 was obtained on the iPSC/2 for a matrix size of approximately 300. For the Gaussian elimination problem, a maximum speed-up of approximately 4 was obtained on the Sun workstations for a matrix of size 500 and a maximum speed-up between 12 and 14 was obtained on the iPSC/2 for a matrix of size 350. The Mentat authors acknowledge that better speed-up could probably be obtained in each case by specialized programming for each individual architecture. They argue, however, that the simplicity and portability of the Mentat approach is ultimately of higher value especially given the rapid advances in parallel processing architectures and the need to execute applications on different architectures throughout their lifetimes without the substantial cost of re-egineering the application for each new architecture.

4 The Inheritance Anomaly and its Solution in ACT++

In a concurrent object-oriented language, one would like to be able to inherit behavior and realize synchronization control without compromising the flexibility of either the inheritance mechanism or the synchronization mechanism. A problem called the inheritance anomaly [19] arises when synchronization constraints are implemented within the methods of a class and an attempt is made to specialize methods through inheritance.

In this section, concurrent object behavior is formalized using the equational notation of the CCS [20]. The formalization facilitates a characterization of the fundamental cause of the inheritance anomaly, and leads to the definition of a set of conditions that are necessary for inheritance and synchronization constraints to coexist in concurrent object-oriented languages.

For those not familiar with the inheritance anomaly, the problem is reviewed in Sect. 4.1. Those familiar with the problem may wish to continue with Sect. 4.2 where the notation of CCS is used to characterize and reason about the behavior of a concurrent object. The characterization results in the definition of the set of conditions that are necessary to avoid having to reimplement synchronization constraints when specializing a class. Section 4.3 demonstrates that the characterization is applicable to real programming problems. Two examples are given that show that behavior sets can be implemented in C++ extended through inheritance to support Actor-style concurrency.

4.1 The Inheritance Anomaly

An object is an encapsulation of state and code in the form of instance variables and methods, respectively. In object-oriented languages, the declaration of the types and names of instance variables and the signatures of the methods are typically declared in an explicit class definition. The class definition serves as a contract between objects of that class and clients or subclasses that intend to use instances of the class. A subclass is a specialization of a particular class. Specialization is achieved through the class inheritance mechanism.

In standard object-oriented models, all the methods declared in a class definition are always available for execution by a client regardless of the internal state of an object of that class. The class implementor typically provides, within the implementation of each method, the code necessary to determine whether or not the object is in a state in which execution of the requested method is appropriate.

For example, a stack object with a pop method must first verify that the stack is non-empty before proceeding. Typically, the pop method will return an error value indicating that an underflow condition has occurred. The traditional mechanism for communicating the underflow condition to the client requires an overloading of the return type of the method; that is, the return value is outside the domain of values returned by a legitimate operation. The client of the stack object must be aware of this value and always verify that either the stack is a state consistent with the required operation or check the return value of each operation. Alternatively, an exception mechanism might be used.

A non-standard object-oriented model can be defined in which the collection of methods in a class definition are partitioned into subsets with respect to the values of the state variables of an object. Depending on the object state, only a subset of the methods declared in the class definition are available for execution at any one time. In a sequential object-oriented language, an object interace with these semantics may or may not be useful. In a concurrent object-oriented language, an interface mechanism based on such semantics provide a natural and elegant means for expressing synchronization control.

In this section only concurrent objects that by necessity employ some form of synchronization control are of interest. Of particular interest is the specialization of the synchronization constraints defined as part of the representation for such objects. The concurrent behavior of an object is captured in part by the static class definition of the object and in part by the dynamic mechanism employed by the method interface to guarantee synchronization. The inheritance anomaly occurs when an attempt is made to specialize concurrent behavior using an inheritance mechanism. The anomaly occurs when a subclass violates the synchronization constraints assumed by the base class. A subclass should have the flexibility to add methods, add instance variables, and redefine inherited methods. Ideally, all the methods of the base class should be reusable. However, if the synchronization constraints are defined by the superclass in a manner prohibiting incremental modification through inheritance, the methods cannot be reused, they must be reimplemented to refelect the new constraints; hence, inheritance is rendered useless. Recent work on the problem has demonstrated that the anomaly occurs across a spectrum of concurrent object-oriented languages, regardless of the type of mechanism employed for specifying synchronization constraints [2, 7, 12]. A deeper issue is that the concurrent object-oriented research community does not yet have a good semantic model that relates the type features and the concurrency features of concurrent object-oriented languages. In the following, the inheritance-synchronization conflict is addressed in a formal way. A formalism is presented that exposes the essential elements of concurrent object behavior and leads to conditions that must exist if the inheritance anomaly is to be avoided.

4.2 Defining Concurrent Object Behavior

The behavior of an object is defined by the set of messages that the object will accept at a given point in time; or alternatively, the set of methods that are visible in the interface of the object upon receipt of a message. From this perspective, the behavior of an object is its observable behavior since all that is relevant is how the object appears to those clients that communicate with the object. This notion of observable behavior is motivated by a similar notion described in [20]; however, the machinery of CCS is used here in a superficial manner to characterize the behavior of individual objects, not systems of objects.

In dealing with concurrent objects, the relationship between the state of an object and the subset of methods that define its observable behavior is critical. In order to understand how to implement and then inherit synchronization constraints without encountering the inheritance anomaly, this relationship must be clearly defined.

4.3 Specifying Behavior

The behavior of an object may be defined as a set of behavior equations that capture the states of an object and the subset of methods that are visible when the object is in a particular state. Informally, the "state of an object" is the set of instance variable-value pairs that define the object at a particular step in a computation.

As an example, the behavior of an object that maintains some prescribed linear order over a collection of items, whose size is bounded, is defined. The observable behavior of an object representing a bounded linear order is completely described by the following equations:

$$\begin{aligned} A_0 &\stackrel{\text{def}}{=} \mathtt{in}(x_1).A_1(x_1) \\ A_1(x_1) &\stackrel{\text{def}}{=} \mathtt{in}(x_2).A_2(x_1,x_2) + \overline{\mathtt{out}}(x_1).A_0 \\ &\vdots \\ A_n(x_1,\ldots,x_n) &\stackrel{\text{def}}{=} \overline{\mathtt{out}}(x_1).A_{n-1}(x_1,\ldots,x_{n-1}) \end{aligned}$$

This set of behavior equations is similar to an example in [20]. The equations capture precisely the states that an object representing a bounded linear order may occupy during its lifetime. In the equations, only the prefix (.) and summation (+) combinators of CCS are required. In each of the equations the name on the left-hand side denotes an agent whose behavior is defined by the right-hand side. Intuitively, agent $A_i(x_1,\ldots,x_i)$ represents the behavior of the object when the size of the collection is i, where $1 \le i \le n$, with A_0 representing the empty collection. One can verify through induction that this set of equations defines all possible behaviors of a bounded linear order.

In the behavior definition of the $A_1(x_1)$ agent, the summation combinator conveys that the agent offers both the `in` and $\overline{\mathtt{out}}$ operations simultaneously to a client. If the `in` operation is chosen, the prefix combinator requires that an agent accept an input value denoted by x_2 and then become agent $A_2(x_1,x_2)$. Similarly, if the $\overline{\mathtt{out}}$ operation is chosen, the agent outputs a value denoted by x_1 and then assumes the behavior defined by agent A_0.

In general, agent $A_i(x_1,\ldots,x_i)$ becomes agent $A_{i+1}(x_1,\ldots,x_{i+1})$ following an `in` operation and agent $A_{i-1}(x_1,\ldots,x_{i-1})$ following an $\overline{\mathtt{out}}$ operation, with the behavior of agents A_0 and $A_n(x_1,\ldots,x_n)$ being special cases. From this perspective, the behavior equations define the operations offered by an agent as well as a replacement behavior. The notion of replacement behavior is a fundamental aspect of the Actor model [1]. Hence, it seems appropriate to use behavior equations as a formal means for specifying and reasoning about the behavior of individual objects with actor-like semantics.

Although a generic bounded linear order is specified, the above set of behavior equations is isomorphic to a set of equations representing a bounded buffer accepting `put` and `get` operations, a stack accepting `push` and `pop` operations, or a queue accepting `enq` and `deq` operations. The isomorphism is realized through an application of the CCS relabeling (/) operator to yield the desired name substitution:

$$\begin{aligned} \text{Buffer} &\equiv A_0[\mathtt{put/in},\mathtt{get/out}],\ldots,A_n(x_1,\ldots,x_n)[\mathtt{put/in},\mathtt{get/out}] \\ \text{Stack} &\equiv A_0[\mathtt{push/in},\mathtt{pop/out}],\ldots,A_n(x_1,\ldots,x_n)[\mathtt{push/in},\mathtt{pop/out}] \\ \text{Queue} &\equiv A_0[\mathtt{enq/in},\mathtt{deq/out}],\ldots,A_n(x_1,\ldots,x_n)[\mathtt{enq/in},\mathtt{deq/out}] \end{aligned}$$

The isomorphism can be achieved because at this level of abstraction there is no concern for the actual semantics of the in and out operations, e.g., whether or not the out operation returns values according to FIFO or LIFO semantics. To maintain generality, the equations describing a bounded linear order are used in the remainder of this paper with the understanding that the isomorphism can be applied at any time.

4.4 Object States and Behavior Sets

Behavior equations may be viewed as defining independent agents representing the various states of an object. In this section a model is defined that captures the essential elements used in developing a programming abstraction to represent a collection of behavior equations.

In the model, each agent is associated with an object state σ_i and a set β_i called the observable behavior set. For a given behavior equation, the observable behavior set β_i is constructed from the non-restricted prefix operations on the right-hand side of behavior equations. Non-restricted prefix operations are those operation names that do not appear within the scope of the CCS restriction operator, thereby removing those operations from the set of observable behaviors.[3] The collection of all states is given by the state set $S = \{\sigma_0, \sigma_1, \ldots, \sigma_n\}$. The set of all possible observable behavior sets is the powerset $B = \mathcal{P}(M)$, where $M = \bigcup_{i=0}^{n} \beta_i$. To complete the model, a function relating states to behavior sets is required.

4.5 Mapping States to Behavior Sets

The function $f_\beta : S \rightarrow B$ maps elements of the state set to elements of the powerset of observable behaviors. Once can argue that in developing an abstract data type of a bounded linear order in a sequential object model, a programmer implicitly defines a mapping from S to B when defining the operations on the type. More precisely, each σ_i is always mapped to a single element in B, namely M, where $M = \left\{\mathtt{in}, \overline{\mathtt{out}}\right\}$. The map f_β is called the *behavior function* since it defines the observable behavior (the set of available methods) of an object in a given state. In standard object models, all methods are usually visible regardless of the state of the object; hence, f_β is defined in a manner that does not distinquish the internal object states, i.e.:

$$\begin{aligned} f_\beta(\sigma_0) &= M \\ f_\beta(\sigma_1) &= M \\ &\vdots \\ f_\beta(\sigma_n) &= M \end{aligned}$$

Hence, objects in a standard object model always have the same behavior set regardless of any transitions in the state of the object.

The definition of f_β in the case of σ_0 and σ_n is unnatural. In defining $f_\beta(\sigma_0) = \left\{\mathtt{in}, \overline{\mathtt{out}}\right\}$, a programmer is forced to implement the method implementing the out operation in such a way that an underflow condition is detected. A similar situation

[3] The restriction operator is a primitive mechanism for defining a scope.

occurs for $f_\beta(\sigma_n)$. A more natural mapping from S to B can be given as follows:

$$\begin{aligned} f_\beta(\sigma_0) &= \{\texttt{in}\} \\ f_\beta(\sigma_1) &= M \\ &\vdots \\ f_\beta(\sigma_n) &= \{\overline{\texttt{out}}\} \end{aligned}$$

Consider that one wants to realize the semantics of the behavior equations by implementing an object in an appropriate object-oriented language that embodies the abstraction of a linear order. The mapping defined by f_β indicates that a useful abstraction is to partition the above behavior equations into three sets based on the notion of the state of the object. This partitioning appeals to intuition about the behavior of a linear ordering; furthermore, it is only necessary to reason about three behaviors, not $n+1$; that is, the structure is either empty, full, or partially full. A reasonable implementation of the abstraction just formed is to define a class that exports two public methods, `in` and `out`, and that either explicitly or implicitly implement synchronization constraints consistent with the behavior equations previously formulated. That is, the `out` method is prohibited when the object corresponds to the agent defined by A_0 (state σ_0) and the `in` method is prohibited when the object corresponds to $A_n(x_1, \ldots, x_n)$ (state σ_n).

To extend the example, suppose that an additional constraint is introduced. For example, the behavior given by the $A_1(x_1)$ equation is distinquished from the other behaviors because a new operation is introduced that augments the other behavior sets. In distinguishing the $A_1(x_1)$ behavior, a new partitioning must be defined that is distinquishes the conditions empty, full, singleton, and partial.

If an attempt is made to specialize the previously implemented abstraction through inheritance, it becomes necessary to extend the mapping given by f_β. Extending the mapping means that the domain S, the codomain B, and the mapping of elements in S to elements in B must be redefined. If the linear order abstraction has been implemented such that these components are implicitly imbedded in the class methods, then it is impossible to reuse the methods. Reuse is made impossible because the components of the mapping f_β are implicitly imbedded in the implementation. The only way the mapping can be redefned to permit the new synchronization constraints is to reimplement the methods in which the components representing the mapping f_β are embedded. The solution presented in the next section is to separate the components of the mapping from the method implementations.

4.6 Inheriting Concurrent Behavior

The types of concurrent object-oriented systems of interest are composed of objects with concurrency properties consistent with those described in the Actor model. Each object possesses its own thread of control and communicates with other objects via message passing. Concurrency in the system is limited to inter-object concurrency, which is achieved using message passing and an actor-like become operation. The actor `become` operation results in a replacement behavior with its own thread of control. Fine-grained intra-object concurrency is not a feature of objects in the systems under consideration here.

A specific interest is expressing and inheriting concurrent object behavior in ACT++ [13, 14, 18], a prototype object-oriented language based on the Actor model and C++ [9]. ACT++ is a collection of classes that implement the abstractions of the Actor model and integrates these abstractions with the encapsulation, inheritance, and strong-typing features of C++. The language falls in the non-orthogonal category of concurrent object-oriented languages [23], since there are both active and passive objects. Active objects are instances of any class derived from a special `Actor` class defined as part of the language run-time. Any instance of a class not derived from the `Actor` class is a passive object. Concurrency is achieved using the become operation that is implemented in the `Actor` class.

The notion of "behavior abstraction" was originially proposed in ACT++ as a mechanism for capturing the behavior of an object. Upon initial examination, behavior abstraction seems powerful since synchronization can be achieved naturally by dynamically modifying the visibility of the object interface using the become operation. The efficacy of this mechanism and its degree of interaction with the C++ inheritance mechanism has been examined by others and has been found to have serious limitations [19, 23]. The most serious limitation occurs because a behavior abstraction is not a first-class entity in the language and is thus subject to the effects of the inheritance anomaly.

A construct called an "enabled set" improves on the notion of behavior abstraction by promoting the control of the visibility of an object's interface to a dynamic mechanism that can be manipulated within the language. Enabled sets were implemented in Rosette, an interpreted actor language with dynamic typing [25].

The flexibility offered by enabled sets is difficult to achieve in a statically typed language like ACT++. Behavior sets represent a compromise between the enabled sets and behavior abstraction. The ACT++ language mechanism that represents the behavior set has the following properties:

1. it is a natural extension of formal methods for specifying concurrent object behavior,
2. it does not interfere with the C++ inheritance mechanism,
3. it is free from known inheritance anomalies,
4. it can be expressed entirely within ACT++ (hence C++), and
5. it can be enforced efficiently at run time.

In the following sections, the syntax of ACT++ is used to illustrate how to express elements of the object state set S, subsets of the observable behavior powerset B, and the behavior function f_β, such that synchronization constraints may be implemented and inherited in a manner that supports method reuse.

4.7 Expressing Concurrent Behavior

To represent concurrent object behavior within the ACT++ language, three first-class entities expressible within the language are defined:

1. *state functions* representing some or all of the elements of the state set S,

Fig. 16. ACT++ linear order class definition

[4] Technically the `LinearOrd` class is defined as a C++ *template* that is parameterized by a type `T` representing the type of the elements in the linear order.

Although not shown, the functions are computed based on implementation dependent instance variables representing the actual number of elements in the structure representing the linear order. Both functions are used by the `nextBehavior` function, which maps the current object state to a behavior set represented by an instance of the `BehaviorSet` class. There are three behavior sets defined: B_0, B_n, and B_i. The B_0 and B_n behavior sets correspond to the previously expressed behavior equations A_0 and $A_n(x_1, \ldots, x_n)$, respectively. The B_i behavior set is used in this abstraction to collectively represent the observable behaviors of the intermediate behavior equations. Each behavior set is initialized in the class constructor. Instances of the `BehaviorSet` class are first class objects and an overloading is given to the binary addition operator denoting set union when applied to two behavior sets; hence, B_i is formed as the union of the behavior sets B_0 and B_n.

4.8 Inheriting Concurrent Behavior

To substantiate the claim that the inheritance anomaly is avoided, a new class `HybridLinearOrd` is derived from the `LinearOrd` class. The main feature of the `HybridLinearOrd` class is that a new method is introduced that forces a change in the mapping given by f_β. The new method allows a client of an instance of the `HybridLinearOrd` class to atomically extract a pair of elements instead of a single element. The method cannot simply invoke the `out` method twice since the `out` method executes a `become` operation after each invocation. Due to the concurrency in the system, another object may have its `out` request executed before the second `out` request is processed; therefore, a new operation $\overline{\texttt{outp}}$ is required to output a pair. The behavior of this new type of object is specified by the following behavior equations:

$$\begin{aligned}
A_0 &\overset{\text{def}}{=} \texttt{in}(x_1).A_1(x_1) \\
A_1(x_1) &\overset{\text{def}}{=} \texttt{in}(x_2).A_2(x_1, x_2) + \overline{\texttt{out}}(x_1).A_0 \\
A_2(x_1, x_2) &\overset{\text{def}}{=} \texttt{in}(x_3).A_3(x_1, x_2, x_3) + \overline{\texttt{out}}(x_1).A_1(x_1) + \overline{\texttt{outp}}(x_1, x_2).A_0 \\
&\ \ \vdots \\
A_n(x_1, \ldots, x_n) &\overset{\text{def}}{=} \overline{\texttt{out}}(x_1).A_{n-1}(x_1, \ldots, x_{n-1}) + \overline{\texttt{outp}}(x_1, x_2).A_{n-2}(x_1, \ldots, x_{n-2})
\end{aligned}$$

The behavior equations for the hybrid linear order differ from the equations specifying the behavior of a traditional linear order only in the addition of the choice of an $\overline{\texttt{outp}}$ operation in the definitions of the $A_i(x_1, \ldots, A_i)$ equations, where $2 \leq i \leq n$. There are two effects of this refinement. First, the $\overline{\texttt{outp}}$ operation is added to the appropriate observable behavior sets and a new powerset B' is computed. Second, the $A_1(x_1)$ behavior is now a distinguished behavior and $B' \supset B$; hence, a new mapping f'_β is required:

$$\begin{aligned}
f'_\beta(\sigma_0) &= f_\beta(\sigma_0) \\
f'_\beta(\sigma_1) &= f_\beta(\sigma_1) \\
f'_\beta(\sigma_2) &= \{\texttt{in}, \overline{\texttt{out}}, \overline{\texttt{outp}}\}
\end{aligned}$$

Fig. 17. ACT++ hybrid linear order class definition

The `singleton` function corresponds to distinguishing the agent $A_1(x_1)$ from the remaining agents. The new `BehaviorSet` object B_1 corresponds to the behavior set associated with agent $A_1(x_1)$. The B_i and B_n behavior sets are augmented in the class constructor with the `outPair` method corresponding to the enlarged codomain B'. Thus, the inherited `nextBehavior` function can be trivially redefined to correspond to the new mapping f'_β by adding a check for the state corresponding to agent $A_1(x_1)$ and invoking the superclass behavior function `LinearOrd::nextBehavior` for all other states.

In order to inherit synchronization constraints it must be possible to specialize the mapping given to f_β by the superclass. This implies that the elements of S, the elements of B, and the function f_β must be representable in the language and the representations must be:

1. first-class,
2. inheritable, and
3. mutable.

The inheritance anomaly occurs in previous formulations of this problem precisely because the behavior sets and the behavior function, as they occurred in the superclass, were neither first-class nor mutable.

State functions representing elements of S, instances of the `BehaviorSet` class representing elements of B, and the `nextBehavior` function representing f_β have these properties. All are firstclass language entities inheritable by a subclass. Instances of the `BehaviorSet` class are mutable by a subclass since they are defined within the scope of a protected clause. The `empty` and `full` predicates representing object states and the `nextBehavior` function representing the behavior function are subject to redefintion since they have the `virtual` attribute are within the scope of the `protected` clause.

5 Summary and Status

This paper has presented the beginnings of a formal framework for investigating the relationship between concurrent object behavior and inheritance. The approach emphasizes the relationship between the state of an object and subsets of methods visible in the interface to the object, called behavior sets. This relationship is embodied in the mapping given by the behavior function. If the inheritance anomaly is to be avoided, behavior sets and the behavior function must be first-class, inheritable, and mutable. It was shown that the language mechanisms of ACT++ (and therefore C++) are sufficiently expressive in this regard.

We are exploring the techniques presented here in the context of distributed object-oriented systems with a high degree of both intra-node and inter-node concurrency. In particular, we are developing an object-oriented structure for peer-to-peer protocols and are investigating concurrency issues. We are also addressing the semantic issues in a more rigorous fashion than is presented here. We suspect that type-theoretic semantics currently applied to object-oriented languages are incapable of addressing the temporal nature of a changing object interface as captured by the behavior function. Interesting work in this area is [22], which also uses CCS as a starting point.

We have not discussed the details of the run-time enforcement of behavior sets. ACT++ and behavior sets have been implemented on the Sequent Symmetry, a shared memory multiprocessor [14]. A subject of our current research is to determine the relationship between our implementation approach and others based on reflection. Our Actor language prototype continues to evolve as we gain an understanding of the semantic issues underlying concurrent object-oriented languages.

6 Acknowledgements

The previous work of Keung Hae Lee on behavior abstraction in ACT++ and the work on enabled sets in Rosette by Chris Tomlinson and Vineet Singh set the stage for the results described here. Satoshi Matsuoka formulated the inheritance anomaly in the context of behavior abstractions in ACT++, which motivated us to pursue the issue more deeply. We thank Mani Mukherji for implementing the behavior set mechanism in ACT++.

References

1. G. Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems, MIT Press, 1986.
2. P. America. "Inheritance and subtyping in a parallel object-oriented language," ECOOP'87 Conference Proceedings, pp. 234–242, Springer-Verlag, 1987.
3. P. America. "POOL-T: A parallel object-oriented language," in Object-Oriented Concurrent Programming, ed. A. Yonezawa, M. Tokoro, pp. 199–220, The MIT Press, Cambridge, Massachusetts, 1987.
4. B. Anderson. "Fine-grained parallelism in Ellie," Journal of Object-Oriented Programming, 5(3), pp. 55–61, June 1992.
5. L. Bergmans, M. Aksit, K. Wakita, and A. Yonezawa. "An object-oriented model for extensible concurrent systems: the composition-filters approach," in Trese Papers on OO Software Engineering, Department of Computer Science, University of Twente, Netherlands.
6. B. Bershad and E. Lazowska and H. Levy. "Presto: A system for object-oriented parallel programming," Software: Practice and Experience, 18(8), pp. 713–732, August 1988.
7. J-P. Briot and Akinori Yonezawa. "Inheritance and synchronization in object-oriented concurrent programming," in ABCL: An Object-Oriented Concurrent System, (ed. A. Yonezawa), MIT Press, 1990.
8. D. Caromel. "Concurrency: An object-oriented approach," TOOLS-2 Conference Proceedings, (eds. J. Bexivin, B. Meyer and J. M. Nerson), pp. 183–197, 1990.
9. M. A. Ellis and B. Stroustrup. The Annotated C++ Reference Manual, Addison-Wesley, 1990.
10. A. Grimshaw. "Easy-to-Use object-oriented parallel processing with Mentat," IEEE Computer, 26(5), pp. 39–51, May 1993.
11. M. Jackson. System Development, Prentice-Hall, 1983.
12. D. Kafura and K. H. Lee. "Inheritance in actor based concurrent object-oriented languages," ECOOP'89 Conference Proceedings, pp. 131–145, Cambridge University Press, 1989.

13. D. Kafura and K. H. Lee. "ACT++: building a concurrent C++ with actors," Journal of Object-Oriented Programing, 3(1), pp. 25–37, May/June 1990.
14. D. Kafura, M. Mukherji, and G. Lavender. "ACT++ 2.0: A class library for concurrent programming in C++ using actors," Journal of Object-Oriented Programing, to appear 1993.
15. S. Krakowiak, et al. "Design and implementation of an nbject-oriented, strongly typed language for distributed applications," Journal of Object-Oriented Programming, September/October, pp. 11–22, 1990.
16. C. Laffra. PROCOL: A Concurrent Object Language with Protocols, Delegation, Persistence and Constraints, Ph.D. Dissertation, Erasmus University, Rotterdam, 1992.
17. W. J. Leddy and K. S. Smith, "The design of the experimental system's kernel," Proceedings: Hypercube and Concurrent Applications Conference, Monterey, California, 1989.
18. K. H. Lee. Designing a Statically Typed Actor-Based Concurrent Object-Oriented Programming Language, Ph.D. Dissertation, Department of Computer Science, Virginia Tech, June 1990.
19. S. Matsuoka, K. Wakita, and A. Yonezawa. "Analysis of the inheritance anomaly in concurrent object-oriented languages," extended abstract presented at the ECOOP-OOPSLA'90 Workshop on Object-based Concurrency, October, 1990.
20. R. Milner. Communication and Concurrency, Prentice-Hall, 1989.
21. O. Nierstrasz. "Active objects in hybrid," OOPSLA'87 Proceedings, pp. 243–253, 1987.
22. O. Nierstrasz and M. Papathomas. "Towards a type theory for active objects," in Object Management, (ed. D. Tsichritzis), pp. 295–304, Centre Universitaire D'Informatique, Université De Geneva, 1990.
23. M. Papathomas. "Concurrency issues in object-oriented languages," in Object Oriented Development, (ed. D. Tsichritzis), pp. 207–245, Centre Universitaire D'Informatique, Université De Geneva, 1989.
24. M. Papathomas. Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming, Ph.D. Dissertation, University of Geneva, 1992.
25. C. Tomlinson and V. Singh. "Inheritance and synchronization with enabled-sets," OOPSLA'89 Conference Proceedings, ACM Sigplan Notices, pp. 103–112, 1989.
26. A. Tripathi and M. Aksit. "Communication, scheduling and resource management in Sina," Journal of Object-Oriented Programming, 1(4), pp. 24–37, November/December 1988.
27. J. Van Den Bos and C. Laffra. "Procol: A parallel object language with protocols," ACM Sigplan Notices, Proceedings OOPSLA'89, 24(10), pp. 95–102, October, 1989.
28. T. Watanabe and A. Yonezawa. "Reflection in an object-oriented concurrent language," ACM Sigplan Notices, 23(11), pp. 306–315, 1988.
29. P. Wegner. "Dimensions of object-based language design," OOPSLA'87 Conference Proceedings, ACM Sigplan Notices, pp. 168–182, December 1987.
30. A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. "Modelling and programming in an object-oriented concurrent language ABCL/1," in Object-Oreinted Concurrent Programming, ed. M. Tokoro, pp. 55–89, MIT Press, 1987.

This article was processed using the LaTeX macro package with LLNCS style