Object-Oriented Design in Ada: A Transformational Approach Based on Object-Oriented Analysis

Rick Hill

Project Technology, Inc.
10940 Bigge Street
San Leandro, California 94577-1123
510 567-0255
http://www.projtech.com

June 1, 1990

This paper presents a systematic method for producing an Object-Oriented Design from an Object-Oriented Analysis (OOA) application model [9, 10]. The approach taken is consistent with the design philosophy of Recursive Design [11]: We adopt an object-oriented architecture and describe a technique for mapping components of a given application model to that architecture. Thus, rather than designing each implementation object independently, we focus on producing a set of clear and well-defined rules which are applied in the design of every implementation object. The transformational nature of this approach leverages the application analysis: The application model provides not only a mechanism for understanding a problem but also a framework for its solution [2].

We believe that any design strategy that has these properties offers significant benefits:

1) **Automation** - Significant portions of the target implementation can be mechanically or automatically generated.

2) **Traceability** - Since the components of the specific system design are mechanically derived, by reversing the derivation process, every design component can be traced back to its corresponding application component or components.

3) **Reusability** - Since the design strategy is stated in general terms, it can be used to produce the design for any number of application models, depending only on the performance requirements of the implementation.

4) **Integration** - The design strategy dictates integration policy for all design components. As a result, system integration is part of the design process and not an afterthought. Furthermore, that policy is applied system wide, greatly simplifying the integration task.

Using the design strategy described below, the transformation of an OOA application model into a object-oriented design is primarily a mechanical process. Very little other than the object actions requires hand coding. This has the effect of radically simplifying

Earlier form published in Proceedings of the Structured Development Forum XI, 1990.

the design process. The overall system design is derived by applying the design rules described below to each object in the application analysis according to its class; the result is a specific object-oriented system design.

The design rules are stated in terms of the OOA formalism and in no way depend on a specific problem domain. Changes to application models will not impact the design strategy and, conversely, changes in the design strategy will not affect application models. This strategy is only one of many based on OOA. The design rules can be customized or even drastically altered to accommodate the specific requirements of a given application. The architecture is stated in terms of Ada but can be implemented more or less directly in other computer languages as well.

In the following sections, the design strategy is described. After a brief overview of OOA, the fundamentals of the design strategy are presented, followed by four sections that describe specific rules for transforming an OOA model to Ada code. The rules are accompanied by diagrams illustrating the architecture for the basic kinds of analysis objects. We conclude with a discussion of the properties of the architecture.

1. Overview of Object-Oriented Analysis

This paper assumes a basic understanding of Object-Oriented Analysis. The following section summarizes the important concepts; subsequent sections discuss further details of OOA. However, readers not familiar with OOA may wish to refer to the references [9, 10] for a more detailed treatment.

An OOA model consists of three formal models: an information model, a set of state models, and a set of process models. OOA prescribes a set of integration rules for these models as well as an order for their construction:

> **Step 1**: Information Models. In this step, the conceptual entities of the problem are identified and formalized in terms of objects, attributes, and relationships.
>
> **Step 2**: State Models. State models are used to formalize the "lifecycles" or "life histories" of objects and relationships. The state models communicate by means of events and are organized in layers to make the system of communication clear. One state model is built for each object which has dynamic behavior.
>
> **Step 3**: Process Models. Every action on a state model is composed of a set of processes. These processes may operate on object attributes and generate events. The process models describe the flow of data between processes and data stores, and the sequencing of processes within actions.

2. Fundamental Design Strategy

A well-executed OOA application model provides a complete, consistent, concise, and unambiguous description of a problem. In the context of the design strategy described here, that model serves as a specification for a solution to the problem. By applying the design rules described below, a design conforming to that specification is generated.

In an OOA application model, every object appears in the Information Model. In addition, if an object has behavior (in the sense of a lifecycle), the behavior of the object is described by a State Model. There are two fundamentally different kinds of state models: those that describe instance behavior and those that describe the collective behavior of the object instances. In the former case, each instance has an associated copy of the state model that describes its behavior. In the latter case, instances have no independent behavior but their class behavior is described by a single copy of the state model. Each copy of a state model is called a "state machine".

The set of all OOA objects is partitioned into three classes based on the state model of the object:

> **Active Objects**: An active object has a state model, a copy of which is associated with each instance of the object.
>
> **Passive Objects**: A passive object is one which has no state model.
>
> **Monitor Objects**: A monitor object has a state model, a single copy of which is used to manage *all* instances of the object.

The design strategy specifies two types of rules: general rules that apply to all analysis objects and specific rules that apply to objects based on their membership in one of the above classes. The general rules of the strategy are described in the following section. The specific rules for each class are described in the three sections following the description of the general rules. Each design rule describes features of an arbitrary application model and corresponding design components to implement those features.

3. General Design Rules

The following design rules apply to all OOA objects. All designs for a given class of objects share a common set of features or a common architecture. The architecture of each class of objects is graphically illustrated in a diagram accompanying the specific rules for that class.

3.1 Object Rule

All objects in the application analysis – active, passive and monitor – are implemented as Ada packages. The name of the package reflects the name of the analysis object it implements.

3.2 Domain Type Rule

Every object attribute has an associated set of legal values: the attribute domain. The design strategy specifies that every object attribute is associated with an Ada type sufficient to represent the domain of that attribute and support basic operations on it. That type is called the "domain type" for the attribute.

3.3 Instance Rule

An OOA object may have multiple instances. The data component of each instance is represented by an instance of a record type called Instance_Type. This type is encapsulated by the package implementing the object. Each attribute of the object is represented by a field of the instance type. The name of each field reflects the name of the attribute it represents. The type of the field representing the attribute is the domain type for that attribute.

3.4 Instance Collection Rule

The data aspect of an OOA object can be viewed as a table. Each column corresponds to a unique object attribute. Each row contains a unique object instance. The intersection of a row and a column contains the value of the attribute associated with the column for the instance contained in the row.

Every package implementing an analysis object encapsulates a package structure called Instance_Table. The instance table provides for storage and associative retrieval of data records representing object instances. For every instance of a given object, there is one unique corresponding record in the instance table for that object. The instance table provides the following basic operations:

1. insert - insert a record
2. retrieve - retrieve a record
3. remove - remove a record
4. update - update the non-key fields of a record
5. record_exists - test for the existence of a record
6. empty - test for an empty table
7. clear - delete all records
8. number_of_records - return the number of records in the table

The instance table may also export a generic package for instantiating active iterators [3].

A detailed specification for the instance table is beyond the scope of this paper. However, the specification should describe the interface to the table in detail while abstracting from the details of the table implementation. Different specific implementations can then be chosen on an object-by-object basis. For a given object, the change from one table implementation to another would have minimal impact on other parts of the object implementation.

3.5 Concurrency Rule

In OOA, state machines operate concurrently. The design strategy represents this concurrent execution by using a single action executive task (described in detail below) for each active or monitor object. As a result, any operation exported from a package implementing an object may be invoked by concurrently-executing Ada tasks. To support this concurrent implementation [4] of an object, every package implementing an object encapsulates a semaphore called Lock. Lock is an instance of the task type Semaphore [5] and exports two entries: Request and Release.

3.6 Atomic Read/Write Rule

Reads and writes to data stores are atomic in OOA process models. Representing this fact, each of the operations described in sections 3.7 through 3.12 below calls Lock.Request immediately before and Lock.Release immediately following access to the instance table. From the design perspective, this simple approach ensures the integrity of the instance table**.** We assume the instance table uses a sequential implementation [4].

3.7 Read Accessor Rule

A process in an OOA model may directly read values from object instances without generating events. To read an instance, one must identify the instance to be read. In OOA, every object has at least one identifier which consists of one or more attributes that uniquely identify an instance of the object. An object may have many such identifiers. An object instance is identified by specifying a set of identifying attribute name/value pairs. For example, a model may refer to "some power supply with Manufacturer = C. M. Wang's Avionics Supply".

For every identifier used to read an instance of an object, the package implementing that object defines a read access procedure or "read accessor". The arguments to the procedure compose a key value identifying a record in the instance table of the object. The read access procedure then fetches the identified record and returns all attribute values through OUT parameters. The procedure uses the retrieve operation of the instance table to read the indicated instance.

3.8 Write Accessor Rule

A process in an OOA model may also directly write values to object instances without generating events. To write a value to an object instance, one must identify an instance and specify the attribute to be written and the value that attribute is to be given.

For every identifier used to write an attribute of an object, the package implementing that object defines a write access procedure or "write accessor". The parameters to the procedure include the attribute value to be written and a key value identifying a record in the instance table. The name of the write procedure reflects the name of the attribute being written. The procedure uses the update operation of the instance table to write the indicated value.

3.9 Existence Test Rule

A process in an OOA model may also query the existence of object instances. To query the existence of an instance, that instance must be identified.

For every identifier used to determine the existence of instances of an object, the package implementing that object defines a Boolean existence test function. The function uses the record_exists operation of the instance table.

3.10 Set Operation Rule

An OOA model may specify operations on sets of object instances. For example, a process may specify "Find the disk request with the earliest arrival time."

The design strategy supports operations on sets of instances by means of a generic active iterator package. Every package implementing an object whose instances are operated on in this mannner defines a generic iterator package. Packages requiring set operations may instantiate this package to yield an iterator instance. Figure 1 shows an instance of the generic iterator package as the exported procedure Iterate_p.

3.11 Instance Creation Rule

An OOA model may specify creation of object instances. If creation of instances of an object is specified, the package implementing that object will define a single procedure called Create. The arguments to this procedure include a value for every attribute composing an identifier of the object. Create calls the insert operation on the instance table to create a record representing the added instance.

3.12 Instance Deletion Rule

Similarly, an OOA model may specify deletion of object instances. For every identifier used to delete an instance of an object, the package implementing that object defines an instance deletion procedure. The arguments to this procedure include a key value identifying the target instance. Each deletion procedure calls the remove operation on the instance table to remove the record representing the target instance.

3.13 Operator Visibility Rule

As described in sections 3.7 through 3.12, OOA processes may specify various operations on object instances. When an object performs such operations on the instances of other objects, those operations are called "external" operations on that object. When an object performs such an operation on its own instances and no other object requests the operation, that operation is called an "internal" operation. An operation is either an internal or an external operation.

Each procedure, function, or package implementing an external operation on an object is exported by the package implementing that object. Constructs implementing only internal operations on an object are encapsulated by the package implementing that object.

3.14 Relationship Rule

In OOA, relationships are represented by referential attributes. The record representing an object instance may identify related instances through the referential key values it contains. The setting and updating of those key values is handled by the action procedures described below and directly reflects behavior specified by the application model. Multiplicity and conditionality constraints specified by the Information Model are also maintained by the action procedures.

4. Specific Design Rules for Active Objects

Figure 1 illustrates the architecture for active objects. The diagram is in the form of an Ada Structure Graph, or Buhr diagram [6]. The architectural components are described below.

4.1 Event Queue Rule

In OOA, state machines may communicate by means of events. That is, one state machine may generate an event to which another state machine responds. When an event is generated it takes a finite and indeterminate amount of time to reach the target state machine. Once it arrives, the event is "queued" on the target state machine.

Every package implementing an active object encapsulates an event queue (called Event_Queue) used to hold the names of events that have been generated and to which a response is pending.

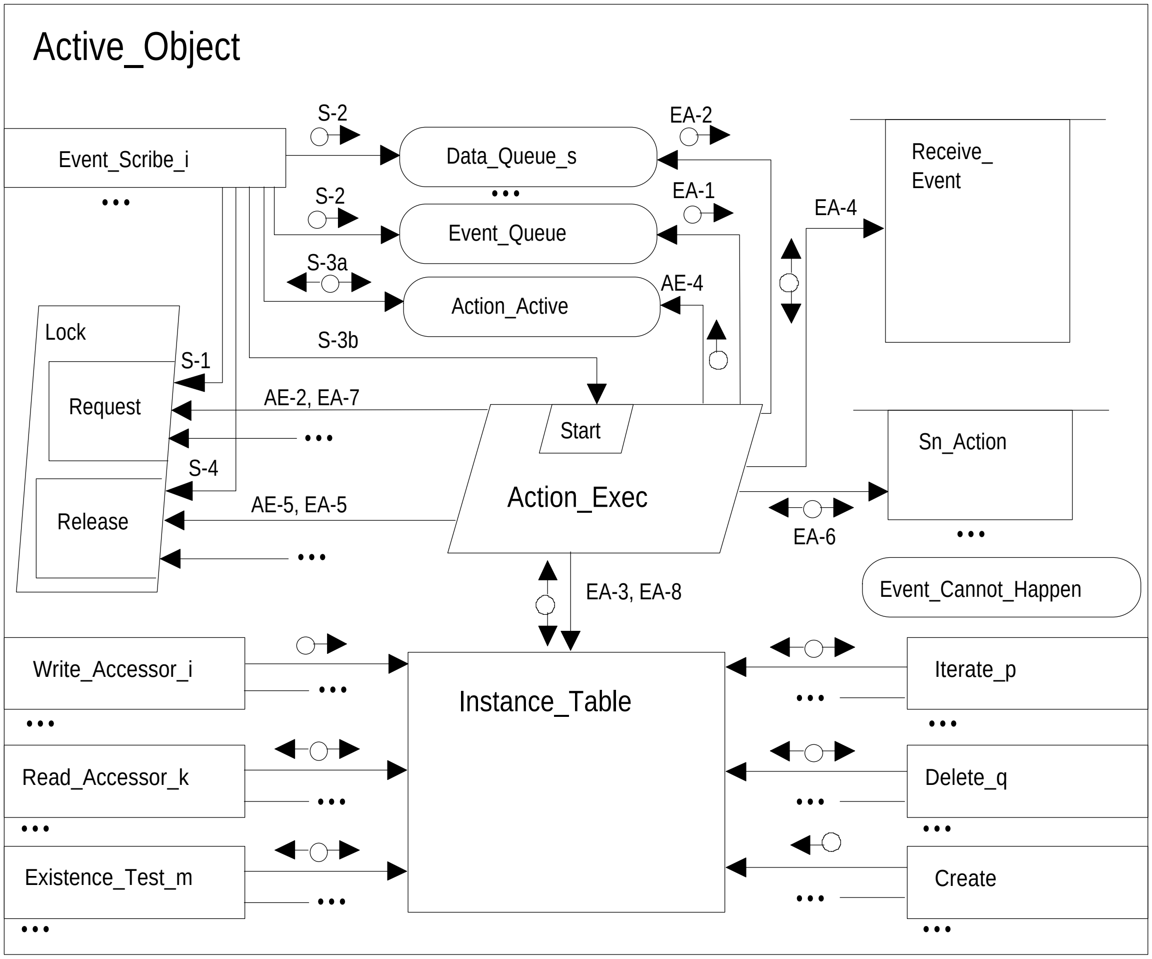

Figure 1: Architecture for Active Objects. Those structures having multiple possible occurrences are followed by ellipses. The lock.request and lock.release invocations are shown in a reduced form for greater clarity. Also, arrows impinging on the instance table illustrate the invocation of instance table operations.

4.2 Event Data Rule

Data carried by an OOA event is called event data. Each datum associated with an event is an attribute value for some object instance. Thus, by the Domain Type Rule, every event is associated with a multiset [7] of Ada types. (A multiset is a set that allows multiple occurrences of the same element.) Note that the multiset of types associated with one event may be identical to that of another event.

Every package implementing an active object will encapsulate one event data queue for every such unique multiset of types characterizing an event received by the object. We assume that a single multiset of Ada types characterizes all events leading to a given state in an OOA state model.

Each event data queue is a queue of records where each record represents the event data accompanying a single event. Each event data queue is named Data_Queue_s, where s is some string characterizing the queue. For example, a queue holding data accompanying events E1 and E2 might be called Data_Queue_for_E1_and_E2.

4.3 Event Scribe Rule

As mentioned previously, when an event is generated it takes a finite and indeterminate amount of time to reach a target state machine. Conceptually, once the event arrives, the event name and the data it carries are queued (FIFO) for the target state machine. Thus a record or transcript is made of every event when that event reaches its target.

As such, every package implementing an active object exports a set of event transcription procedures or "event scribes". One event scribe is exported for every event labeling a transition on the object state model. The name of the event scribe reflects the name of the event. The parameters of an event scribe carry the event data. The names of those parameters reflect the names of the attributes that provide the event data.

Event scribes are reentrant and may be called concurrently by any number of clients. A call to an event scribe simply sets up a response in the target object. This reflects the fact that in OOA, the semantics of event generation include no notion of either control transfer or return of computational results.

Every object has an action executive (described below). It is the responsibility of the action executive of the target object to execute a response to an event once that event is queued. An action executive will block when it depletes its event queue. The package variable Action_Active reflects the status of the action executive. From the perspective of an event scribe, Action_Active is false if and only if the action executive is blocked.

On Figure 1, the invocations labeled S-1, S-2, and so on illustrate the steps of an event scribe:

S-1. Request a lock on the object (i.e., call Lock.Request) to reserve exclusive access to the package data structures (e.g., the event and data queues).
S-2. Enqueue the event name on the event queue and a record containing the event data on the data queue.
S-3. If (and only if) Action_Active is false (the local action executive is blocked) then
  a) set Action_Active to true and
  b) call the Start entry on the local action executive.
S-4. Release the lock.

When an event scribe parameter shares a type with the event scribe parameter of another package, that type must be visible *from the spec level* of both packages. The impact of visibility requirements of this sort can be minimized using shared types packages. Other cases of required spec-level visibility are rare. Therefore, under this design strategy, visibility cycles are easily avoided and constraints on the compilation-order of the target system are few.

4.4 Internal/External Event Rule

There are two kinds of events in OOA: internal and external. External events are generated from outside some responding object, and internal events are generated from within the responding object. The event scribe for an external event is exported by its defining package. The event scribe for an internal event is not exported. Otherwise, internal and external events are implemented identically.

4.5 Event Generation Rule

The generation of an event is represented by calls to the event scribes of every object with at least one state transition labeled with the name of that event. The order of the calls is arbitrary. All packages implementing objects that respond to an event must be visible *from the body level* of any package that generates that event. Thus, each package representing an object that receives an event is *with*ed from the body of the package implementing the object that generates that event.

4.6 State Transition Table Rule

The state transition table for an object is implemented by a two dimensional matrix called the State Event Matrix (SEM). The SEM is indexed in one dimension by the state names labeling rows of the corresponding state transition table. The SEM is indexed in the other dimension by the event names labeling columns of the state transition table. Elements of the SEM may contain the names of states in the represented state transition table or the special symbols IGNORE or CANNOT_HAPPEN. The value of a given element directly reflects the value of the corresponding element of the state transition table of the analysis object.

4.7 Identifier Data Rule

Event data for an active object must include an identifier value. Consequently, when an event scribe of an active object is called, one or more parameters of the procedure will compose a key identifying a record in the instance table.

4.8 State Rule

Every active object has a state attribute. That state attribute is represented as a field of the instance type. It follows that every instance of an active object has a state characteristic and the instance table records the state characteristic for every instance of an object.

4.9 State Transition Rule

Execution of a state transition for a given instance of an active object is implemented by a procedure named Receive_Event. Receive_Event is encapsulated by every package implementing an active object and encapsulates the SEM for that object. It has three parameters: a state parameter that describes the state of an object instance, an event parameter that names an event sent to that instance, and a status code. Using the supplied state and event, Receive_Event fetches a value from the SEM. If the value read is a state name, the procedure signals a transition to that state by updating the state parameter and returns a status code of MOVED. If the value read is IGNORE, the procedure returns a status code of EVENT_IGNORED. If the value read is CANNOT_HAPPEN, the procedure returns a status code of EVENT_CANNOT_HAPPEN.

4.10 Action Rule

Each action on the state model of an active object is implemented by an Ada procedure called an "action procedure" and encapsulated by the package implementing that object.

The precondition and postcondition [1, 8] of that procedure are identical to the precondition and postcondition of the corresponding action in the application model.

The package implementing an object may define or import utility subprograms in support of its action procedures. These procedures are called action utilities. To protect the integrity of package data structures, the design strategy requires that an action procedure call only action utilities and those subprograms exported by the package in which it is defined. Action utilities are never exported by a package implementing an object and are also subject to this constraint. For example, an action procedure may not call the request entry on the lock task. However, it may call an event scribe or any of the procedures and functions described in sections 3.7 through 3.12.

4.11 Action Execution Rule

In OOA, when a state machine reaches the end of an action, it removes the first event on its event queue and responds to that event. The event response entails two steps. The first step is to attempt a transition. If some transition out of the current state of the state machine bears the name of the event, then a transition occurs. Otherwise, the event is ignored and another event is dequeued. Given that a transition occurs, the second step of the event response is to execute the action of the new current state of the state machine. When the event queue is exhausted, the state machine will remain idle until a new event is queued. The arrival of a new event will cause an idle state machine to remove the first item in the queue and carry on as described above.

The action executive of an object is a task (called Action_Exec) responsible for executing event responses for every state machine of that object (one machine exists for each instance of the object). The execution of those event responses is interleaved within the task. While the event queue contains an item, the action executive does the following: It dequeues the first event name on that queue. Then, depending on the event name, it dequeues the accompanying event data from the appropriate data queue. The action executive then attempts to retrieve the record representing the target instance from the instance table. Assuming the target instance is found, the action executive calls Receive_Event to attempt a transition on the state of that instance. Assuming a transition occurs, the action executive calls the action procedure associated with the current state of the instance. The arguments to the action procedure include the record dequeued from the data queue (an IN parameter) and a record representing the target instance (an IN/OUT parameter). Upon termination of the action procedure, the action executive updates the instance table to reflect the new attribute values of the target instance.

The action executive locks the object while manipulating the queues and the instance table but releases that lock while executing the action procedure. The action executive executes event responses as long as items are on the event queue. Once that queue is exhausted, the action executive blocks. Event scribes may call Action_Exec.Start to awaken the action executive.

On Figure 1, the invocations labeled AE-1, AE-1, and so on illustrate the steps of the outer control loop of the action executive. (AE-1 and AE-3 are internal to the action executive and are not shown.) The executive encapsulates the procedure Execute_Action (described below). The package variable Action_Active is initially false.

Repeat forever:

AE-1. Execute an accept on the Start entry. (The accept has no do ... end clause and serves only to block the action executive until an event is on the event queue and the Start entry is called by an event scribe.)

AE-2. Request a lock on the object.

AE-3. While the event queue contains an item, call Execute_Action.

AE-4. Set Action_Active to false.

AE-5. Release the lock.

On Figure 1, the invocations labeled EA-1, EA-2, and so on illustrate the steps of the Execute_Action procedure called by the action executive:

EA-1. Dequeue the head of the event queue and let Event Name be that value.

EA-2. Dequeue the head of the appropriate data queue. Let Event Data be this record. By the Identifier Data Rule, Event Data will include a key value identifying an instance of the object.

EA-3. Use the key value provided in Event Data to retrieve a record from the instance table. Let Identified Instance be the value of that record excluding the state field.

EA-4. Call Receive_Event with Event Name and the state field of Identified Instance as arguments. If the resulting value of the status parameter is:

a) Ignore - Return from Execute_Action.

b) Cannot_Happen - Generate the exception Event_Cannot_Happen.

c) Moved - Let New State be the resulting value of the state parameter of Receive_Event.

EA-5. Release the lock.

EA-6. Call the action procedure named by New State. The parameters will include Event Data (as an IN parameter) and Identified Instance (as an IN/OUT parameter).

EA-7. Request the lock.

EA-8. Update the identified instance in the instance table to reflect the values of New State and Identified Instance.

5. Specific Design Rules for Passive Objects

Passive objects have no dynamic behavior. As such, they do not require scribes, event queues, action executives, or any other structure described by the specific rules for active and monitor objects. The general design rules are sufficient to describe passive objects.

The Buhr diagram in Figure 2 illustrates the architecture for passive objects.

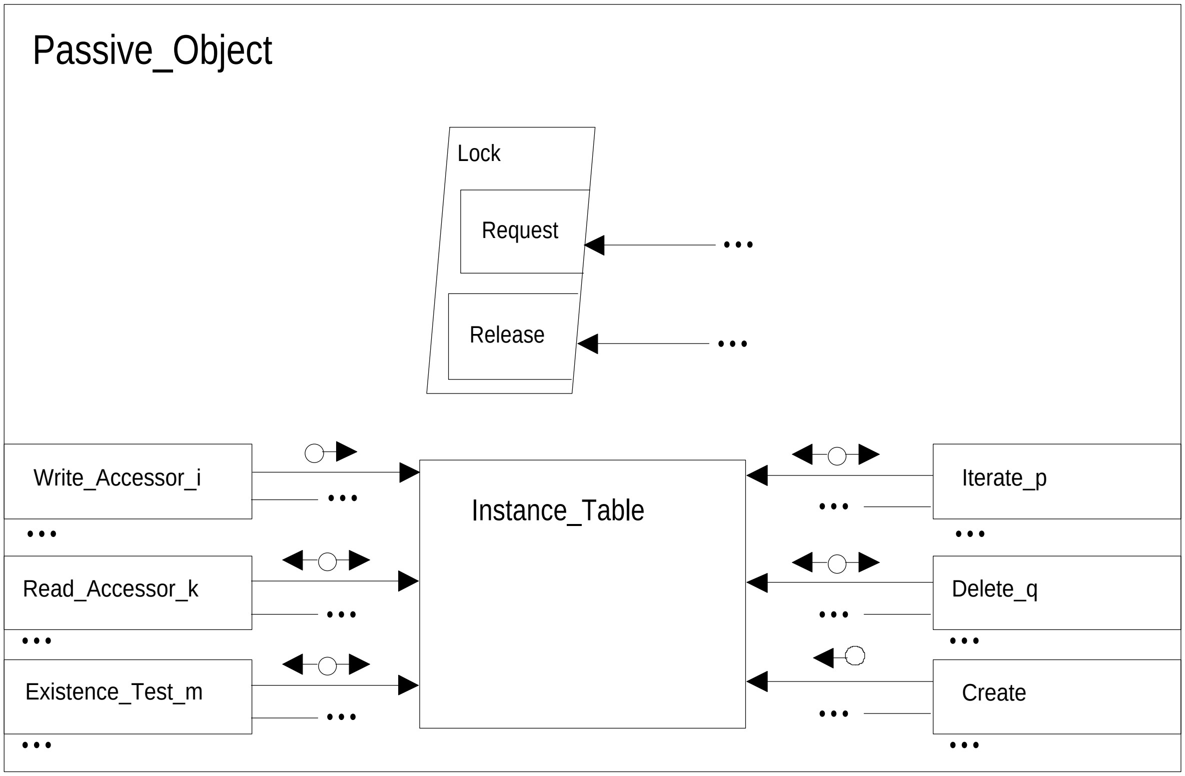

Figure 2: Architecture for Passive Objects

6. Specific Design Rules for Monitor Objects

With the following exceptions, the design rules for active objects also apply to monitor objects. The state of a monitor object is represented by *one* state variable encapsulated by the package implementing an object. Therefore, state is not represented in the instance type or instance table and the State Rule must be revised for monitor objects accordingly.

Since the actions of monitor objects are written in terms of the set of all instances of the object, no single instance is necessarily targeted by an event. Reflecting this fact, the Identifier Data Rule is not required for monitor objects and the Action Execution rule must be altered for monitor objects.

The Buhr diagram in Figure 3 illustrates the architecture for monitor objects.

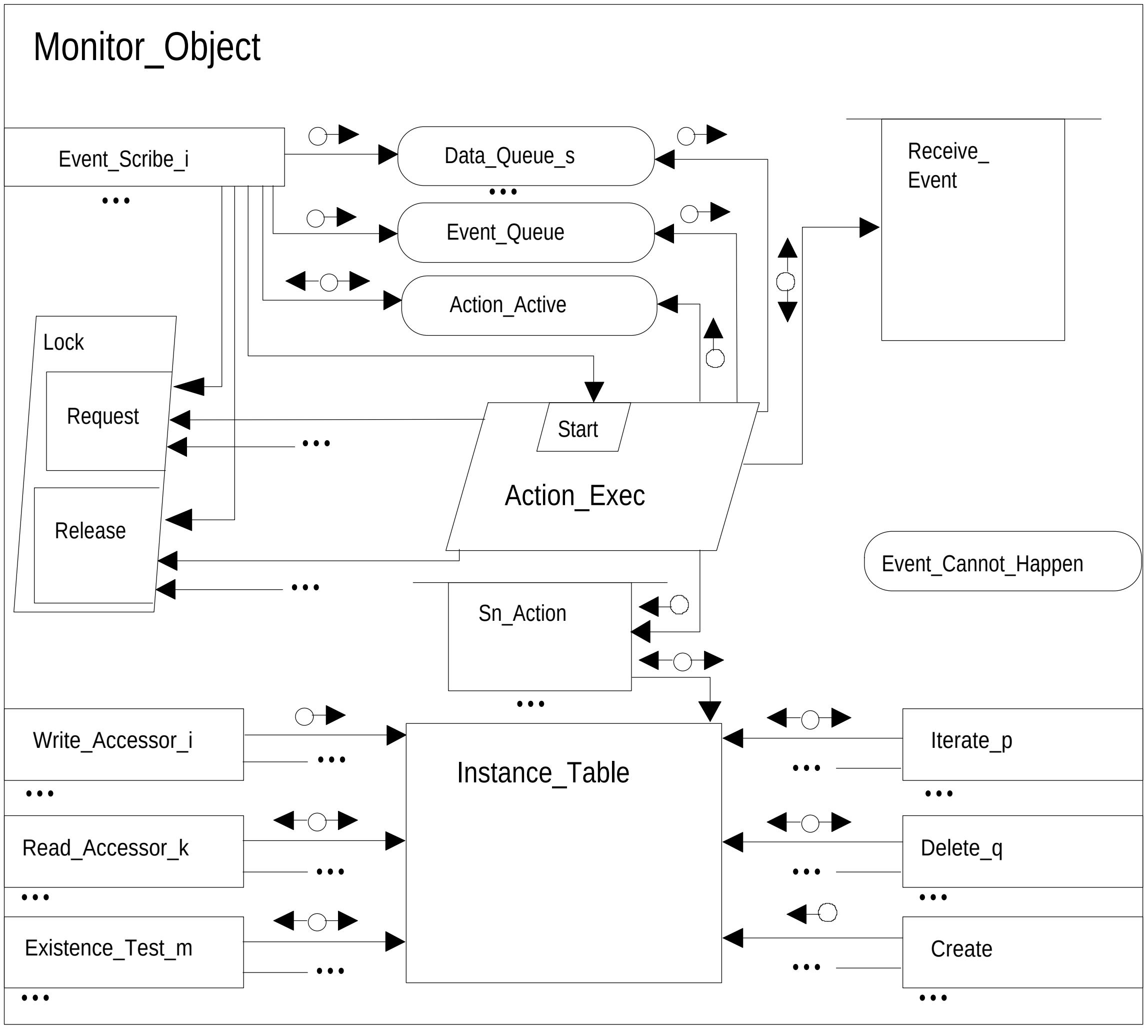

Figure 3: Architecture for Monitor Objects

7. Properties of the Architecture

The details of the architecture illustrated in figures 1, 2, and 3 are described in the preceding body of design rules. The architecture is described in terms of Ada, but may be implemented in other languages as well. For example, rather than using an Ada package to implement an object, we could have used a C++ class. However, notice that the choice of language is much less significant than the choice of architecture. It is the architecture that determines the policies for data storage and processing. The language *must* allow and *may* enforce implementation of those policies. For example, it is convenient if encapsulation of the event queue is enforced at compile time. That encapsulation is specified by the architecture and is necessary for any implementation of the architecture. However, if we implement the architecture in assembler, we may establish conventions that yield the required data encapsulation. We may enforce those conventions in code reviews.

It is possible to talk about properties of this architecture apart from its specific implementation. For example, objects encapsulate all their data, communication between objects is asynchronous, and so on. However, in order to make very detailed statements about this architecture, we must assume an implementation. For example, if the architecture is implemented in Ada as described above, we can state with certainty that there will be at least two procedure calls and two rendezvous associated with generating an event. In the following paragraphs, we note various properties of the architecture in the context of the Ada implementation described above.

The architecture takes a very strict view of data encapsulation. Objects encapsulate all their data and export only those procedures required for messages and instance operations. Actions directly affect only local data. Instances are never directly updated, but must be accessed through the operations exported by the instance table. The benefits of data encapsulation are well-known. For this architecture, encapsulation minimizes coupling between objects and between object components.

Notice that the design rules describe only the interface for the instance table. For objects with persistent data, the instance table may simply interface with a commercial relational data base. Hashing schemes offer the possibility of constant-time lookup; however, if the table contents change much over time, the hashing function must be maintained to prevent degradation. For objects with static populations requiring fast access, a hash table may be appropriate. Alternately, binary search trees can be used to implement relatively fast and low maintenance associative indexing.

Since objects execute simultaneously, the architecture must not only restrict what may be accessed, but also when an access may occur. Thus, for each object, a semaphore enforces mutually exclusive access of all the object data structures. The Concurrency Rule (3.5) suggests implementing this semaphore using an Ada task. This approach takes advantage of the Ada facilities for task synchronization and is simple, portable, and sufficient to implement the desired mutual exclusion; however, there may be efficiency considerations. If timing deadlines are tight, and the overhead associated with the rendezvous for the target compiler is relatively large, then another approach (or another compiler) may be required.

The concurrent execution of state machines in OOA is represented in the architecture using two mechanisms. For simplicity and efficiency, concurrent execution of state machines within an object is implemented by executing one action at a time in response to events on the event queue regardless of the instance involved. Concurrent execution between objects is implemented using a task for every action executive. For a system with twenty active objects, the resulting implementation will include at least forty tasks. This number is frightening in the context of traditional development methods. However, notice that the use of tasking in this architecture is strictly regulated according to a well-defined set of rules. The tasking strategy has been carefully crafted to avoid deadlock and starvation. An implementation built using this strategy will exhibit deadlock if and only if the application analysis exhibits deadlock. Assuming the scheduling algorithm for the target compiler is fair, the same claim holds for starvation problems.

Most of the overhead in multitasking comes from context switching. However, the maximum number of *active* state machines for a given application model is typically quite small: around two or three. In terms of implementation, this means that at a given time at most a handful of tasks will be active. Usually, only a single task is active and the overhead associated with context switching is not an issue. Of course, some application models do incorporate significant concurrent processing. If context switching overhead is a concern, the analyst must determine what part of the concurrency is necessary and what can be sequentialized. The architecture described can then be modified to yield sequential forms of both the active and monitor objects. The sequential forms require no lock and use a procedure rather than a task to implement the action executive.

8. Acknowledgements

Much of this work was performed in the Payload Test Development Division at Sandia National Laboratories where Jeff Lenberg and Karen Weber provided invaluable critical support and encouragement. At Project Technology, Sally Shlaer, Steve Mellor, Klancy deNevers, and Wayne Hywari have provided numerous significant insights regarding both the technical content and presentation of this paper.

9. References

[1] Roland C. Backhouse, Program Construction and Verification, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[2] Robert Balzer, Thomas E. Cheatham, Cordell Green, *Software Technology in the 1990's: Using a New Paradigm*, IEEE Computer, November 1983.

[3] Grady Booch, *Software Components with Ada*, Benjamin/Cummings, Menlo Park, CA, 1987, pp. 157-159.

[4] Ibid, p. 41.

[5] Grady Booch, *Software Engineering with Ada*, Benjamin/Cummings, Menlo Park, CA, 1987, pp. 306-307.

[6] R. J. A. Buhr, *System Design with Ada*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[7] Sartaj Sahni, *Concepts in Discrete Mathematics*, Camelot, Fridley, Minnesota, 1986, p. 129.

[8] Ibid, pp. 93-118.

[9] Sally Shlaer and Stephen J. Mellor, *Object-Oriented Systems Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[10] Sally Shlaer and Stephen J. Mellor, *An Object-Oriented Approach to Domain Analysis*, Software Engineering Notes, ACM Press, July 1989.

[11] Sally Shlaer and Stephen J. Mellor, *Recursive Design*, Computer Language, March 1990.