A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis

Jose L. Fernandez

December 1993

A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis
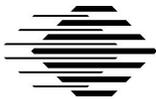
Jose L. Fernandez

Application of Software Models Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

Table of Contents

List of Figures

Acknowledgments

This work was performed during my stay at the Software Engineering Institute as a resident affiliate working in the Applications of Software Models Project.

Sholom Cohen, responsible of the project, contributed valuable advice to the use of the Features Oriented Domain Analysis in this report.

The comments and suggestions of the following reviewers were also considered in the report.

Jorge Diaz-Herrera	(SEI)
Patrick Donohoe	(SEI)
Francisco Gomez	(Tour and Andersson)
Robert Krut	(SEI)
Juan A. de la Puente	(Universidad Politecnica de Madrid)
Ragunatham Rajkumar	(SEI)
Lui Sha	(SEI)
James Withey	(SEI)

A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis

Abstract: A taxonomy of the coordination mechanisms for the synchronization and communication of concurrent processes is proposed. The taxonomy deals with the issues of a real-time software architecture that are application domain independent. The taxonomy will help the designer to find the appropriate coordination mechanisms for building a real-time domain specific software architecture.

Features Oriented Domain Analysis methodology has been used to describe the taxonomy. While Ada is the programming language that has been used here, some of the attributes and guidelines are still valid for other programming languages.

1　Introduction

1.1　Problem Statement

In the design of a real-time system, the identification, synchronization, and communication of concurrent activities is a major issue. Frequently software architectures are designed based only on functional or object-oriented decomposition criteria, postponing to the coding phase decisions about processes, synchronization, and communication.[1] While such an approach may be appropriate for those systems where timing constraints are not the issue, this is not the case in real-time applications.

Time is a critical factor in a real-time software architecture, just as load stress is critical in a building structure. Some apparently well-designed bridges and buildings have failed because of miscalculating loads or the bad design of the structure connections [Petroski 92].

It is important to find engineering principles that will help software engineers build reliable real-time software architectures that will meet timing constraints. Much research has been done on scheduling issues [Xu 93],[Sha 89], but engineers also need guidance in the selection of the coordination mechanisms for use between processes, just as civil engineers must select structure connections suitable for their needs.

A domain analysis of the services provided by real-time executives and extended Ada runtime libraries allows us to understand and categorize the coordination mechanisms into a taxonomy. The taxonomy is related to the Ada programming language, but it is not constrained by

---

[1]. Synchronization is the blocking of a process until some specified condition is met. Communication is the transfer of information between processes. Coordination encompasses both properties.

the Ada rendezvous communication mechanism. This is because although practical applications do use Ada because of Ada's modularization constructs (i.e. packages, generics), they do not use Ada rendezvous. In its place, the services of a real-time executive are used.

While real-time executives may improve efficiency, they may also result in reduced portability and reduced reusability. Standardization efforts, some of them mentioned in this report, would diminish these drawbacks.

The new version of Ada, Ada 9X, gives to the software engineer options not available in Ada 83. Ada 9X protected types offer the option of building coordination mechanisms that are efficient, reusable, and portable. Barnes, for example, describes a bounded-buffer using a protected type [Barnes 93]. More work remains to be done in this area.

The above concerns lead us to consider both kinds of mechanisms in the domain analysis: those based on pure Ada features and those based on real-time executives that can be used with Ada.

1.2   Scope of the Work

Design issues concerning coordination mechanisms can be organized in a taxonomy which describes their properties as decision choices. Many methods for organizing collections of soft ware components have been proposed, including traditional library and information science methods and knowledge-based methods.

Our approach is to develop a taxonomy of the coordination mechanisms based on the outputs of a domain analysis process. Domain analysis is the key to systematic, formal, and effective reuse. It is through domain analysis that knowledge is transformed into generic specifications, designs, and architectures.[2]

Although the scope of this work is not as broad as a typical software reuse effort [Cohen 92], domain analysis offers a systematic and structural way for classifying the features[3] of the coordination mechanisms.

Feature Oriented Domain Analysis (FODA) [Kang 90] is the domain analysis method chosen in this work. FODA is based on identifying features of a class of systems. The FODA process defines three basic activities:

> • *Context Analysis.* The purpose of context analysis is to define the scope of the domain that is likely to yield exploitable domain products.

[2.]   R. Prieto-Diaz. Status Report: Software Reusability. IEEE Software, May 1993.

[3.]   Features are the attributes of a system that directly affects end users [Kang 90].

- *Domain Modelling.* Commonalities and differences of the problems that are addressed by the applications in the domain are analyzed and a number of models representing different aspects of the problems are produced (features model, entity-relationship-attribute model, data flow model, and finite-state machine model).
- *Architecture Modelling.* The purpose of architecture modelling is to provide a software solution to the problems defined in the domain modelling phase.

In our particular situation, we are not performing a full domain analysis of a family of systems; our goal is less ambitious. We are trying to classify consistently the coordination mechanisms to be used for Ada real-time applications.

The FODA domain modelling process and particularly its features model output provides the taxonomy user with a structure for representing the attributes of the coordination mechanisms. FODA does this by decomposing features in a hierarchy and classifying them as either mandatory, alternative, or optional.

The contents of the report reflects the approach followed in the domain analysis of the coordination mechanisms.

Chapter 2 reviews the coordination mechanisms used in Ada applications. Coordination mechanisms are described in terms of three aspects: abstraction supported, services provided, and implementation. Chapter 2 is recommended for those readers not familiar with the standardization efforts of ARTEWG [ARTEWG 93] or RRG [RRG 93].

Chapter 3 describes the main properties that characterize a coordination mechanism. This description is a prerequisite step in the classification of these properties in the features model.

Chapter 4 represents the FODA features model for coordination mechanisms attributes. FODA features model notation is also explained.

Chapter 5 gives a set of rules identifying the key choices made in selecting the coordination mechanism best suited to a particular problem. The rules are formulated relating the problem requirements with the taxonomy of coordination mechanisms proposed.

Appendix A represents the taxonomy of the coordination mechanisms analyzed. The features corresponding to a particular coordination mechanism are represented.

Appendix B describes the catalog of the coordination mechanisms analyzed. The information in this appendix is important in determining which coordination mechanisms are Ada-supported and which use the services of a real-time operating system.

1.3 Related Work

Several taxonomies, which consider different coordination mechanisms and their intrinsic properties, have been developed earlier [Bloom 79], [Ripps 89], [Chi 91]. The following three approaches will be the focus of this section:

1. *Evaluation of Synchronization* Mechanisms. A set of categories is proposed by Bloom to evaluate the modularity, the expressive power, and the ease of use of a synchronization mechanism.

2. *Taxonomy of Coordination Methods.* A taxonomy of coordination methods supported by a real-time operating system is presented by Ripps. The classification of the coordination methods rests upon a set of dichotomies.

3. *Ada Task Taxonomy.* A taxonomy of mechanisms implemented in Ada is proposed by Chi. The classification is based on task interactive characteristics, functions, and procedural behavior.

The following sections describe each of these approaches.

1.3.1 Evaluation of Synchronization Mechanisms

Bloom [Bloom 79] determines that synchronization mechanisms are composed of exclusion and priority constraints, where each has the form:

if condition then exclude process A

        or

if condition then process A has priority over process B

Within these two main classes, constraints differ mainly in the kinds of information referred to in these conditional clauses. The information falls in several categories:

1. *Access Operation Requested.* Exclusion or prioritization are based on the type of operation requested.

2. *Arrival Times of Requests.* Frequently the relative time of arrival of the requests determines their order.

3. *Request Parameters.* In some cases the arguments are used to evaluate the ordering of the requests.

4. *Synchronization State of the Resource.* This includes all state information needed only for synchronization purposes.

5. *Local State of the Resource.* All state information as it relates to the abstraction represented by the resource is included here.

6. *History Information.* Information concerning complete operations or events regarding the resource is considered.

The methodology described is used to evaluate three synchronization mechanisms: path expressions, monitors, and serializers.

1.3.2   Classification of Coordination Methods

Ripps proposed a classification of coordination methods[4] to be used in C applications. The methods are supported by an operating system [Ripps 89]. An Ada applications version of this operating system already exists [Industrial 88].

The coordination methods are analyzed and classified according to the following dichotomies:

1. *Double-Sided versus Single-Sided.* Double-sided methods are symmetrical: whichever process gets to the coordination point first waits for its partner. In contrast, single-sided methods are asymmetrical: one process coordinates with another but not vice-versa.

2. *Directed versus Non-Directed.* In directed methods the identity of the target process must be specifically known to the coordinator. In non-directed methods, the identity of the target process is hidden.

3. *Single-Enabling versus Multiply-Enabling.* Single-enabling methods allow any one partner process to proceed at a time when more than one is waiting for coordination. Multiply-enabling methods release all partner processes that satisfy the coordination condition.

4. *Stored versus Transient.* For stored methods, the operating system retains the coordination information until it is needed by those methods with storing capability. With transient methods, information is not latched so it is lost if the target process is not waiting.

5. *Unidirectional versus Bidirectional.* The information flow may be in either one or two directions.

These dichotomies are applied in classifying diverse coordination mechanisms including event-oriented mechanisms, controlled-shared variables, semaphores, mailboxes, and Ada rendezvous.

1.3.3   Ada Task Taxonomy

A taxonomy encapsulating nine kinds of tasks implemented in Ada is proposed [Chi 91]. The taxonomy not only includes coordination mechanisms but also other roles: manager task, agent task, secretary task, scheduler, and input/output. The coordination mechanisms included are buffers, relays, and transporters.

---

4.   The concept of a coordination method described by Ripps and the coordination mechanism used in this report are equivalent.

Tasks are classified according to their interactive characteristics, the relationships between them, and the types of entries that characterize the task kind.

Interactive task characteristics describe the interaction patterns between the coordinating partners. Namely:

- One- and two-way interactions
- Degrees of interaction
- Master to task calling
- Task to task calling

Task relationships are of three major types: basic relations, cohesive relations, and coupling relations.

Chi Tau Lai [Chi91] describes seven possible interactive types of entries that characterize a task. These entries are:

- Get data.
- Give data.
- Open guard.
- Close guard.
- Add event.
- Get event.
- Do it.

A set of tools based on this taxonomy is proposed. The tools being developed are taxonomy-based editors, reverse engineering tools, and design aid tools using design and check rules.

1.3.4   Relationship to Previous Research

The previous works establish the foundation for the domain analysis activity described in this report.

The work by Bloom [Bloom 79] defines the main concepts regarding the synchronization properties of a coordination mechanism. Communication properties are not described adequately. The mechanisms analyzed are less sophisticated than the mechanisms available today.

The classification proposed by Ripps [Ripps 89] includes the mechanisms supported by a real-time operating system. The attributes are represented as dichotomies, however, and are not structured hierarchically.

The taxonomy proposed by Chi Tau Lai [Chi 91] not only includes coordination mechanisms but also other roles: manager task, agent task, scheduler, and input/output. The coordination mechanisms classified are Ada-based. For this reason, synchronization properties are not fully covered.

The FODA domain analysis methodology applied in this report allows us to represent hierarchically and completely the attributes describing a coordination mechanism. The variety of coordination mechanisms analyzed is not found in previous research. Coordination mechanisms analyzed are either entirely Ada-based, based on an extended solution using a common set of user- accessible runtime environment interfaces, or use the services provided by a real-time operating system. The currently standardization efforts allow us to consider this variety of approaches.

2 Review of the Coordination Mechanisms Used in Ada Real-Time Systems

Before describing the different coordination mechanisms that can be used in real-time applications developed in Ada, it is important to understand the Ada runtime environment and the differences between Ada targeted for a real-time executive or Ada targeted to a bare machine.

2.1 The Ada Development Alternatives

The diversity and importance of the implementation features (e.g., tasking, exceptions and interrupts) handled in real-time applications developed in Ada require a development environment more complex than other programming languages.

The Ada Runtime Environment (RTE) consists of abstract data structures, code sequences, and predefined subroutines that are used by the compilation system to produce a translated program.

The set of predefined routines included in the loaded image of a program is called the Runtime System (RTS). The set of routines available for inclusion in a RTS is called the Runtime Library (RTL).

The major functionalities of the Ada runtime system are:

- Task scheduling and dispatching
- Time management
- Dynamic storage management
- Task coordination by rendezvous
- Exception handling
- Task lifetime control
- Interrupt management
- Input/Output management
- Application boot
- Floating point arithmetic

More information about the Ada RTE can be found elsewhere [Kamrad 92].

The underlying hardware, the portability requirements, or other software requirements point out the different implementation alternatives that exist when developing an Ada application. The most likely alternatives are:

- Solution constrained by the Ada language features.
- Hybrid solution using the services of a real-time executive or operating system.
- Extended solution using a common set of user-accessible runtime environment interfaces.

The plain Ada alternative is recommended in those situations where portability is the main concern. Some of the coordination mechanisms described below could be implemented using the Ada language primitives, but in some cases this would lead to abstraction inversion situations with severe performance penalties.[5]

The hybrid solution is chosen when the underlying hardware requires it or because the software architecture is not adequately implemented by a plain Ada solution. For example, some multiprocessing and distributed systems are developed in Ada using the services of an operating system. There are commercial products supporting this approach. Some standardization efforts currently underway also support this approach [ARINC 93]. Some coordination mechanisms are well-suited to this solution. The portability drawback, however, should be considered.

A third alternative is useful when increased portability is required. This solution, supported by an underway standardization effort [ARTEWG 93], provides a set of interfaces from a user's perspective. These interfaces, named "entries" in the ARTEWG's document, represent "contracts" provided to the application developers rather than to the compiler builders.

The entries supported by the Catalogue of Interface Features and Options (CIFO) not only include the coordination mechanisms that are supported but other real-time issues. The entries regarding asynchronous cooperating mechanisms are:

- Barriers
- Blackboards
- Broadcast
- Buffers
- Events
- Mutually exclusive access to shared data
- Resources
- Shared locks
- Signals

---

5.   Abstraction inversion exists when a simple coordination mechanism is simulated while using a complex one. An example would be simulating a semaphore with rendezvous. Sometimes (but not always) compiler optimizers are able to overcome the penalties of abstraction inversion.

2.2   Description of the Coordination Mechanisms

The description of the coordination mechanisms follows the 3C model of a software component[6] that is decisive for any software reusable component:

- The abstraction supported by the component (concept)
- The software environment necessary for the component to be meaningful (context)
- The implementation of the component (content)

For our purposes the description of each coordination mechanism is split into:

- Abstraction supported
- Services provided
- Implementation issues

2.2.1   Barrier

1. Abstraction Supported

    The barrier controls the simultaneous resumption(s) of some fixed number of waiting tasks.

2. Services Provided

    Once created (CREATE) tasks may wait (WAIT) at the barrier until CAPACITY tasks are waiting. When this condition occurs, the CAPACITY waiting tasks are resumed simultaneously. A barrier may be observed (COUNT, VALUE, CAPACITY) and deleted (DESTROY).

    The interfaces proposed for these services could be:

    ```
    function CREATE (CAPACITY: in CAPACITY_RANGE; NAME:
                    in        STRING:="") return BARRIER;
    function CAPACITY (B: in BARRIER) return CAPACITY_RANGE;
    procedure WAIT (B: in BARRIER);
    procedure DESTROY (B: in out BARRIER);
    function COUNT (B: in BARRIER) return WAITING_RANGE;
    function VALUE (B: in BARRIER) return CAPACITY_RANGE;
    ```

3. Implementation Issues

    The barrier is implemented as an extended solution using a common set of user-runtime environment interfaces. A more detailed description can be

---

[6.]Edwards, S. The 3C Model of Reusable Software Components. Proceedings of the Third Annual Workshop: Methods and Tools for Reuse. Syracuse University TR No. 9014. Syracuse, N.Y., 1990.

---

found in CIFO entries [ARTEWG 93]. In the presence of priority inheritance discipline, the active priority of the agents will be raised or lowered to the highest priority of any task waiting to use the services of the barrier.

2.2.2 Blackboard

1. Abstraction Supported

The blackboard makes a message visible to various reading tasks.[7]

2. Services Provided

Once created (CREATE) a blackboard may be written on (DISPLAY). The written information may be read (READ) by tasks until the information is cleared (CLEAR). A blackboard may be observed (COUNT, STATE) and deleted (DE-STROY).

The interfaces provided for these services could be:

```
function CREATE (NAME: in STRING:= "") return BLACKBOARD;
procedure READ (B: in BLACKBOARD; M: out MESSAGE);
procedure DISPLAY (B: in BLACKBOARD; M: in MESSAGE);
procedure CLEAR (B: in BLACKBOARD);
procedure DESTROY (B: in out BLACKBOARD);
function COUNT (B: in BLACKBOARD) return WAITING_RANGE;
function STATE (B: in BLACKBOARD) return BLACKBOARD_STATE;
```

3. Implementation Issues

The blackboard is implemented as an extended solution using a common set of user-runtime environment interfaces. A more detailed description can be found in CIFO entries [ARTEWG 93].

In the presence of priority inheritance discipline the active priority of the blackboard will be raised or lowered to the highest priority of any task waiting to use the services of the blackboard.

---

7. The blackboard described here is not necessarily the blackboard concept used in artificial intelligence. An example in Ada can be found in Steven P. Stockman, ABLE: An Ada-Based Blackboard System. Proceedings of AIDA-88, November 1988.

2.2.3 Bounded-Buffer

1. Abstraction Supported

    The bounded-buffer, also called *buffer*, is a temporary data holding area. Data producers and consumers make calls to send and get data from the bounded-buffer. Since the producers and consumers can run asynchronously, the buffer has a bounded capacity for storing data.

2. Services Provided

    Once created (CREATE) a bounded-buffer may be used by producer tasks to send (SEND) messages. Such messages may be consumed (RECEIVE) by the consumer tasks. CAPACITY messages may be kept in the buffer. A bounded-buffer may be observed (MESSAGE_COUNT, MESSAGE_VALUE) and deleted (DESTROY).

    The interfaces provided by these services could be:

    ```
    function CREATE (CAPACITY: in CAPACITY_RANGE:= 1; NAME:
                          in STRING:="") return BUFFER;
    function CAPACITY (B: in BUFFER) return CAPACITY_RANGE;
    procedure RECEIVE (B: in BUFFER; M: out MESSAGE);
    procedure SEND (B: in BUFFER M: in MESSAGE);
    procedure DESTROY (B: in out BUFFER);
    function MESSAGE_COUNT (B: in BUFFER) return MESSAGE_COUNT_
                              RANGE;
    function MESSAGE_VALUE (B: in BUFFER; I: in CAPACITY_RANGE:= 1)
    return MESSAGE;
    ```

3. Implementation Issues

    Examples of bounded-buffers are implemented either using Ada language [NSWC 88], a real-time executive [ARINC 93] , or a common set of user-runtime environment interfaces knows as CIFO entries [ARTEWG 93]. Ada 9X protected types are also used implementing a bounded-buffer [Barnes 93].The bounded-buffer could be implemented as a passive task.

    In the presence of priority inheritance discipline the active priority of the agents will be raised or lowered to the highest priority of any task waiting to use the services of the buffer. The queues that are used in the bounded-buffer would ideally implement priority queues.

2.2.4 Broadcast

1. Abstraction Supported

    The broadcast coordination mechanism supports sending a message to all
    the tasks waiting for it.

2. Services Provided

    Once created (CREATE) a broadcast may be used to send (SEND) a message
    to all the tasks waiting for it (RECEIVE). This message is then consumed. A
    broadcast may be observed (COUNT) and deleted (DESTROY).

    The interfaces provided for these services could be:

    ```
    function CREATE (NAME: in STRING:= "") return BROADCAST;
    procedure      RECEIVE (B: in BROADCAST; M: out MESSAGE);
    procedure SEND (B: in BROADCAST; M: in MESSAGE);
    procedure DESTROY (B: in out BROADCAST);
    function COUNT (B: in BROADCAST) return WAITING_RANGE;
    ```

3. Implementation Issues

    The broadcast is implemented as an extended solution using a common set
    of user-runtime environment interfaces. A more detailed description can be
    found in CIFO entries [ARTEWG 93].

    In the presence of a priority inheritance discipline the active priority of the
    agents will be raised or lowered to the highest priority of any task waiting to
    use the services of the broadcast mechanism.

2.2.5 Event

1. Abstraction Supported

    The event is a mechanism that allows notification of any waiting tasks upon
    the occurrence of a discrete condition. There are two possible states, often
    called *up* and *down*. *Up* indicates that the condition is true, and *down* indi-
    cates it is false.

2. Services Provided

    Once created (CREATE) an event coordination mechanism may be set
    (SET) to the state "condition has occurred" or reset (RESET). Tasks may wait
    for this occurrence (WAIT). The event coordination mechanism may be ob-
    served (COUNT, STATE) and deleted (DESTROY).

    The interfaces provided by these services could be:

```
function CREATE(INITIAL: in EVENT_STATE:= DOWN; NAME:
                    in STRING:="")
return EVENT;
procedure WAIT (E: in EVENT);
procedure SET (E: in EVENT);
procedure RESET (E: in EVENT);
procedure DESTROY (E: in out EVENT);
function COUNT (E: in EVENT) return WAITING_RANGE;
function STATE (E: in EVENT) return EVENT_STATE;
```

3. Implementation Issues

Examples of event mechanism are implemented either using a real time executive or a common set of user-runtime environment interfaces known as CIFO entries [ARTEWG 93].

In the presence of priority inheritance discipline the active priority of the agents will be raised or lowered to the highest priority of any task waiting to use the services of the event object. The queries that are used in the event object would ideally implement priority queues.

2.2.6 General-Semaphore

1. Abstraction Supported

The general-semaphore is a non-negative integer value used as a synchronization mechanism.

2. Services Provided

The general-semaphore is created (CREATE) as an object with a specified current value and a maximum value. The maximum value parameter (MAXIMUM_VALUE) is the maximum value to which the general-semaphore can be signalled. The current value (CURRENT_VALUE) is the general-semaphore's starting value after creation. The operation (WAIT) is used to suspend a client process if there are no free resources (that is the current value =0). The operation signal is used to wake a previously suspended process, if any exist. The general-semaphore can be observed (GET_STATUS) and reset (RESET).

3. Implementation Issues

The general semaphore is implemented using a real-time executive [ARINC 93]. In this implementation, a waiting process is queued until either a signal is received or until a specified amount of time expires. Signalling a semaphore increments the semaphore value if no suspended processes wait on the semaphore. When processes are waiting on the semaphore, the signal will be given to the highest current priority queued process. In the event there are multiple processes of the

same highest current priority, a First In First Out (FIFO) algorithm will be applied to determine which process will receive the signal.

2.2.7 Lock

1. Abstraction Supported

The lock is a mechanism that arbitrates the accesses to a resource. The user tasks can either request exclusive or shared access to the lock.

2. Services Provided

Once a lock is created (CREATE), any task user must obtain an identification (NEW_LOCK_USER) prior to using it. Then the task user can request the lock, either waiting until it is granted (WAIT_REQUEST) or continuing in any case (NO_WAIT_REQUEST). The release procedure (RELEASE) is called to change the status of the lock from locked to unlocked. A task user can consult the current status of the lock (STATUS_LOCK). The lock can be deleted (DESTROY).

The interfaces provided by these services could be:

```
function CREATE return LOCK_ID;

function NEW_LOCK_USER return LOCK_USER_ID;

function WAIT_REQUEST (THE_LOCK: in LOCK_ID; THE_USER: in
                      LOCK_USER_ID;REQUEST_TYPE: in
                      ACCESS_TYPE);

return BOOLEAN;

function NO_WAIT_REQUEST(THE_LOCK:in LOCK_ID;THE_USER: in
                      LOCK_USER_ID;REQUEST_TYPE: in
                      ACCESS_TYPE)

return BOOLEAN;

procedure RELEASE (THE_LOCK: in LOCK_ID;THE_USER: in
                  LOCK_USER_ID);

function STATUS_LOCK (THE_LOCK: in LOCK_ID; THE_USER: in
                  LOCK_USER_ID)

return LOCK_STATUS;

procedure DESTROY (THE_LOCK: in out LOCK_ID);
```

3. Implementation Issues

The above interface is based on the Ada implementation of a lock manager [NSWC 88]. An alternative to avoid the overhead of the Ada rendezvous is to take full advantage of hardware or real-time operating system instructions for resource serialization. A common Ada interface to such capabilities is provided in CIFO en-

tries [ARTEWG93]. A generic package is proposed to provide combinations of the desired resource access (*exclusive only* versus *shared and exclusive*) plus the queuing discipline (FIFO, priority) to be used when the lock is unavailable.

2.2.8 Mailbox

1. Abstraction Supported

   A mailbox is an object that two or more processes can use to pass messages between them via the SEND and RECEIVE operations.

2. Services Provided

   Once opened (OPEN), the mailbox can be used to transfer messages. A process sending a message (SEND) has the option of continuing or waiting until the message is received. Similarly, a process seeking a message at a mailbox (RECEIVE) that presently has no messages can either continue or wait for the next message to arrive. The content of the message is not copied until the receiver is available and the transfer is made directly to the receiver process. A mailbox can be closed (CLOSE).

3. Implementation Issues

   An example of a mailbox based on a real-time executive is described elsewhere [Ripps 89]. There is a version of this executive supporting Ada applications [Industrial 88].

2.2.9 Pulse

1. Abstraction Supported

   A pulse is a synchronization mechanism that notifies any waiting tasks upon the occurrence of a pulsed condition. This occurrence is not latched so it does not influence any later waiting task.

2. Services Provided

   Once created (CREATE), a pulse may be set (SET) to notify such an occurrence to the tasks waiting for it (WAIT). A pulse may be observed (COUNT) and deleted (DESTROY).

   The interfaces provided by these services could be:

   ```
   function CREATE (NAME: in STRING:= "") return PULSE;
   procedure WAIT (P: in PULSE);
   procedure SET (P: in PULSE);
   procedure DESTROY (P: in out PULSE);
   function COUNT (P: in PULSE) return WAITING_RANGE;
   ```

---

3. Implementation Issues

The pulse is implemented as an extended solution using a common set of user-runtime environment interfaces. More detailed information can be found in CIFO entries [ARTEWG 93].

In the presence of priority inheritance disciplines active priority of the agents will be raised or lowered to the highest priority of any task waiting to use the services of the pulse mechanism.

2.2.10 Relay

1. Abstraction Supported

The relay is a temporary data holding area. The relay accepts a message from a producer task, and then transfers the message to a consumer or another intermediary via a call.

2. Services Provided

A relay may be used by producer tasks to send messages (SEND). The relay transfers the message to a consumer via an entry call.

The interface provided could be:

```
entry SEND (M: in MESSAGE);
```

3. Implementation Issues

The relay is implemented using Ada. An example of the Ada implementation can be found elsewhere [Nielsen 88].

2.2.11 Rendezvous

1. Abstraction Supported

The rendezvous is a synchronous and unbuffered coordination mechanism that allows two tasks to communicate bidirectionaly.

2. Services Provided

The semantic of the rendezvous is as follows:

- The calling task passes its "in parameters" if any to the callee task via an entry call and is blocked pending completion of the rendezvous.
- The callee task executes the statements in its accept body.
- The "output parameters" if any are passed back to the calling task.
- The rendezvous is now complete and both tasks are no longer suspended.

The interface provided by the callee task could be:

```
entry RECEIVE(INPUT_PARAMETER: in MESSAGE;
              OUTPUT_PARAMETER:out MESSAGE);
```

3. Implementation Issues

The rendezvous is the Ada mechanism for communicating tasks, so it is support-ed by all validated Ada runtime systems.

2.2.12 Signal

1. Abstraction Supported

A task can send a signal to another task or to a group of tasks as a means of co-ordinating those tasks. Signals differ from other coordination mechanisms in that the receiver may either wait for a signal to arrive or may receive a signal asynchro-nously while it is carrying out another activity.

2. Services Provided

Once a task is created, its response to a set of signals may be established (SET_-SIGNAL_RESPONSE). One task can send a signal (SEND_SIGNAL) to a partner, thus invoking whatever response the receiver currently has in face. A receiver can pause for a signal that is sent by another task (PAUSE_SIGNAL).

3. Implementation Issues

A signal is considered to be a software interrupt to be handled at the task level. Signal features described above are based on an implementation using a real-time executive [Ripps 89]. A version of this executive supports Ada applications. Another implementation is proposed using an Ada generic package [ARTEWG 93]:

```
generic
with procedure TO_BE_CALLED;
package SIGNAL
procedure NON_WAITING;
end SIGNAL;
```

The effect of calling the procedure NON_WAITING, which is exported by an in-stance of the SIGNAL generic package, is the same as creating a thread which calls the TO_BE_CALLED procedure.

2.2.13 Timed-Buffer

1. Abstraction Supported

    The timed-buffer provides a temporary storage and blocking of data to be transmitted between asynchronous tasks. The timed-buffer has a bounded capacity and the storage time of a particular message is limited.

2. Services Provided

    A timed-buffer may be used by a producer task to send (SEND) messages. When either the timed-buffer becomes full or its time-out period is reached, the data currently in the timed-buffer is sent to the receiving task, the timed-buffer is empty, and additional data accepted. The time-out period of the timed-buffer can be changed (SET_TIME_OUT). The timed-buffer may be deleted (DESTROY). The interfaces provided by these services could be:

        procedure SEND (B: in TIMED_BUFFER; M: out MESSAGE);
        procedure SET_TIME_OUT (TIME_OUT: in DURATION);
        procedure DESTROY (B: in out TIMED_BUFFER);

3. Implementation Issues

    An example of a timed-buffer implementation using Ada is described in the literature [NSWC 88]. The timed-buffer is a generic package requiring the following instantiation parameters:

    • The data item type which will be placed into the timed-buffer.
    • The size of the timed-buffer.
    • An array of the data type of the data with its size equal to the timed-buffer size.
    • A procedure SEND to send the stored data to the consumer task.
    • An optional value for the time-out period for the buffer.
    • An optional value representing how many real buffers exist (single, double, or triple buffering).

2.2.14 Transporter

1. Abstraction Supported

    A transporter can be considered to be an active data mover. The transporter makes calls to get and send messages from and to the coordinating tasks.

2. Services Provided

    A transporter is strictly a caller which gets a message from one producer (PRODUCER.PROVIDE_MESSAGE) and passes the message on to one consumer or another intermediary via a call (CONSUMER.TAKE_MESSAGE).

3. Implementation Issues

The transporter is implemented using Ada. An example of Ada implementation can be found elsewhere [Nielsen 88].

3 Properties of the Coordination Mechanisms

The previous experiences and the analysis of the coordination mechanisms used in Ada real-time systems allow us to determine the properties of the coordination mechanisms that could be useful in classifying them.

We split the properties into three main groups:

- Synchronization properties
- Communication properties
- Implementation issues

3.1 Synchronization Properties

Synchronization properties support basically the exclusion and prioritization capabilities of the coordination mechanism.

Exclusion properties guarantee the blocking of certain processes wishing to access the mechanisms when those processes will interfere with the coordination activities already in progress for other processes.

Some coordination mechanisms support exclusion of client processes based on the internal state of the mechanism. For example, a full buffer does not process a write request.

Exclusion is sometimes provided by considering the state of the synchronization of the coordination mechanism. An example is the barrier where the value of the count of the processes currently accessing the barrier is the condition to open it.

Prioritization is related to the ordering of the client requests to be processed. Sometimes requests are queued based in the client priority but frequently the time of arrival is the major issue in ordering the client requests. So the majority of the coordination mechanisms manage some kind of FIFO queue for prioritization.

3.2 Communication Properties

The communication properties of a coordination mechanism can be established considering its similarities with a conventional communication link.

The properties of a communication link are related to the support of:

- Unidirectional or bidirectional flow of information
- The throughput of the communication link
- The size of the messages being transported

In the same way, the communication properties of a coordination mechanism are related to:

- The direction of the information flow
- The storage capacity of the coordination mechanism
- The kind of information transmitted

In the case of pure synchronization mechanisms the information transmitted is null: those coordination mechanisms that are event-oriented transmit unitary information. The rest of coordination mechanisms are message-passing mechanisms.

3.3   Implementation Issues

This group of properties is related to the realization of the concept of the coordination mechanism for a particular environment. The main implementation issue is the coordination mechanism's dependency on the services provided by an underlying executive or operating system.

It is well known that the rendezvous is the synchronization and communication mechanism provided by Ada. This mechanism is well suited for building more complex mechanisms such as the relay or the transporter, but rendezvous should not be used to simulate all other mechanisms. The differences between the features of the coordination mechanisms make this approach unsuitable. Therefore, the state of the practice is that some Ada developers are using the coordination mechanisms provided by real-time operating systems.

Another important issue is whether the mechanism can be implemented as an active or passive task.

An active task is that task normally requiring various threads of control. Its entries and the rendezvous mechanism cause task switches which are indeed performance expensive in some applications.

The recognition of passive tasks was brought about by two considerations: Consider the performance penalties of active tasks and the possibility of implementing some coordination mechanisms with more simplistic approaches.

The Ada Run-Time Environment Working Group (ARTEWG) defines passive tasks as those that do not require threads of control [ARTEWG 93]. For passive tasks, entry calls can be transformed into procedure calls, avoiding the cost of task switching.

Some restrictions should apply to avoid multiple threads of control inside a task.

It is important to stress that the declarative part is restricted to any declaration except declarations of dependent tasks, inner packages, and objects of dynamic size or access types. The default initialization for any declaration is restricted from calling user-defined functions.

Also the following restrictions apply in a passive task:

- In the selective wait, no delay alternative or else part is allowed.
- Nested accept statements are not allowed.
- Entry families are not supported.
- In the task body there is one and only one accept statement for each entry declared in the task specification.
- The specification of storage size for task activation is not allowed for a passive task and is ignored by the implementation if present.
- Passive tasks may not have dependent tasks.
- No exception handler is allowed for the task body.
- The guard expressions must not contain calls to user-defined functions.
- The only statement allowed outside of select and accept bodies is an unconditional loop.

Ada 9X introduces the concept of *protected type* which encapsulates and provides synchronized access to the private data of objects of the type without the introduction of an additional task.

As described by Barnes, protected objects are very similar to monitors[8]; they are passive mechanisms with synchronization provided by the runtime system. One advantage protected objects have over monitors is that the protocols are described by the barrier conditions rather than the low-level signals internal to the monitor [Barnes 93].

A bounded-buffer is a mechanism that can be implemented as an active or passive task. Its implementation as an active task can be found elsewhere [Nielsen 88]. ARTEWG proposes the following coding that could be compiled as a passive task [ARTEWG 93]:

```
task BUFFER is
      entry READ (C: out CHARACTER);
      entry WRITE (C: in CHARACTER);
      pragma THREAD_OF_CONTROL(FALSE);
end BUFFER;

task body BUFFER is
      POOL_SIZE: constant INTEGER:= 100;
      POOL: array(1..POOL_SIZE) of CHARACTER;
```

---

8. A monitor is a collection of routines which controls and protects a particular resource. The important characteristic of a monitor is that only one of its procedure bodies can be active at one time; even when two processes call a procedure simultaneously, one of the calls is delayed until the other is completed. Thus the procedure bodies act like critical regions protected by the same semaphore. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall 1985.

```
        COUNT: INTEGER range 0.. POOL_SIZE:= 0;
        IN_INDEX, OUT_INDEX: INTEGER range 1.. POOL_SIZE:= 1;
    begin
        loop
                select
          when COUNT < POOL_SIZE =>
          accept WRITE (C: in CHARACTER) do
          POOL(IN_INDEX):= C;
          IN_INDEX:= IN_INDEX mod POOL_SIZE + 1;
                COUNT:= COUNT + 1;
                 end WRITE;
                 or when COUNT > 0 =>
                 accept READ (C: out CHARACTER) do
                 C:= POOL(OUT_INDEX);
                 OUT_INDEX:= OUT_INDEX mod POOL_SIZE + 1;
                 COUNT:= COUNT - 1;
                 end READ;
      or
                terminate;
                end select;
                end loop;
                end BUFFER;
```

It is important to realize how a pragma is used to give compiler directives. Other approaches are based on the automatic recognition by the compiler of passive task candidates. Research is being done in this area [Schilling 93].

We can compare it with the following implementation given by Barnes using Ada 9X protected types [Barnes 93]:

```
        protected type BOUNDED_BUFFER is
                entry PUT (X: in ITEM);
                entry GET (X: out ITEM);
         private
                A: ITEM_ARRAY (1.. MAX);
                I, J: INTEGER range 1.. MAX:= 1;
                COUNT: INTEGER range 0.. MAX:= 0;
          end BOUNDED_BUFFER;

        protected body BOUNDED_BUFFER is
                entry PUT (X: in ITEM) when COUNT < MAX is
                begin
                A(I):= X;
                 I:= I mod MAX + 1;
                COUNT:= COUNT + 1;
                end PUT;
```

```
entry GET (X: out ITEM) when COUNT > 0 is
begin
 X:= A(J);
J:= J mod MAX + 1;
COUNT:= COUNT - 1;
end GET;
end BOUNDED_BUFFER;
```

Another implementation issue that can be considered in several situations is the information passing mechanism. Sometimes it can be done by copying the entire data transmitted (passing by value); other times the data is passed by reference.

4   Features Model of the Coordination Mechanisms

Features are the attributes of a system that directly affect end-users [Kang 90].

In our particular situation, we are not performing a full domain analysis of a family of systems; our goal is less ambitious. We are trying to classify consistently the coordination mechanisms to be used for Ada real-time applications. So our end user is the software engineer; the attributes to be analyzed are those that will help the software engineers in the selection of the coordination mechanism best matching their design and implementation constraints.

The features model represents those attributes and considers the relationships between them. The structural relationship, which represents a logical grouping of features, is of major interest. Alternative or optional features of each grouping are also represented in the features model.

A graphical representation and a textual description of the features selected is presented below. The graphical notation follows the notation and conventions used in the feature-oriented domain analysis of the Army movement control domain [Cohen 92].

The notation for describing the basic relationships between features (mandatory, alternative, and optional) is given below.

Feature Type	Notation in Graphical Representation
Mandatory	*Feature-name (tuple of: X)*. This notation is used for parent features where X is the number of child features. Feature-name signifies leaf feature.
Alternative	*Feature-name (one of: X)*. This notation is used for parent features where X is the number of alternate children.
Optional	Optional leaf features are denoted by the (boolean) notation after the name.[*]

---

[*]Boolean notation is not used in the features model of the coordination mechanisms.
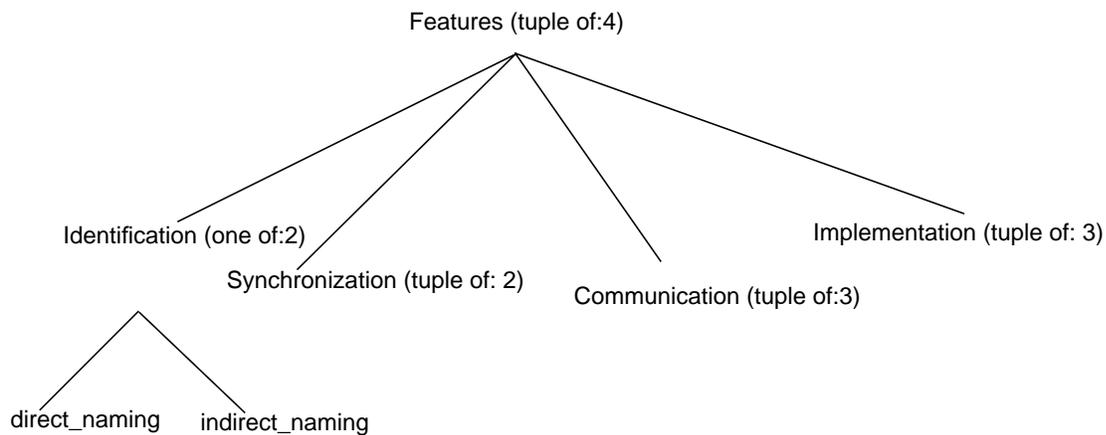
Figure 4-1: Top Level Features Model for the Coordination Mechanisms

As shown in Figure 4-1, there are four main groups of features within the coordination mechanisms domain:

1. *Identification.* Those features that describe the way in which the coordinating processes that want to synchronize or communicate refer to each other.

2. *Synchronization.* Those features that describe the exclusion and prioritization properties of the coordination mechanisms.

3. *Communication.* Those features that describe the information transfer capabilities of the coordination mechanisms.

4. *Implementation.* Those features that describe the realization aspects of the coordination mechanisms.

Each of these groups of features is described in more detail in the following paragraphs.

4.1   Identification Features

The identification features shown in Figure 4-1 fall in two major categories or naming schemes: direct and indirect naming.

1. *Direct Naming.* When using this discipline, a process that wants to synchronize or communicate with another must explicitly name the recipient partner.

   A communication between processes using this naming scheme has the following properties [Silberschatz 91]:

   • A link is established between every pair of processes that want to communicate. The caller needs to know the identity of the task to be called in order to communicate.

- A link is associated with exactly two processes.
- Between each pair of communicating processes, there already exists one link.

2. *Indirect Naming.* When using this discipline, the processes use some kind of intermediate object to place or remove the transmitted information, so coordination partners identities are hidden.

   This naming scheme has the following properties [Silberschatz 91]:

   - A link is established between a pair of processes only if they share a coordination mechanism.
   - A link may be associated with more than two processes.
   - Between each pair of communicating processes, there may be a number of different links, each corresponding to one coordination mechanism.
   - A link may be either unidirectional or bidirectional.[9]

4.2   Synchronization Features

The synchronization features shown in Figure 4-2 provide the means of describing the exclusion and prioritization constraints of the coordination mechanisms.

The two top-level synchronization features are exclusion constraints and ordering constraints. These features are described below.

1. *Exclusion constraint features* describe the properties that guarantee the blocking of certain processes that are attempting to access the coordination mechanism but that will interfere with the coordination activities already in progress by other processes. Some coordination mechanisms support exclusion of client processes based on the internal state of the mechanism. For example a full buffer does not process a write request. A second group of coordination mechanisms supports exclusion based on the synchronization state of the mechanism. Synchronization state includes all state information needed only for synchronization purposes. An example is the barrier mechanism, where the value of the count of the processes currently accessing the barrier is the condition to open it. A third group of coordination mechanisms support exclusion based on the history that has occurred. That is, as in the case of the pulse mechanism, the execution of its set procedure resumes all waiting tasks in an order that is implementation dependent. This occurrence is not latched, so later waiting tasks are not resumed by the previous set procedure execution.

   Exclusion constraints may be either single enabling or multiple enabling. Single enabling mechanisms permit only one client process when more than one

---

9. An unidirectional link supports only flow of data in one direction. A bidirectional link supports data interchange in either direction. This feature is considered as a communication feature. See Section 4.3 of this report.

is waiting for coordination. Multiple enabling mechanisms release all client processes that satisfy the exclusion condition.

2. *Ordering constraint features* are concerned with the prioritization of the client requests to be processed. Sometimes requests are queued based on the client priority. Another group of coordination mechanisms orders the requests by their time of arrival.
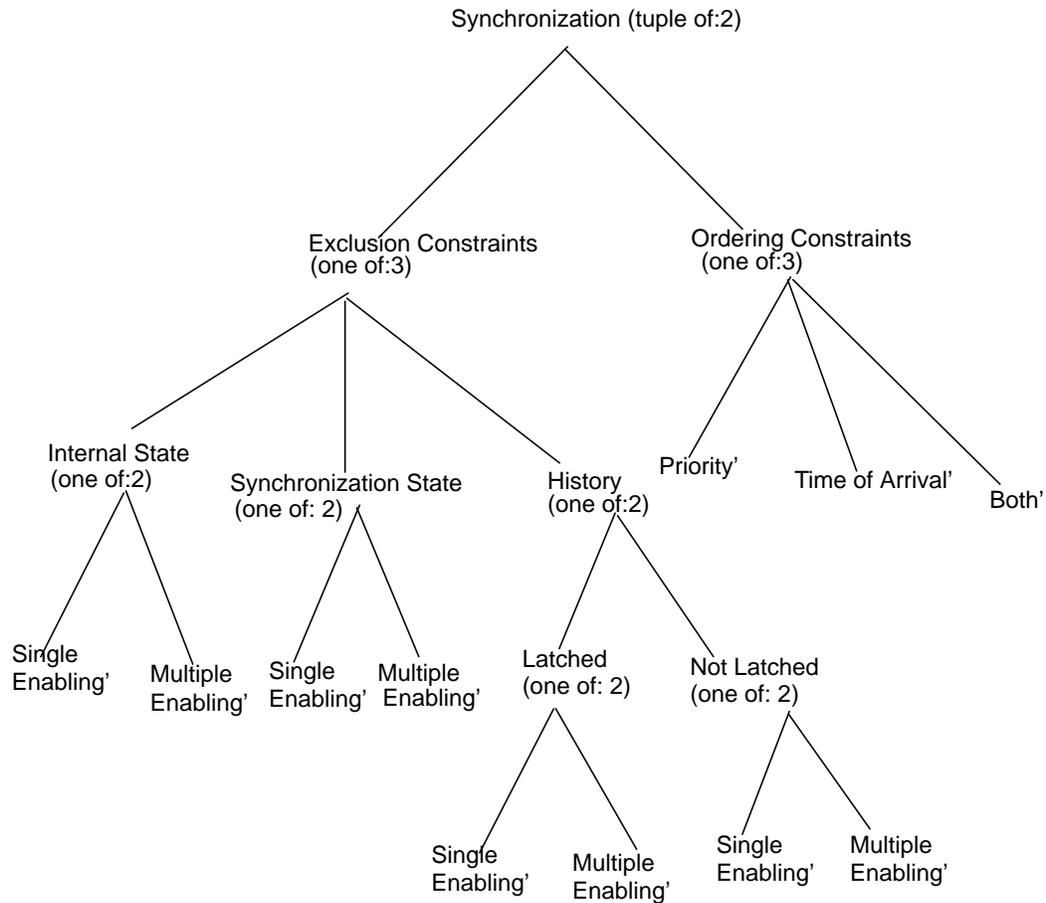
Figure 4-2: Synchronization Features

4.3   Communication Features

The communication features describe the communication capabilities of the coordination mechanism. For the mechanisms analyzed, we found three child features to represent communication properties: storage capacity, flow of information, and transmitted information. These features are shown in Figure 4-3.

1. *Storage Capacity.* This feature characterizes the capacity of the coordination mechanism for buffering the events or messages transferred between communicating partners. The capacity may be either null, bounded, or unbounded. The Ada rendezvous could be considered as an example of null capacity communication mechanism.

2. *Direction of Flow*. When the communication is established the flow of data can be in either one or two directions. Unidirectional mechanisms are those where the flow of data is in one direction. Bidirectional mechanisms are those where the flow of data is in two directions. Whether the mechanism is a pure synchronization mechanism, there is no flow of data.

3. *Transmitted Information*. The information transmitted when a link is established between two or more communicating partners may be either none, unitary, or multiple. In pure synchronization mechanisms, the information transmitted is none. Event-oriented coordination mechanisms transmit unitary information. Message-oriented coordination mechanisms transmit multiple information. Messages may be either fixed length or variable length. One of the differences between a buffer and a mailbox is specifically the length of the messages. The buffer handles fixed length messages; the mailbox handles variable length messages.
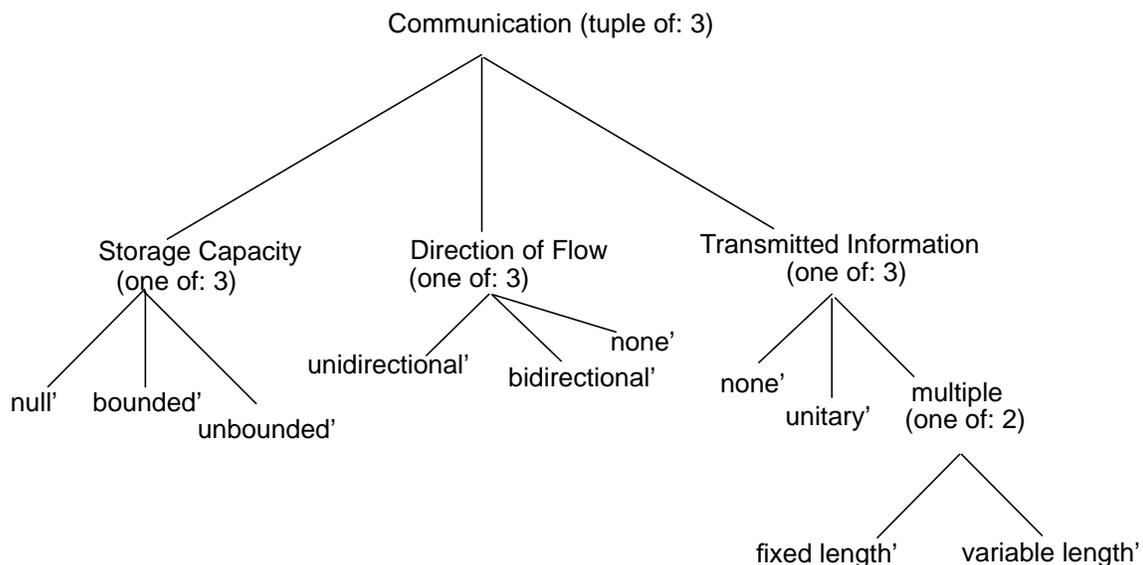
Figure 4-3: Communication Features

4.4   Implementation Features

The implementation features denote those features related to the realization of the coordination mechanism for a particular environment.

The implementation features fall in three major categories: operating system dependency, information passing mechanism, and the existent threads of control.

These three groups of features are shown in Figure 4-4.

Implementation (tuple of: 3)

OS Dependency
(one of: 3)

Information Passing Mechanism
(one of: 2)

Threads of Control
(one of: 2)

OS Services'   Ada'   By Copy'   By Reference'   Active'   Passive'

External Library'
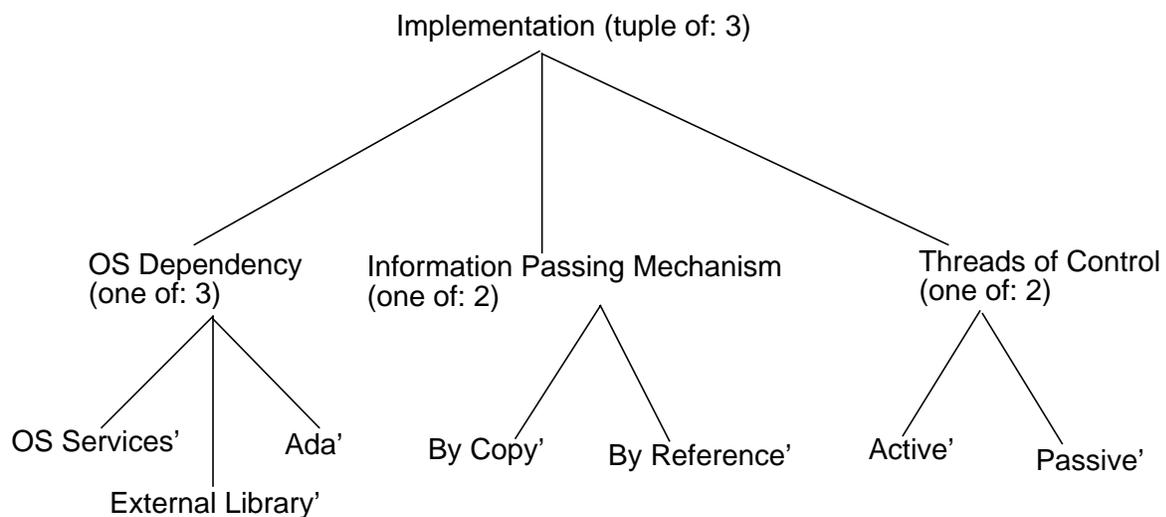
Figure 4-4: Implementation Features

The implementation features categories are:

1. *Operating System Dependency.* This feature determines the dependency between the coordination mechanism and the services supported by a real-time executive or operating system. We can distinguish three different situations:

   • The coordination mechanism is implemented using a real-time executive or operating system.

   • The coordination mechanism is implemented by an extended runtime library that supplements the original runtime library, for example CIFO entries [ARTEWG 93] or EXTRA entries [RRG 93].

   • The coordination mechanism is implemented using the Ada language.

2. *Information Passing Mechanism*. This feature describes the information transferring mode used in the coordination mechanism. Information transfer can be done by copying the entire data transmitted or by passing data by reference.

3. *Threads of Control*. Those features characterize the implementation of the coordination mechanism as an active task or as a passive task. The requirements to be met by an Ada task to be compiled as a passive task are described in Chapter 3.

5   Guidelines for Using the Taxonomy

5.1   Application of the Guidelines

The following sections exhibit a collection of mechanically applicable design rules created based on the features model described above and known experiences in the design of real-time systems.

The order applied to the guidelines tries to be consistent with the specific choices that should be taken during the software development cycle of a real-time system. It is not the purpose of this report to avoid existent design methodologies; rather, we hope to complement them. Our goal is to help the software engineer in the selection of the most suitable coordination mechanism for the particular situation to be solved.

The guidelines are split in several groups related to the features model. Not all the guidelines should always be applied. Designer experience could dictate the optimum approach.

Object boundaries in an object-based concurrent system are determined by the mechanisms for abstraction, distribution, and synchronization [Wegner 92].The abstraction boundary is the interface an object presents to its clients. The distribution boundary is the boundary of accessible names visible from within an object. The synchronization boundary of an object is the boundary at which threads entering an object synchronize with ongoing activities in the object.

The following guidelines deal with issues concerning object boundaries of either the coordination mechanisms or the coordinating partners themselves.

5.2   Abstraction Related Guidelines

What is the problem to be solved?
Three common problems may arise when objects interact with each other:

- the mutual exclusion problem
- the producer consumer problem
- the readers and writers problem

The mutual exclusion problem occurs when objects need exclusive access to a resource, such as a shared data or physical device. Mechanisms such as the general semaphore or the lock are well suited to resolve the mutual exclusion problem.

The producer consumer problem occurs when objects need to communicate in order to pass data from one partner to another partner. Message-passing mechanisms are recommended. The selection of the best candidate is based on guidelines explained later.

The readers and writers problem is similar to the mutual exclusion problem but reader objects are not required to exclude one another. This problem is an abstraction of access to databas-

---

es, where there is no danger in having several processes read concurrently but writing or changing the data must be done under mutual exclusion to ensure consistency. Some multiple enabling mechanisms are well suited to this purpose, for example, the blackboard. Other guidelines should be considered in the selection process.

Can we simulate one mechanism with another?

Three issues should be evaluated when trying to simulate one coordination mechanism with another [Ripps 89]:

- the extent to which the features of the coordination mechanism can be simulated
- the degree to which the unblocking function used in one mechanism can be simulated using different mechanisms
- the efficiency, clarity, and vulnerability of such simulations

Concurrency Coupling

A concurrent system is loosely coupled if the partners' interactions are well balanced in terms of caller/callee decisions and the use of intermediaries; the amount of busy wait has been minimized and appropriate modes have been used for the parameters passed [Nielsen 88].

The direct naming schemes, particularly the Ada rendezvous, imply tightly coupling between the coordinating partners.

When loose coupling is required between the coordinating partners, for example, a producer consumer pair, intermediary coordination mechanisms such as a buffer, transporter, relay or combinations are introduced.

Five different paradigms can be established for loosely coupling between a producer consumer pair:

Producer-Buffer-Consumer. The use of a buffer allows the producer and consumer to operate asynchronously.

Producer-Buffer-Transporter-Consumer. This paradigm avoids the consumer having to wait for the rendezvous with the buffer task. It supports the consumer as a server object.

Producer-Transporter-Buffer-Transporter-Consumer. This paradigm also avoids the producer having to wait for the rendezvous with the buffer task.

Producer-Relay-Consumer. This paradigm is adequate in such cases where the producer is an agent object and the consumer is a server object, but has the drawback of the lack of storage capacity of the relay.

Producer-Timed_Buffer-Consumer. This paradigm is adequate when a full set of data is needed by the consumer but can only be supplied slowly by the producer.

5.3   Implementation Related Guidelines

Portability Issues

Portability requirements are usually specified during the requirements specification phase. Portability requirements can constrain the selection of the available coordination mechanisms.

Those designers using Ada-based coordination mechanisms should avoid the selection of mechanisms implemented by a real-time executive. The simulation of a coordination mechanism implemented by a real-time executive using Ada language is sometimes inadvisable considering the Ada rendezvous features and the differences with other mechanisms. (See Appendix A).

Those designers using a real-time executive should avoid frequent use of the rendezvous mechanism. Therefore, coordination mechanisms based on rendezvous are excluded.

Those mechanisms provided by external libraries (Appendix A), give the designer the highest flexibility in the selection process. Therefore, they are very suitable for medium and complex real-time applications.

5.4   Communication-Related Guidelines

Can the link be associated with more than two partners?

As described in Section 4.1, indirect naming schemes are more flexible in supporting multiple communicating partners, so coordination mechanisms using indirect naming provide software architectures easily adapted to support additional partners.

Indirect naming schemes are also best suited to perform load leveling, that is, distribute the transactions equitable between consumer processes. But if the critical issue is to coordinate with the completion of the transaction, direct naming schemes, for example the Ada rendezvous, are more appropriate.

Modularity is also improved with indirect naming schemes.

Exclusion and ordering constraints should also be considered when selecting a coordination mechanism for multiple partner communication. Message exchange does not necessarily lead to mutual exclusion.

Is a link with some storage capacity needed?

Considering the producer consumer problem, it is necessary to select a coordination mechanism with storage capacity whenever the producer and consumer processes execute at different periods unless blocking situations are permitted.

It is important to mention that in non-zero-capacity coordination, the sender does not know whether a message has arrived at its destination after the SEND operation is completed. This concern is addressed in the following paragraph.

Is the link unidirectional or bidirectional?

Bidirectional links are required when the sender requires a reply from the receiver.

A bidirectional link can be established using a bidirectional coordination mechanism or two unidirectional coordination mechanisms. The first approach implies more coupling between the sender and the receiver. The use of two unidirectional coordination mechanisms is performance-expensive but implies less coupling between the communicating partners.

Which kind of information is transmitted?

If no information is transmitted between the coordinating partners, a pure synchronization mechanism is more suitable. Examples are the barrier, the pulse, and the general semaphore mechanisms.

In case unitary information is transmitted, event-oriented coordination mechanisms are appropriate. But if different event flows are transmitted between the sender and the receiver, a message-passing coordination mechanism should be selected.

Transmitting multiple information requires a message-passing coordination mechanism such as a broadcast, blackboard, buffer, mailbox, relay, or transporter. Frequently the message has an Ada type, so it is subject to the strong Ada-type rules. The mailbox is the most flexible mechanism regarding this issue. Synchronization and implementation features should be taken into account in the selection process.

5.5   Synchronization Related Guidelines

Is a multiple-enabling method required?

When it is necessary to release more than one waiting partner at a time, multiple-enabling coordination methods are required. Blackboard, broadcast, event, and pulse are multiple-enabling coordination mechanisms. The selection of the best candidate is based on diverse considerations including exclusion constraints, message-passing properties, and other features.

Ordering of Requests

Some coordination mechanisms only order pending requests by the time of arrival, so a FIFO ordering is provided. Whenever the relative importance of pending requests is to be handled, coordination mechanisms providing priority queues should be considered.

5.6   Scheduling Issues

It is not the purpose of this report to establish guidelines for the scheduling of real-time applications. A diversity of scheduling techniques is proposed in the literature [Xu 93].

Controlling Priority Inversion

Priority inversion is defined as the situation in which a low-priority process is using a resource while a higher-priority process is forced to wait for the resource.

Unbounded priority inversion may occur in processes that synchronize to share a resource in a mutually exclusive manner or processes implementing the producer consumer paradigm. So the designer's objective should be to bound the priority inversion.

Klein et al. propose diverse recommendations and guidelines to bound priority inversion when using coordination mechanisms [Klein 93]. These guidelines are summarized below.

In the mutual exclusion problem, the use of some synchronization protocol that will bound priority inversion is recommended. If the operating system or the runtime system do not provide one of these protocols, alternatives are proposed:

- Increase the priority of the task just before entering the resource
- Disable task preemption just before entering the resource
- Encapsulate all access to the resource in a task with an assigned priority level equal to the ceiling of the resource

In the consumer producer problem, priority inheritance maybe recommended as the solution to the unbounded priority inversion. This solution is described in the implementation issues paragraph of the coordination mechanisms (Chapter 2).[10] If several producers are sending data to a consumer, it is required to order the messages considering the sender priority.

---

[10]   Priority inheritance tends to work mostly in directed mechanisms where the current blocker can be identified. Hence without modification, it fails in general producer-consumer synchronization models, where the producer does not know the consumer is (or vice versa). Ragunathan Rajkumar. Personal communication. November 1993.

6   Conclusions

The taxonomy proposed provides a design space for real-time systems that can be used for several purposes:

- as a classification scheme for reusable software components
- as guidelines for novice designers of real-time systems
- as criteria for reverse engineering activities, helping to understand existing applications
- as the basis for the developing of real-time generic software architectures in diverse domains

The taxonomy was developed considering Ada applications. Its use for the development of applications in other languages such as C is possible considering only the coordination mechanisms supported by real-time operating systems.

Some of the components classified in the taxonomy have been used in the development of pilot real-time monitoring systems for small factory[11] and laboratory applications.[12] The functionalities implemented are:

- Data acquisition
- Range checking
- Conversion to engineering units
- Alarm checking
- Historical data storage

Testing the taxonomy and the guidelines in other real-time domains is highly recommended.

Further research is recommended for communication and synchronization in distributed systems or between partitions executing in the same or on different core module.[13]

---

[11]   F. J. Garcia. Realizacion en Ada del Software de un Sistema de Adquisicion de Datos para una Pequena Industria. Proyeto Fin de Carrera. Madrid Technical University, 1993.

[12]   B. Alvarez. Desarrollo de un Sistema de Monitorizacion de Procesos para Laboratorio. Proyecto Fin de Carrera. Madrid Technical University, 1993.

[13]   A partition is basically the same as a program in a single application environment: it comprises data, its own context, configuration attributes, etc. [ARINC 93].

---

7   References

[ARTEWG 93]   Ada Run-Time Environment Working Group. *Catalogue of Interface Features and Options for the Ada Runtime Environment.* ACM, Draft Release 3, May 1, 1993.

[ARINC 93]   Aeronautical Radio Inc. *Avionics Application Software Standard Interface.* AECC, Draft 7, August 13, 1993.

[NSWC 88]   Naval Surface Weapons Center. *Ada Run-Time Support Services for Complex Time-Critical, Embedded Applications.User's Manual.* December 5, 1988.

[RRG 93]   Real-Time Rapporteur Group. *Proposed Draft Standard for Extensions for Real-Time Ada.* EXTRA Team Secretariat CR2A, Draft Version 3.00, October 15, 1993.

[Barnes 93]   Barnes, John. *Introducing Ada 9X.* Intermetrics, 1993.

[Bloom 79]   Bloom, Toby. *Evaluating Synchronization Mechanisms.* Proceedings of the 7th Symposium on Operating Systems Principles. Pacific Grove, California: ACM SIGOPS, Dec. 10-12,1979.

[Chi 91]   Chi Tau Lai, Robert. *Ada Task Taxonomy Support for Concurrent Programming.* ACM Software Engineering Notes, 16.1, January 1991.

[Cohen 92]   Cohen, Sholom; Stanley Jay; Peterson, Spencer;& Krut, Robert. *Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain.* (CMU/SEI-91-TR-28, ADA256590). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

[Industrial 88]   Industrial Programming Inc. MTOS-UX/Ada Product profile. New York, 1988.

[Kang 90]   Kang, Kyo; Cohen, Sholom; Hess, James; Novak, William; & Peterson, Spencer. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* (CMU/SEI-90-TR-21, ADA235785). Pittsburgh, Pa.: Software Engineer Institute, Carnegie Mellon University,1990.

[Kamrad 92]   Kamrad, Mike. Chapter 17, Understanding Ada Runtime Environments, 234-293. Mission Critical Operating Systems. IOS Press, 1992.

[Klein 93]   Klein, Mark; Ralya, Thomas; Pollak, Bill; Obenza, Ray; & Gonzalez Harbour, Michael . *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate monotonic Analysis for Real-Time Systems.* Norwell, MA.: Kluwer Academic Publishers,1993.

[Nielsen 88]   Nielsen, Kjell; & Shumate, Ken. *Designing Large Real-Time Systems with Ada.* New York, N.Y.: Multiscience Press, 1988.

[Petroski 92]   Petroski, Henry. *To Engineer is Human. The Role of Failure in Successful Design.* Vintage Books, 1992.

[Ripps 89]   Ripps, David. *An Implementation Guide to Real-Time Programming.* Englewood Cliffs, New Jersey: Yourdon Press,1989.

[Schilling 93]   Schilling, Jonathan. *Monitor (Passive) Task Optimization.* Personal communication. October 1993.

[Sha 89]   Sha, Lui; & Goodenough, John. *Real-Time Scheduling Theory and Ada.* (CMU/SEI-89-TR-14,ADA211397). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University,1989.

[Silberschatz 91] Silberschatz, Abraham; Peterson, James; & Galvin, Peter. *Operating System Concepts.* Reading, MA.: Addison-Wesley, 1991.

[Wegner 92]   Wegner, Peter. *Design Issues for Object-Based Concurrency.* Lecture Notes in Computer Science # 612.Berlin, New York: Springer Verlag 1992.

[Xu 93]   Xu,Jia; & Parnas, David. *On Satisfying Timing Constraints in Hard-Real-Time Systems.* IEEE Transactions on Software Engineering, 19.1, Jan. 93.

Appendix A   Taxonomy

This appendix represents the taxonomy of the coordination mechanisms analyzed.

In some mechanisms there are features characterized To Be Determined (TBD). The reason is that the realization of this feature is implementation dependent so it is responsibility of the coordination mechanism developer to implement the particular feature.

When a feature is not applicable in a particular coordination mechanism, it is indicated by the N/A acronym.

	Barrier	Blackboard	B_ Buffer	Broadcast	Events
Identification	indirect	indirect	indirect	indirect	indirect
Synchronization					
Exclusion Constraints					
Internal State	no	m.enabling	s. enabling	no	m. enabling
Synchronization State	m. enabling	no	no	no	no
History					
Latched	no	no	no	no	no
Not Latched	no	no	no	m. enabling	no
Ordering Constraints					
Priority	no	TBD	no	TBD	no
Time of Arrival	no	TBD	yes	TBD	no
Communication					
Storage Capacity	null	bounded	bounded	bounded	bounded
Direction of Flow	N/A	unidirectional	unidirectional	unidirectional	unidirectional
Transmitted Information					
None	yes	no	no	no	no
Unitary	no	no	no	no	yes
Multiple	no	fixed length	fixed length	fixed length	no
Implementation					
OS Dependency					
OS Services	no	no	yes	no	yes
External Library	yes	yes	yes	yes	yes
Ada	no	no	yes	no	no
Information Passing Mechanism	N/A	copy	copy	copy	copy
Threads of Control	TBD	TBD	passive	TBD	TBD

	G.Semaphore	Lock	Mailbox	Pulse	Relay
Identification	indirect	indirect	indirect	indirect	indirect
Synchronization					
Exclusion Constraints					
Internal State	s. enabling	s. enabling	no	no	no
Synchronization State	no	no	no	no	no
History					
Latched	no	no	s.enabling	no	s. enabling
Not Latched	no	no	no	m.enabling	no
Ordering Constraints					
Priority	yes	no	yes	TBD	no
Time of Arrival	yes	yes	yes	TBD	yes
Communication					
Storage Capacity	null	null	null	null	bounded
Direction of Flow	N/A	N/A	unidirectional	N/A	unidirectional
Transmitted Information					
None	yes	yes	no	yes	no
Unitary	no	no	no	no	no
Multiple	no	no	var. length	no	fixed length
Implementation					
OS Dependency					
OS Services	yes	no	yes	no	no
External Library	no	yes	no	yes	no
Ada	no	yes	no	no	yes
Information Passing Mechanism	N/A	N/A	copy	N/A	copy
Threads of Control	TBD	TBD	TBD	TBD	TBD

	Rendezvous	Signal	T_Buffer	Transporter
Identification	indirect	indirect	indirect	indirect
Synchronization				
Exclusion Constraints				
Internal State	no	no	s. enabling	no
Synchronization State	no	no	no	no
History				
Latched	s. enabling	no	no	s. enabling
Not Latched	no	s. enabling	no	no
Ordering Constraints				
Priority	no	N/A	no	no
Time of Arrival	yes	N/A	yes	yes
Communication				
Storage Capacity	null	null	bounded	bounded
Direction of Flow	bidirectional	unidirectional	unidirectional	unidirectional
Transmitted Information				
None	no	no	no	no
Unitary	no	yes	no	no
Multiple	fixed length	no	yes	fixed length
Implementation				
OS Dependency				
OS Services	no	yes	no	no
External Library	no	yes	no	no
Ada	yes	no	yes	yes
Information Passing Mechanism	copy	N/A	copy	copy
Threads of Control	TBD	active	TBD	TBD

Appendix B   Coordination Mechanisms Catalog

The Coordination Mechanisms Catalog represents the coordination mechanisms found in the various real-time executives, pure Ada applications, or extended libraries analyzed during the domain analysis activity.

In the feature catalog table the following conventions are used

1. Name: This is the name used for this coordination mechanism.

2. "Yes/No": This coordination mechanism does/does not exist on this system.

3. <TBD>: No information was collected for this coodination mechanism.

The systems analyzed and represented in the catalog are the following:

Ada: Here are represented those systems using coordination mechanisms implemented in Ada. Examples are found elsewhere [NSWC 88], [Nielsen 88].

ARINC: The primary objective of the document analyzed is to define a general purpose interface between the operating system of an integrated avionics core processor module and the application software [ARINC 93].

CIFO: The document proposes a common set of user-runtime environment interfaces from a user's perspective [ARTEWG 93].

EXTRA: The purpose of the document is to define a standard Ada library for hard real-time (HRT) to support application portability at the source level [RRG 93].

MTOS-UX: It represents a commercial runtime support executive. The MTOS-UX/Ada is an operating system that can handle multiprocessing for real-time and the Ada language [Industrial 88].

	Ada	ARINC	CIFO	EXTRA	MTOS-UX
Coordination Mech.					
Barrier	no	no	yes	yes	no
Blackboard	no	no	yes	yes	no
Bounded_Buffer	yes	yes	yes	yes	TBD
Broadcast	no	no	yes	yes	no
Event	TBD	yes	yes	yes	yes
General_Semaphore	TBD	yes	no	no	yes
Lock	yes	no	yes	TBD	no
Mailbox	no	no	no	no	yes
Pulse	no	no	yes	yes	no
Relay	yes	no	no	no	no
Rendezvous	yes	TBD	yes	yes	yes
Signal	TBD	no	yes	yes	yes
Timed_Buffer	yes	no	no	no	no
Transporter	yes	no	no	no	no

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS None			
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-93-TR-34		5. MONITORING ORGANIZATION REPORT NUMBER(S) ESC-TR-93-321			
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute	6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office			
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213		7b. ADDRESS (city, state, and zip code) HQ ESC/ENS 5 Eglin Street Hanscom AFB, MA 01731-2116			
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office	8b. OFFICE SYMBOL (if applicable) ESC/ENS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003			
8c. ADDRESS (city, state, and zip code)) Carnegie Mellon University Pittsburgh PA 15213		10. SOURCE OF FUNDING NOS.			
		PROGRAM ELEMENT NO 63756E	PROJECT NO. N/A	TASK NO N/A	WORK UNIT NO. N/A

11. TITLE (Include Security Classification)
A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis

12. PERSONAL AUTHOR(S) Jose L. Fernandez

13a. TYPE OF REPORT Final	13b. TIME COVERED FROM TO	14. DATE OF REPORT (year, month, day) December 1993	15. PAGE COUNT 58

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
FIELD	GROUP	SUB. GR.	taxonomy, coordination mechanisms, concurrent processes, real-time
			software architectures, domain analysis

19. ABSTRACT (continue on reverse if necessary and identify by block number)

A taxonomy of the coordination mechanisms for the synchronization and communication of concurrent processes is proposed. The taxonomy deals with the issues of a real-time software architecture that are application domain independent. The taxonomy will help the designer to find the appropriate coordination mechanisms for building a real-time domain specific software architecture. Features Oriented Domain Analysis methodology has been used to describe the taxonomy. While Ada is the programming language that has been used here, some of the attributes and guidelines are still valid for other programming languages.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ■ SAME AS RPT □ DTIC USERS ■	21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution	
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF	22b. TELEPHONE NUMBER (include area code) (412) 268-7631	22c. OFFICE SYMBOL ESC/ENS (SEI)

DD FORM 1473, 83 APR   EDITION of 1 JAN 73 IS OBSOLETE

ABSTRACT — continued from page one, block 19