THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

OBJECT-BASED COMPUTER SYSTEMS AND THE ADA PROGRAMMING LANGUAGE

G. D. Buzzard
T. N. Mudge

CRL-TR-29-83

AUGUST 1983

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1]Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

engn
UHR1093

OBJECT-BASED COMPUTER SYSTEMS AND THE ADA PROGRAMMING LANGUAGE

by

G. D. Buzzard and T. N. Mudge
Computer Research Laboratory and Robotics Research Laboratory
Department of Electrical and Computer Engineering
University of Michigan
Ann Arbor, MI 48109

This work was supported in part by the Air Force Office of Scientific Research under contract F49620-82-C-0089 and a Kodak Fellowship.

ABSTRACT

This paper gives an introduction to object-based computer systems. In particular, it is shown how they can support the Ada programming language. A case study using the Intel iAPX432 is given.

I. INTRODUCTION

There has been a growing interest in object-based computer systems recently. Much of this is attributable to the Department of Defense's (DoD) massive commitment to the Ada® language project. Although Ada (DoD's proposed standard system implementation language) may not fit everyone's definition of an object-based language, it does incorporate key object-based concepts. In this discussion we will explain several concepts that are central to object-based computer systems and the Ada programming language. Computer architectures and the design implications for such systems are discussed, and examples are given which draw from our use of Ada in programming a robot-based manufacturing cell [1, 2].

In software systems the term object-based describes software environments which incorporate the concepts of data abstraction, program abstraction [3] and protection domains [4] through the use of "objects." Objects are individually addressable entities that uniquely identify their own contents. For example, in Intel's implementation of Ada for the iAPX 432 [5,6,7], objects form the basis for Ada packages that provide abstractions for either programs or data.

In hardware, the term object-based is normally used to refer to the architectural support provided for data abstraction, program abstraction and protection domains. This type of support is exemplified by an architecture where the primitive operations for memory management, process dispatching, interprocess communication, or other operating system features are provided by the hardware. In such systems the implementation details of the memory pool, process dispatching, or interprocess communication mechanisms are hidden, and a concise interface in the form of instructions which operate on the objects corresponding to the respective mechanisms is presented to the software operating system.

®Ada is a registered trademark of the Ada Joint Program Office-DoD.

Also in hardware, the term object-based is sometimes used to describe subsystems in which abstract data types have been directly implemented. This idea has significant implications for the future when design automation tools become more sophisticated. One could envision CAD systems capable of economically customizing individual batches of VLSI processors for specific operations which appear unusually often in a given application. This concept is closely related to the hardware/software transparency issue discussed below.

There are two major goals in developing object-based software. The first is to reduce the total life-cycle software cost by increasing programmer productivity and reducing maintenance costs. Both increases in programmer productivity and reductions in maintenance costs are aided by the object-based modularization promoted by Ada. The second goal is to implement software systems which resist both accidental and malicious corruption attempts. Protection domains are used for this purpose.

The major goal in developing object-based hardware is to provide an efficient execution environment for the software system. A future extension to this goal is to design the system so as to be able to extend the abstraction mechanism to obtain hardware/software (HW/SW) transparency [2,8]. The availability of HW/SW transparency would allow system designs to proceed without regard for the final placement of the HW/SW boundary. The entire system could be described by a suitable system implementation language and implemented in software as far as physically possible. Then, after appropriate performance analyses, selected modules, which would typically realize abstract data types, could be shifted to a direct hardware implementation. If the design methodology was implemented in a development environment with sufficient sophistication the migration from software to hardware would not require any code to be rewritten and the interface would already be specified. The hardware interface could occur at several levels: bus level, similar to current arithmetic coprocessors such as the Intel 8087; memory (I/O) level, like many existing I/O devices; or, conceivably, at a level internal to the CPU involving an

actual change in the CPU architecture.

There are a few commercially available computer systems which incorporate various object-based concepts. These include the IBM System/38 [9], the Intel iAPX 432 [5], and the PP250 [10]. The PP250, designed by Plessey Telecommunications Research Laboratories for applications in telephone switching systems, was the first commercially available machine to incorporate many object-based concepts. For use in telephone switching systems it had to meet very stringent reliability standards requiring the inclusion of software error detection and recovery facilities. These requirements were met through the use of a capability mechanism [11]. Many of the ideas incorporated in the commercial systems were based on the results of several university projects. The most recent of these projects include Cm* [12], C.mmp [13] and CAP [14]. A common thread running through all of these machines is the use of capability addressing techniques to implement secure protection domains. These protection domains (packages in Ada) can then be appropriately structured to provide data and program abstractions.

Object-based machines are particularly well suited to applications which have stringent requirements for data security and program integrity. The high degree of abstraction provided by the architecture also facilitates the interconnection of several processors into either tightly coupled multiprocessor systems and/or distributed networks. For example, through the use of process/processor abstraction Intel has achieved software transparent multiprocessing in their iAPX 432 system. In addition, the Cm* system provides an example employing both tightly coupled multiprocessing and distributed networking concepts in one system.

As might be expected, all of the benefits of an object-based system do not come free of charge. Present systems rely on some form of capability addressing. In current implementations these addressing mechanisms greatly increase the address generation and translation times, even when translation look-aside and caching

schemes are employed. For example, to copy a capability requires ten memory references on Cm* [15], and between two and twelve on the iAPX 432 [5].

The following section discusses key "object" concepts and their implementation in Ada. Section III is a case study that describes the architectural support for object-based concepts in the Intel iAPX 432. Finally, comments on areas of further research and concluding remarks are made in Section IV.

II. OBJECT-BASED CONCEPTS

Abstraction plays a central role in the object-based design methodology, particularly data and program abstraction. Of the two, data abstraction is the most widely used and understood.

Data Abstraction. Shaw gives the following definition of an abstract data type [3]:

> In most languages that provide the facility, the definition of an abstract data type consists of a program unit that includes the following information:
>
> - *Visible outside the type definition:* the name of the type and the names and routine headers of all operations (procedures and functions) that are permitted to use the representation of the type; some languages (e.g., Ada--our insert) also include formal specifications of the values that variables of this type may assume and of the properties of the operations.
>
> - *Not visible outside the type definition:* the representation of the type in terms of built-in data types or other defined types, the bodies of the visible routines, and hidden routines that may be called only from within the module.

The constructs for implementing abstract data types in Ada are "packages" and "private" (hidden) types [16]. The Ada package effectively places a wall around a group of declarations and permits access only to those declarations which are intended to be visible. Packages actually come in two parts, the specification and the body. The package specification formally specifies the abstract data type and its interface to the outside world, along with other information which may be necessary to enforce type consistency across separate compilation boundaries. The body of the package contains the hidden implementation details. The relationship between Ada packages and objects (in the context of the iAPX 432) will be discussed in the next section.

Consider, for example, the Ada package ROBOT (the specification for which is shown in Figure 1). The specification is delimited by "**package** ROBOT **is**" and "**end** ROBOT;". Ada reserved words are shown in bold lower case. User defined and prede-

fined package names, procedure names, function names, types and variables are shown in upper case.

The types defined in the specification are as follows. ROBOT_ARM--the abstracted data type. JOINT_TYPE--an enumeration type that defines joints as either rotating or sliding. LINK_REP--a record that specifies the translational and rotational relationship between adjacent links of an arm, and the type of joint at one end of the link. ARM_REP--an array of links that make up the robot arm (the exact number is declared when an instance of the type ROBOT_ARM is created). ARM--a pointer to ARM_REP (*access* type in Ada). The pointer is necessary only because Ada does not allow unconstrained arrays as part of records (see the ROBOT_ARM record in Figure 1). FRAME--a 4x4 matrix that represents a homogeneous transformation. It can represent the position and orientation of the hand of the robot arm by indicating the matrix necessary to transform the coordinate system of the base of the arm to the coordinate system in the hand. GRASPED--a boolean type to indicate whether the hand's gripper is closed and has grasped something.

The procedures are as follows. SET_ATTRIBUTES--this creates an instance of type ROBOT_ARM. The output of the procedure is of type ROBOT_ARM and the input is of type ARM. The actual parameter that is substituted for the formal parameter ATTRIBS carries the values that define the links of a specific arm. MOVE--takes as input an arm and a transformation and moves the arm and its hand to the position and orientation corresponding to the transformation. The arm is output. Its STATE has been changed to reflect its new position and orientation. OPEN and CLOSE open and close an arm's gripper. GET_FRAME is a function that returns an arm's position and orientation.

As already noted, the abstracted data type is ROBOT_ARM. It is the intent of the package ROBOT that this type be known only through the procedures and the function mentioned above and declared in the visible part of the package specifica-

```
package ROBOT is

        type ROBOT_ARM is limited private;
        type JOINT_TYPE is (REVOLUTE, PRISMATIC);

        type LINK_REP is
                record
                        JOINT: JOINT_TYPE;
                        JOINT_ANGLE: FLOAT;
                        JOINT_LENGTH: FLOAT;
                        OFFSET_ANGLE: FLOAT;
                        OFFSET_LENGTH: FLOAT;
                end record;

        type ARM_REP is array (INTEGER range <>) of LINK_REP;
        type ARM is access ARM_REP;

        type FRAME is array (1..4, 1..4) of FLOAT;
        type GRASPED is new BOOLEAN;

        procedure SET_ATTRIBUTES (X: out ROBOT_ARM; ATTRIBS: in ARM);
        procedure MOVE (X: in out ROBOT_ARM; DESTINATION: in FRAME);
        procedure OPEN (X: in out ROBOT_ARM);
        procedure CLOSE (X: in out ROBOT_ARM);
        function GET_FRAME (X: ROBOT_ARM) return FRAME;

private
        type ROBOT_ARM is
                record
                        RBT: ARM;
                        STATE: FRAME;
                        GRIPPED: GRASPED := FALSE;
                end record;

end ROBOT;
```

Figure 1. Specification for the Package ROBOT.

tion (the visible or public part of the package specification extends up to the reserved word "private"). Also abstracted, or hidden from public view, are all of the other procedures, functions and data structures which are necessary to actually effect the movement of the physical robot arm and to update its defining data structure. These other abstracted procedures, functions and data structures are located within the package body. The fact that the type ROBOT_ARM is declared to be

"limited private" and that its definition is given in the private part of the package specification, means that while routines in packages external to ROBOT may possess an object of the type ROBOT_ARM, they cannot use it in any way other than as a parameter to pass to one of the routines defined in the visible part of the ROBOT package specification. Even tests for equality between two objects of type ROBOT_ARM are not allowed outside of the ROBOT package. Hence, possibilities for programming errors that directly affect the ROBOT_ARM are restricted to the domain defined by the package ROBOT. Figure 2 illustrates how the package ROBOT is used to move an ASEA robot or a PUMA robot. The ASEA or the PUMA (both of type ROBOT_ARM) are manipulated by the procedure MOVE within ROBOT. The movement is defined by TRANS of type FRAME.

The data structures, excluding those that are explicitly declared "private" or "limited private", along with the procedures and functions which appear in the public part of the package specification are directly available for use by other external packages. In fact, in this example it is strictly necessary that external packages have access to all of the "public" data structures. Access to the FRAME and GRASPED types are necessary to provide destination coordinate and gripper information to the robot. Access to ARM (and hence ARM_REP), LINK_REP and JOINT_TYPE (and, of course FLOAT, which is predefined in the language) are required in order for the user to set the attributes of the specific robot that is being used. As already mentioned, attributes are set with the SET_ATTRIBUTE procedure which takes a ROBOT_ARM and assigns to it the attributes specified indirectly by an ARM type (i.e., by an ARM_REP type). Note that even though external packages have access to the types ARM, FRAME or GRASPED they still cannot directly (i.e., without the use of the SET_ATTRIBUTE procedure) manipulate them if they are components of another type to which the external packages do not have access. This prevents external packages from being able to directly manipulate components of the ROBOT_ARM record.

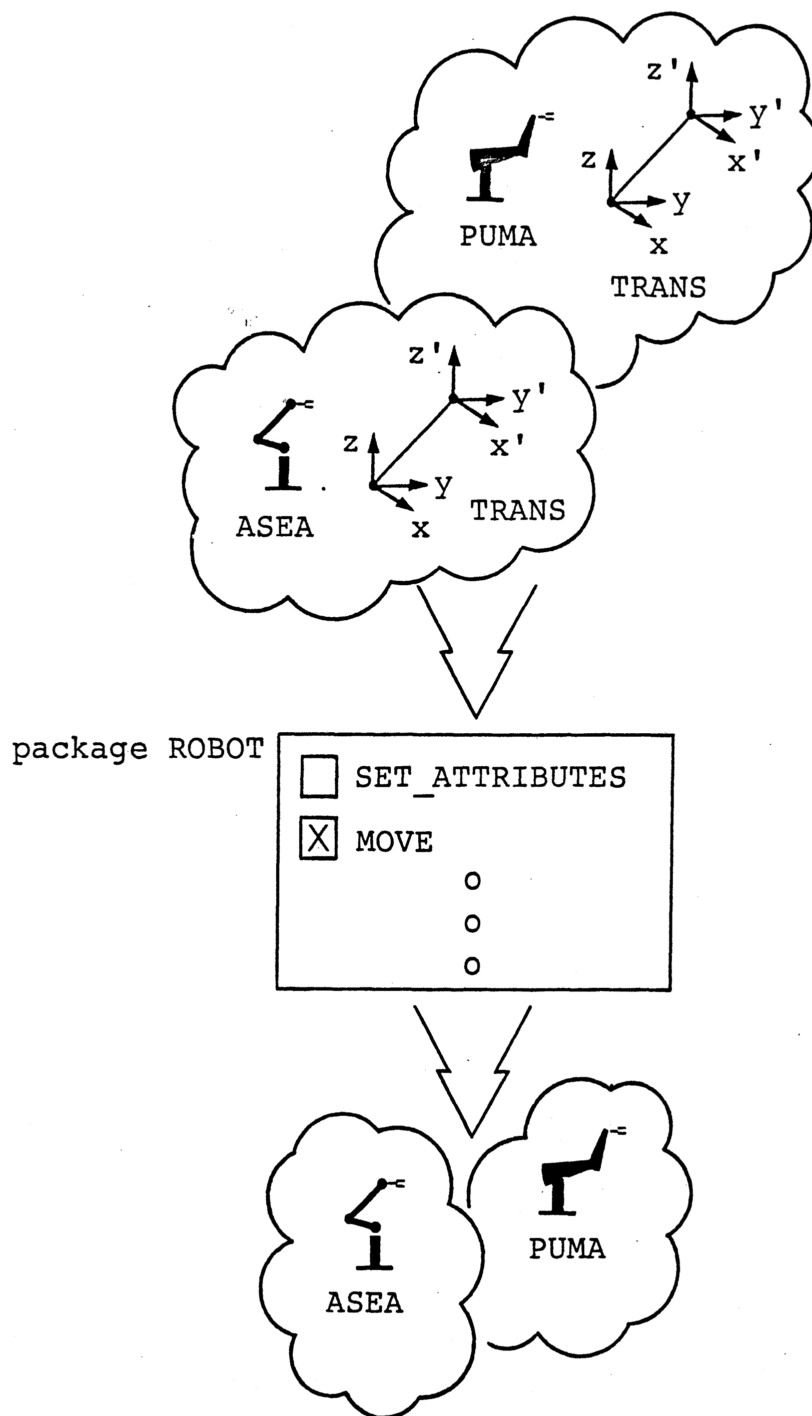


Figure 2. Data Abstraction.

Program Abstraction. Programs and subprograms provide another common level of abstraction. Program abstraction provides operations on implicit objects. In other

words, in addition to hiding the representation of, and access to, the objects, even the existence of the objects are hidden. This provides a more complete form of hiding and, usually, a more concise interface than data abstraction, since the user is prevented from even possessing an object of the abstracted type.

Program abstraction in Ada is realized through generic package instantiation. The generic package is really a template for packages which will accept abstractions as actual parameters. These parameters may be either data types or subprograms (functions or procedures). This represents a slightly higher level of abstraction than data abstraction, because the object(s) being manipulated is(are) completely hidden within the package body. The hidden object structure is accessed through the internal package variables that are non-local to the subprograms in the package. Manipulation of the object structure occurs as a controlled side effect--which is strictly contained within the package body--of the requested operation. In this manner generic program abstraction supports an environment in which the specified (public) operations either directly or indirectly transform a hidden internal state which depends only on past operations applied to the initial state of the system.

The ROBOT package example is repeated in Figure 3 using program abstraction techniques. It can be seen that the interface is more concise in this example. The external packages are no longer required to specify the attributes of the given robot arm that they wish to use. Instead, they just specify the name of the arm that they wish to use when they instantiate the generic package ROBOT. This allows the elimination of the procedure SET_ATTRIBUTES and all of the types except FRAME from the package specification. The proper attributes are automatically assigned to the instantiated generic package based upon the value of the generic formal parameter ARM at the time of instantiation. Hence, the statements:

```
package MY_ASEA is new ROBOT (ASEA);
package MY_PUMA is new ROBOT (PUMA);
```

```
type ARM_MODEL is (ASEA, PUMA, ... );
        •
        •
        •

generic
        ARM: ARM_MODEL;
package ROBOT is

        type FRAME is array (1..4, 1..4) of FLOAT;

        procedure MOVE (DESTINATION: in FRAME);
        procedure OPEN;
        procedure CLOSE;
        function GET_FRAME return FRAME;

end ROBOT;
```

Figure 3. Specification for the Generic Package ROBOT.

would create instances of the package ROBOT specifically for an ASEA robot and a PUMA robot, respectively. Such packages would then be instantiated in a one-to-one correspondence with all of the active robots in a given system. Therefore, it is no longer necessary to pass a parameter representing the specific robot arm (type ROBOT_ARM in the previous example) to the procedures which will be manipulating it. This is illustrated in Figure 4. Adding new models of robots entails adding values to ARM_MODEL and making the corresponding changes in the generic body of ROBOT. ARM_MODEL is an enumeration type visible within the instantiating program unit.

The attributes for all arms listed in the enumeration type ARM_MODEL are now hidden from the user within the package body ROBOT. These attributes completely specify all of the interlink relationships. It is required, however, that certain procedure bodies contain conditionally executed code segments to account for the differences between number and type of links used in the different robots, and also to allow individual robots to communicate with their respective device drivers. An alternative to the conditionally executed code segments would entail instantiating

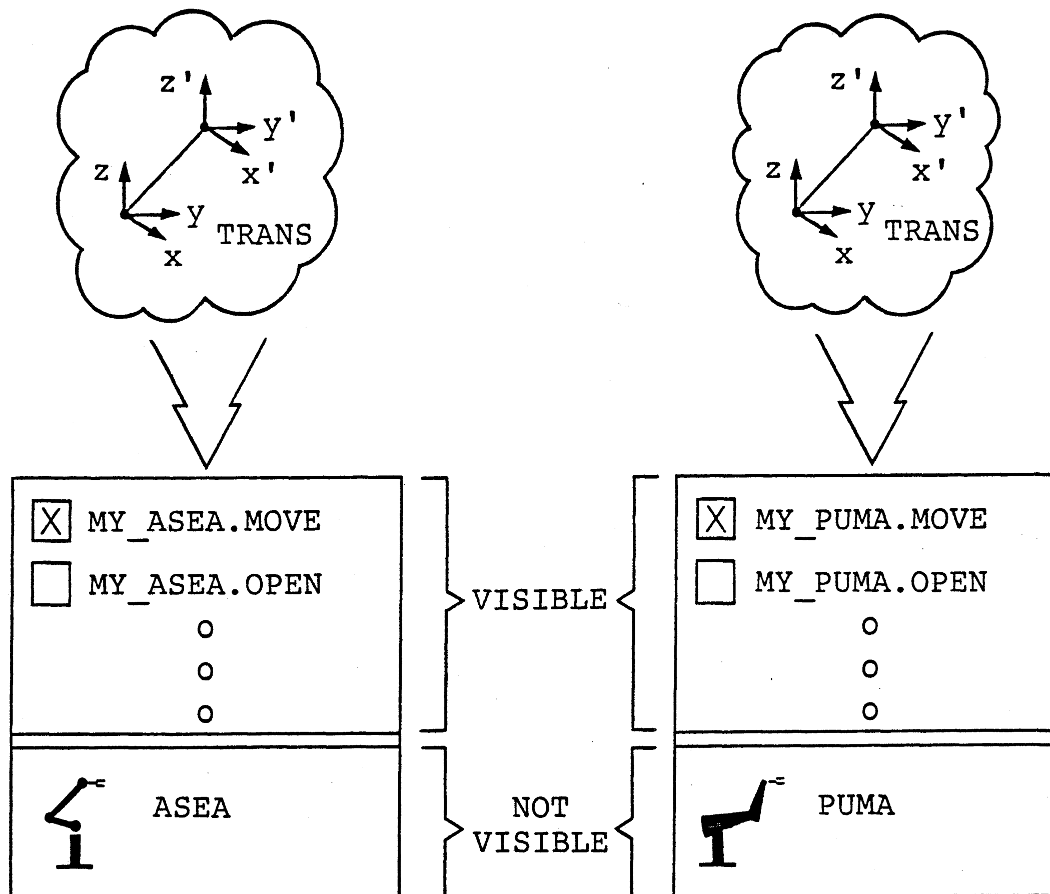


Figure 4. Program Abstraction.

the generic package with subprograms (procedures or functions) as actual parameters.

Intel, through one of their extensions to the Ada language, have provided a powerful construct for use in such situations. This has been accomplished by allowing packages to be types, and hence, allowing them to assume values. For example, we could define a package type ROBOT (Figure 5), create instances of ROBOT for each different physical robot (Figure 6), declare a variable MY_ROBOT of package type ROBOT, and then, during program execution assign a value (package body) to the variable depending upon which robot we chose to use (see Figure 7). The

procedures and functions which operate on MY_ROBOT are invoked in the manner shown in the last line of Figure 7.

In Smalltalk, a language that takes the object-based programming philosophy further than Ada, the concepts of data and program abstraction have been rationalized so that objects are all treated alike regardless of whether the objects represent program modules or data structures [17]. It has been proposed that these concepts

---

```
package type ROBOT is

        type FRAME is array (1..4, 1..4) of FLOAT;

        procedure MOVE (DESTINATION: in FRAME);
        procedure OPEN;
        procedure CLOSE;
        function GET_FRAME return FRAME;

end ROBOT;
```

Figure 5. A Package Type.

```
package ASEA is constant ROBOT;

package body ASEA is
            •
            •
            •
end ASEA;

package PUMA is constant ROBOT;

package body PUMA is
            •
            •
            •
end PUMA;
```

Figure 6.

```
MY_ROBOT: ROBOT;
NEW_POSITION: FRAME;

case ARM_TYPE is
      when ASEA_ARM      => MY_ROBOT := ASEA;
      when PUMA_ARM      => MY_ROBOT := PUMA;
                    •
                    •
                    •
end case;

MY_ROBOT.MOVE(NEW_POSITION);
```

Figure 7.

be merged together in Ada as well [18]. In fact, Intel has already taken a big step in this direction with their "package type" extension to the Ada language mentioned above [19]. The merging of program and data abstraction concepts results in a common abstraction mechanism. The software designer is then relieved of the artificial choice between program-oriented or data-oriented programming methodologies.

Protection Domains. Protection domains, and the inherent security that they provide, are another key object concept. The basis for secure and error-tolerant execution environments lies in the principle of system closure [4]. This principle basically states that the effects of all operations on a closed system shall remain strictly within that system. One common construct used for providing system closure is the protection domain [20]. Briefly stated, a protection domain is an environment or context that defines the set of access rights that are currently available to a specific user for objects of the system. This concept maps very well on to Ada packages. Capability based addressing schemes are the most efficient known mechanism for implementing protection domains.

Protection domain schemes generally provide facilities for error confinement, error detection and categorization, reconfiguration, and restarting. Error confinement

and security strategies generally involve both process isolation and resource control. The basic premise of process isolation is that processes are given only the capabilities necessary to complete their required tasks. This implies that interactions with any external objects (e.g., sending messages to other processes) must be strictly formalized and controlled. Resource control refers to the binding of physical resource units to computational objects. Examples of this include the binding of processes to processors, or the assignment of memory to currently executing contexts. The idea here is to ensure that when the resource units are released, or preempted, that all information contained within the unit is returned to a null state. This prevents any information from "leaking" out of a protection domain by being innocently left in an area that will eventually become accessible to other users. Error confinement also aids the program debugging process since bugs will be located in the module in which the error is detected. Program maintenance also benefits since the protection domain defines the maximal set of modules which could be affected by a modification to the system. Error detection and categorization involves dynamic checking for object type inconsistencies and access constraint violations during procedure execution. The categorization of detected errors can then be used to aid in restoring the system to a known consistent state. Reconfiguration facilities attempt to restore the system to an operable state by removing from service the failed component, be it hardware or software. If the reconfiguration attempt is successful the system is then restarted.

The most efficient known mechanism for implementing protection domains is the capability mechanism. While much can be done at compile time to enforce the concepts of protection domains, there are many cases where a dynamic enforcement mechanism is essential. The real-time sharing of data between programs provides an obvious example. But compile-time protection enforcement also lacks the ability to support the detection of, and recovery from, failures in the run-time system.

A capability can be thought of as the name of an object. An object cannot be accessed--in fact, its existence cannot even be determined--unless its name is possessed. The capability also contains the access rights to the object (e.g., read, write, or capability copy rights, see Figure 8). The only subsequent modification allowed outside of the creating context is the restriction of these rights. Capabilities are created along with their respective objects. The initial control of the capability, hence the object, belongs solely to the creating context. Consider the case of a user--package USER in Figure 8--that uses the package ROBOT of Figure 1. It is the creating context for a variable of type ROBOT_ARM (it creates the variable with a call to SET_ATTRIBUTES). It restricts its own rights to "copy" because ROBOT_ARM is

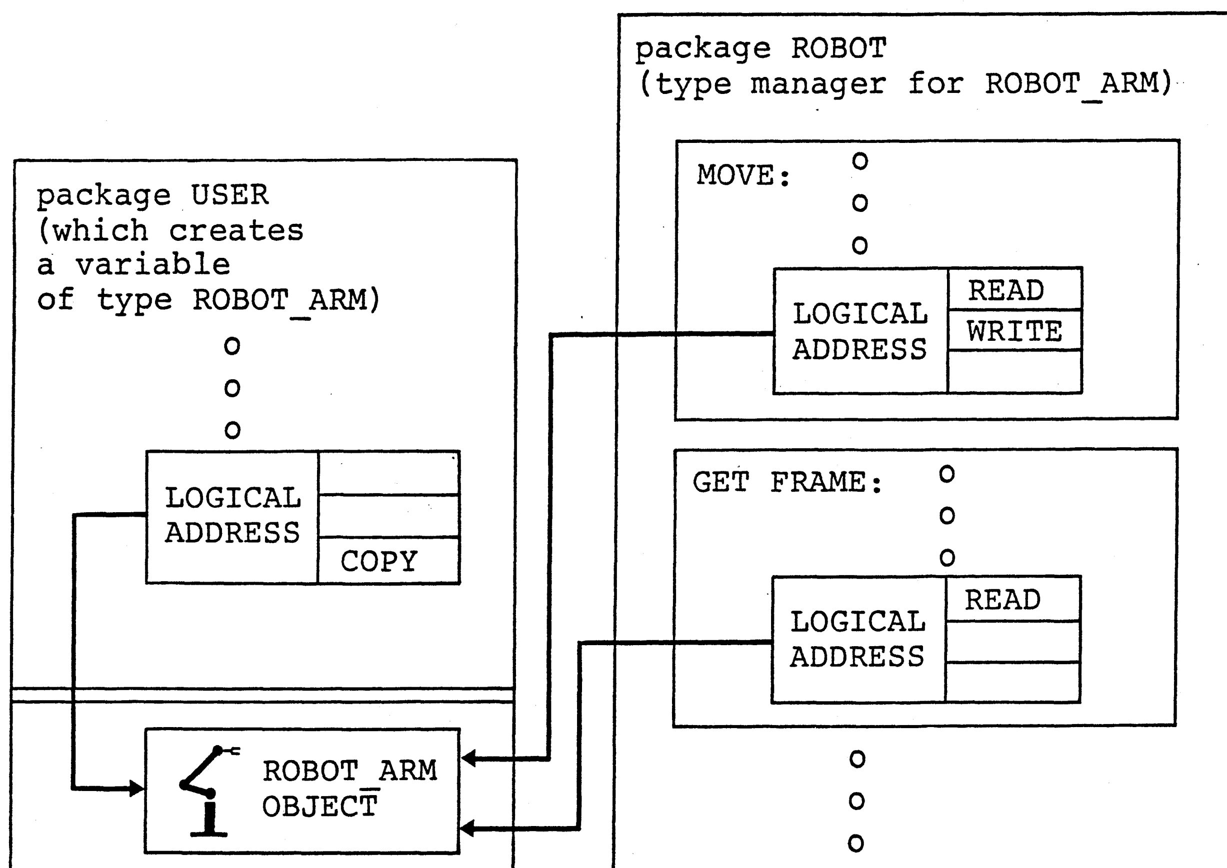


Figure 8. Capabilities and Protection Domains.

a limited private type. However, this does not prevent package USER from providing capabilities with appropriate additional rights since it is the creating context. For example, a variable of type ROBOT_ARM is passed as a parameter to both the MOVE and GET_FRAME procedures. In the case of MOVE, ROBOT_ARM is both an "in" and "out" parameter requiring read and write rights. In the case of GET_FRAME, ROBOT_ARM is an "in" parameter, hence it requires only read rights. Specific implementation details are given in the case study.

III. CASE STUDY

The Intel iAPX 432 is an object-based microcomputer system which provides architectural support for many of the concepts discussed above. Specifically, the mechanisms used to support the concepts of data abstraction, program abstraction, and protection domains, as well as interprocessor communication and transparent multiprocessing, are provided by the hardware/microcode.

Objects and Type Checking. All information in the iAPX 432 system is represented by typed objects. An object is defined by the following four characteristics [21]. First, an object is a data structure containing organized information. Second, objects also define the set of operations which may be performed on themselves. In fact, these are the *only* operations that are allowed. Third, iAPX 432 objects are referenced as a single entity regardless of the length of the object. Finally, every object has a unique label that contains the information about its type. The first characteristic is, in general, rather restrictive. Consider the case of I/O for example. The iAPX 432 avoids this restriction by not directly handling its own I/O. I/O responsibilities are left to an attached processor subsystem which partially operates in the iAPX 432 general data processor's (GDP) memory space and communicates with the GDP's via a hardware implemented message passing scheme. If this were not the case, the above definition of objects would have to include "pseudo data structures" which would actually provide the description of a physical device interface.

Objects are implemented as a collection of one or more segments. Segments are variable length groups of contiguous memory locations with two distinct parts, data and access. Either part, data or access, may be null. Segment rights are identified by header information which is stored in the segment itself, but which is invisible to the software. Type information, along with the respective sizes of the access and data parts, is stored in the storage descriptors of each individual object. The access

part of a segment can contain only access descriptors or null entries. Access descriptors are the "capabilities" in the iAPX 432 system. The data parts contain all of the other information in the system, including things such as instructions or process status information.

The ability of the hardware to identify access segments is one of the key mechanisms used by the iAPX 432 to enforce protection domain security. Any attempts to modify access segments can be closely monitored. Further, data references in the instruction stream always use access selectors to choose an access descriptor available within the currently executing context to gain access to the desired object. This limits the set of access descriptors that a particular context can possess, hence controlling the scope of accessible objects. In fact, there are no iAPX 432 general data processor machine instructions which allow data to be referenced in any other manner, including by physical address. Hence, it is impossible for a process executing in a given context to corrupt, either maliciously or inadvertently, any system data for which it has not explicitly been given a capability (access descriptor).

The security mechanisms inherent in the iAPX 432 architecture are not limited to the above, however. As an example we will consider the protection against executing data. The currently executing process references instructions via two indices, one of which is an instruction pointer that provides an offset into the current instruction object. The other is an index into the domain object access segment (described later) which selects a capability that determines the current instruction object. Since the physical base address of the currently executing instruction object is cached on chip and since length bounds checks are automatically performed by the hardware on all memory references, instruction fetching type consistency is guaranteed by checking the object type of the instruction object referenced by the indexed capability. This needs to be done only when the current instruction object index is modified, that is only when an intersegment branch is executed.

There are a number of hardware/microcode recognized system objects in the iAPX 432 which support many of the object-based concepts discussed in Section II. This includes objects which represent instructions, protection domains, activation records, processes, and even physical processors. The routines which manipulate these objects are primarily located in the microcode. Some of these routines are available to the user, in the form of machine level instructions, others remain below the user interface and are invoked indirectly by mechanisms such as process dispatch.

Two key objects are the context and domain objects; these represent activation records and protection domains respectively. The domain objects and context objects are used to realize the concept of type managers (see Figure 8), the system's primary mechanism for implementing data and program abstraction concepts. Type managers are modules that provide information hiding by containing a data structure (in the case of program abstraction), or a type definition of one (in the case of data abstraction), and all the necessary procedures to manipulate that data structure. By allowing only a tightly controlled subset of these procedures to be invoked from outside of the protection domain (i.e., the domain object), the implementation of the data structure and the procedures which directly manipulate it are effectively "hidden" from the outside world.

The domain object represents the static structure of a type manager. It is the root node of a tree-like structure of objects which includes all of the instruction and static data objects that are contained within the given Ada package. Consider, for example, Figure 9 which shows the domain object and sub-objects corresponding to the Ada package shown in Figures 1 and 2. Notice that all of the "public" information is accessible from contiguous locations within the domain object. The iAPX 432 architecture provides refinements which are subsets of objects. The object is not modified in any way; a new access descriptor is created (possibly with more restricted rights) which provides access to a contiguous subset of memory locations

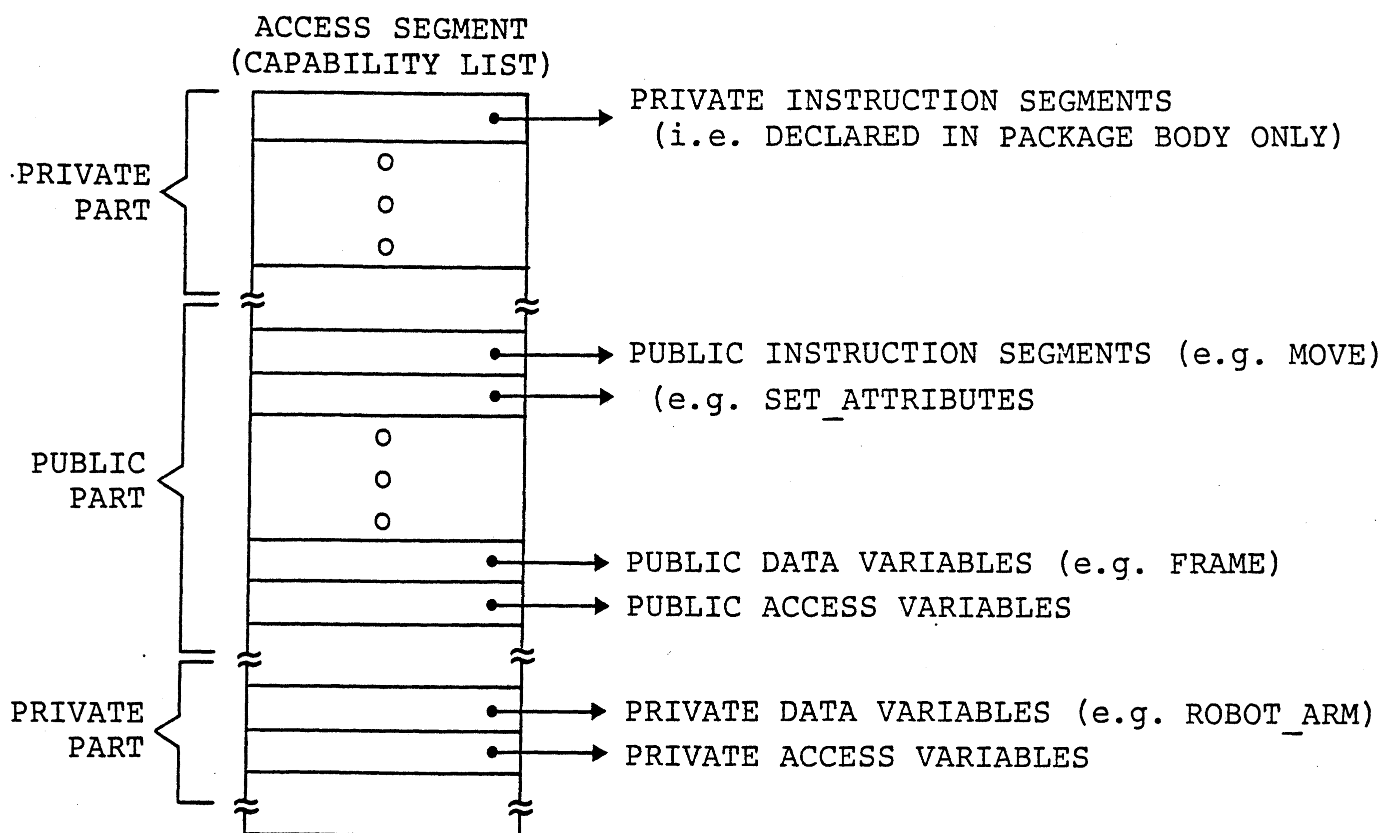


Figure 9. A Domain Object.

within the object. Capabilities for refinements are then given to external routines so that they may access the "public" routines and data structures. The private routines and data structures remain inaccessible outside of the domain and, hence, realize the data or program abstraction.

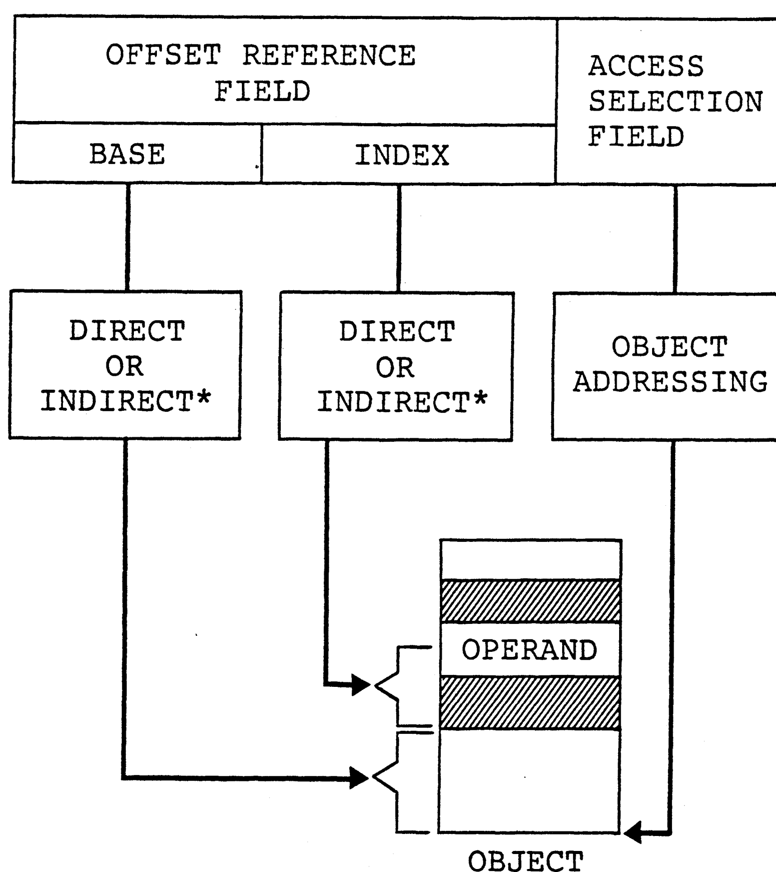
The context object in the iAPX 432 contains the dynamic run-time information which describes the execution environment of an invoked procedure, much as a stack frame does for VAX-like architectures. The information in the context object includes the current domain of definition, the indices that make up the instruction pointer, the previous context, the operand stack and stack pointer, the capability addressing environments and message objects (see [5]). Every activated procedure in the sys-

tem has at least one context object associated with it. When a procedure is called, a "new" context object is automatically allocated with a return link to the "old" one stored in it; conversely, when a procedure returns to the "old" context the "new" context object is automatically reclaimed. In the interest of run-time efficiency contexts are created and destroyed in groups, and allocated and reclaimed as needed.

The iAPX 432 architecture supports dynamic type checking. System objects have type information embedded in their storage descriptors which also contain physical addressing information. User defined types have associated with them a type definition object that defines the corresponding type manager for each user defined type. A pointer to the type definition object is found in the storage descriptors of user defined objects. This, coupled with the support provided by system objects, especially domain and context objects, allows the iAPX 432 to easily provide support for dynamic package types. Furthermore, these package types can easily be passed as parameters (messages) to other routines. An example of the utility of package types was provided earlier in the ROBOT example (Figures 5, 6 and 7). Garbage collection provides another example. Garbage collection algorithms manipulate objects of an unknown and arbitrary structure by performing very general operations on them, e.g., returning the memory space occupied by any arbitrary object to the free memory pool. Another possibility is to have the operating system define its I/O devices as packages types. Then, as devices are added or removed, the system can dynamically reconfigure itself without operator intervention. Standard Ada requires all types to remain static, and to be known at compile time. Through Intel's extensions to Ada, the iAPX 432 supports such dynamic applications using arbitrary types directly.

Address Generation. As noted earlier, address generation and translation mechanisms are generally more complex in object-based systems. In the iAPX 432 all of the object references for a given protection domain exist in the domain object's access segment, or indirectly, in the access segment portion of objects which are referenced from the domain. Operand references, then, consist of two major parts,

one for selecting the access descriptor for the object, the other for providing the offset into the object for the requested operand (see Figure 10). The access selection field which selects the access descriptor for the object can contain either a direct reference embedded in the instruction stream (see Figures 11), or an indirect reference. Indirect references are access selectors located on the top of the operand stack (see Figures 12), or access selectors located within a data object that is in turn selected using a direct access selector (a general indirect access--see Figures 13). Direct access selectors are limited to either four or eight bits in



*INDIRECT MAY BE FROM STACK, LOCATION WITHIN OBJECT SELECTED BY ACCESS SELECTION FIELD, OR LOCATION WITHIN ANOTHER OBJECT

Figure 10. Operand Reference.

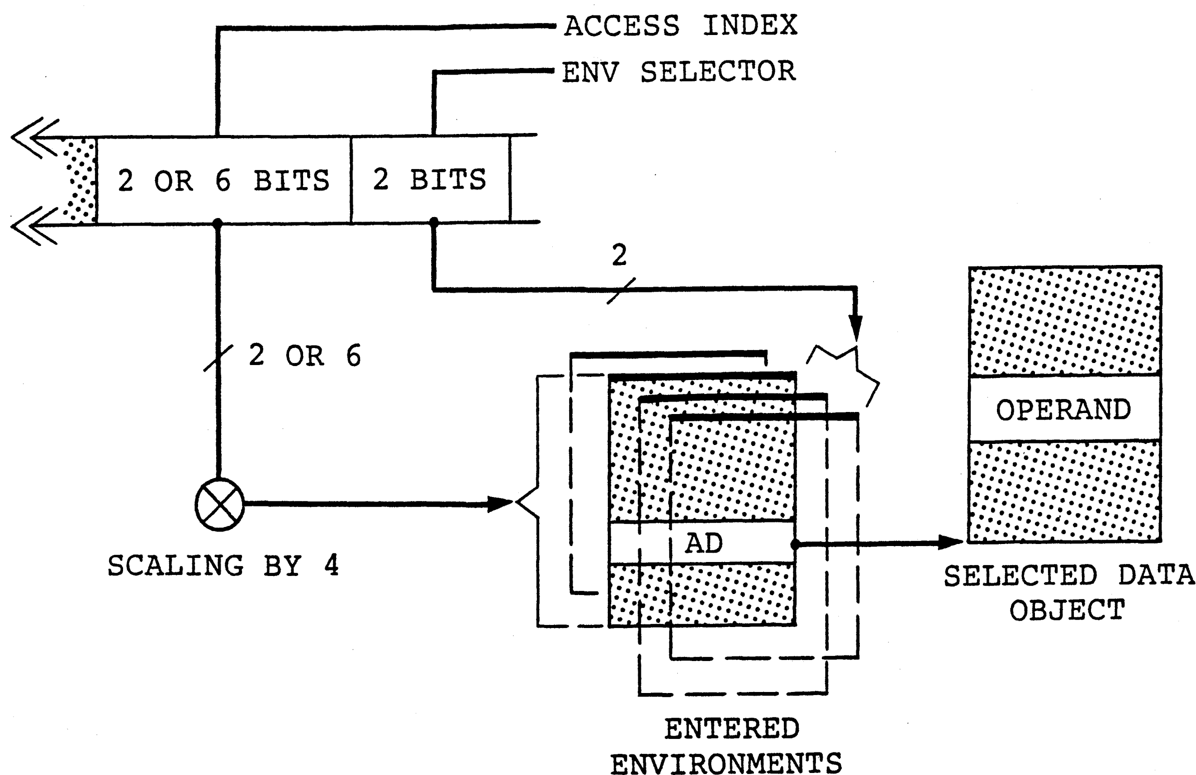


Figure 11. Direct Access Selection.

length. In contrast, indirectly referenced access selectors are all sixteen bits in length, allowing the selection of a much larger number of access descriptors albeit at a higher cost in terms of memory references and instruction length. For each of the access selection reference modes two bits of the selector are used to select one of four addressing environments, with the remaining bits (fourteen in the case of the indirect references) used to provide an index into the selected environment. Since each access segment entry is a four byte access descriptor the fourteen bit index, which is scaled, is able to address 2^{16} bytes--the maximum size for an object. The four environment entries which hold access descriptors for the access segments are held in registers on chip.

The offset reference field itself consists of two basic parts, a base part and an index part (Figure 10). Both the base and index parts can be referenced either

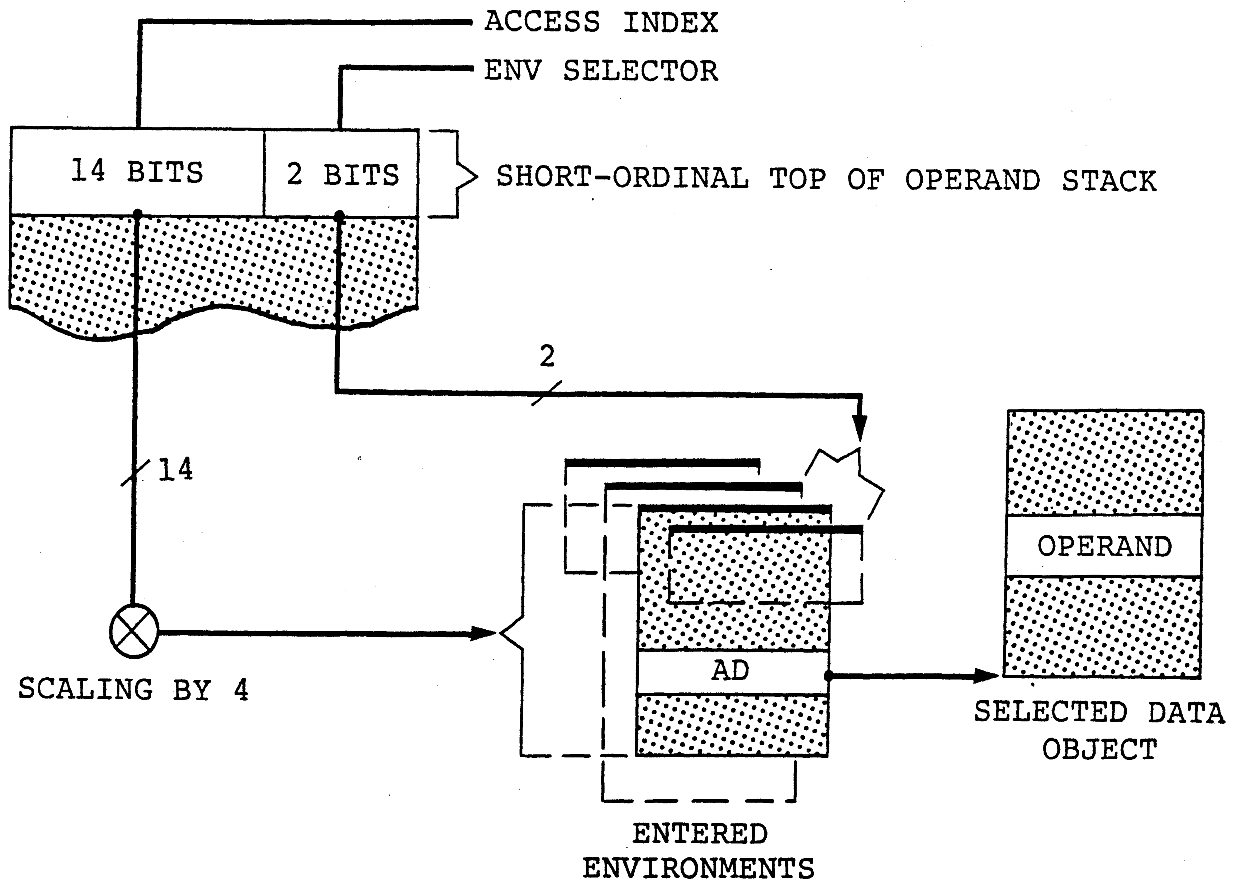


Figure 12. Stack Indirect Access Selection.

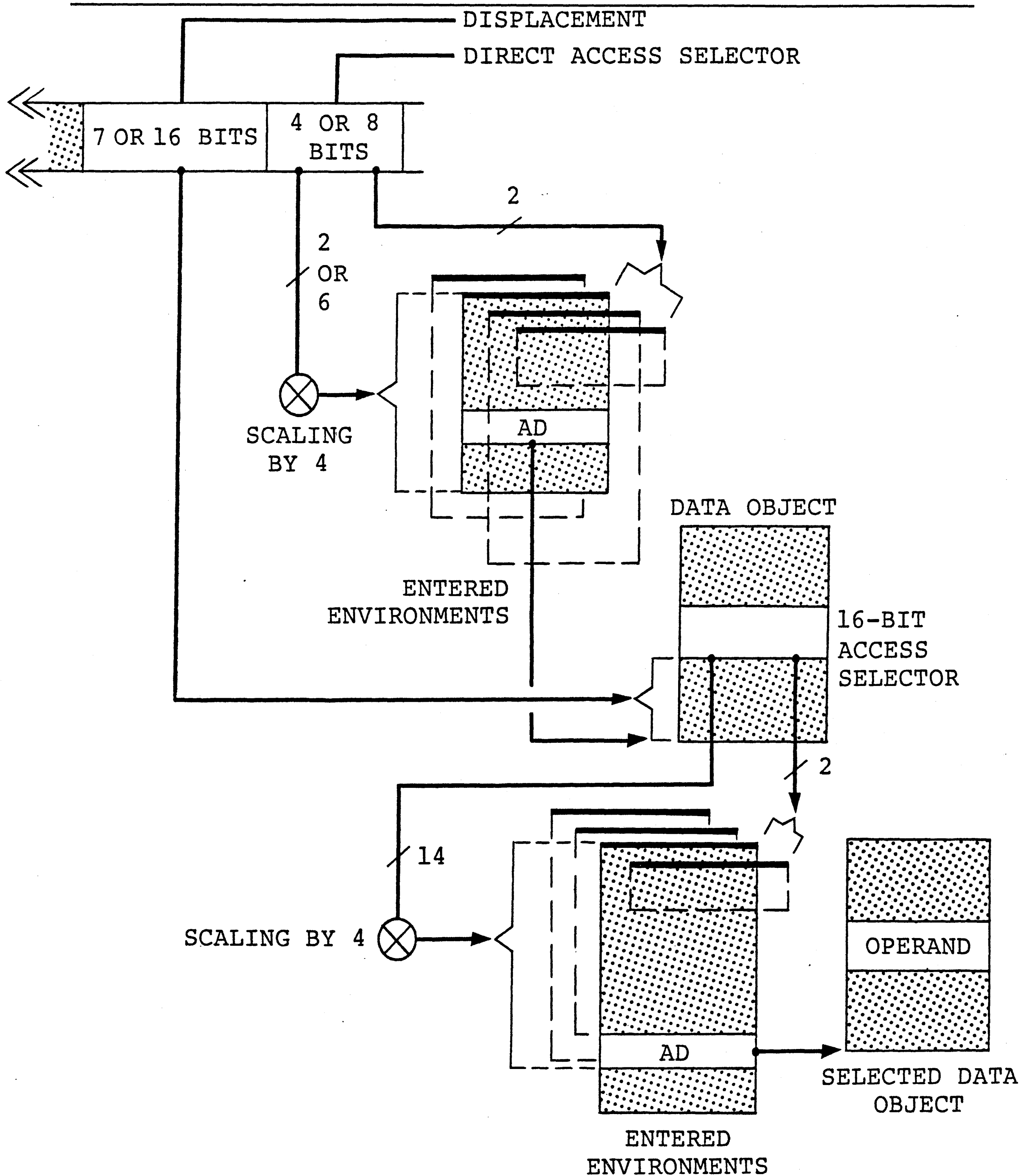


Figure 13. General Indirect Access Selection.

directly or indirectly. The direct offsets, which, again, are embedded in the instruction stream, consist of either a zero or sixteen bit field in the case of the base part, and either a seven or sixteen bit field in the case of the index part. Indirect offset

references can be in any one of three forms: stack indirect, general indirect or intrasegment indirect. Stack indirect references provide the sixteen bit base or index value on the top of the operand stack. General indirect references use a direct access selector (see Figure 11) to select an access descriptor for an object which holds the base or index value. They then use a directly specified displacement field of either seven or sixteen bits to locate the desired value within the selected object. The intrasegment indirect references provide a seven or sixteen bit displacement into the same object which was selected by the access selection component to extract either the base or index value. There are four combinations of direct and indirect base and index parts, and each is tailored to referencing a specific type of data structure. The table below summarizes this:

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="2">BASE</td></tr>
<tr><td>Direct</td><td>Indirect</td></tr>
<tr><td rowspan="2">INDEX</td><td>Direct</td><td>scalar</td><td>record item</td></tr>
<tr><td>Indirect</td><td>static array</td><td>dynamic array</td></tr>
</table>

In the foregoing discussion we saw how operands were addressed. The addressing mechanism made use of access descriptors whenever referencing an object. (These were capabilities for the object.) An access descriptor represents a logical address for an object. Although it was not shown in Figures 11, 12, and 13, logical to physical address translation for access descriptors requires a two level table similar in nature to the page and segment tables of the IBM System 360 (see Figure 14).

A system-wide table, the object table directory, exists at a known physical address. This table represents the first level of the mapping process and contains

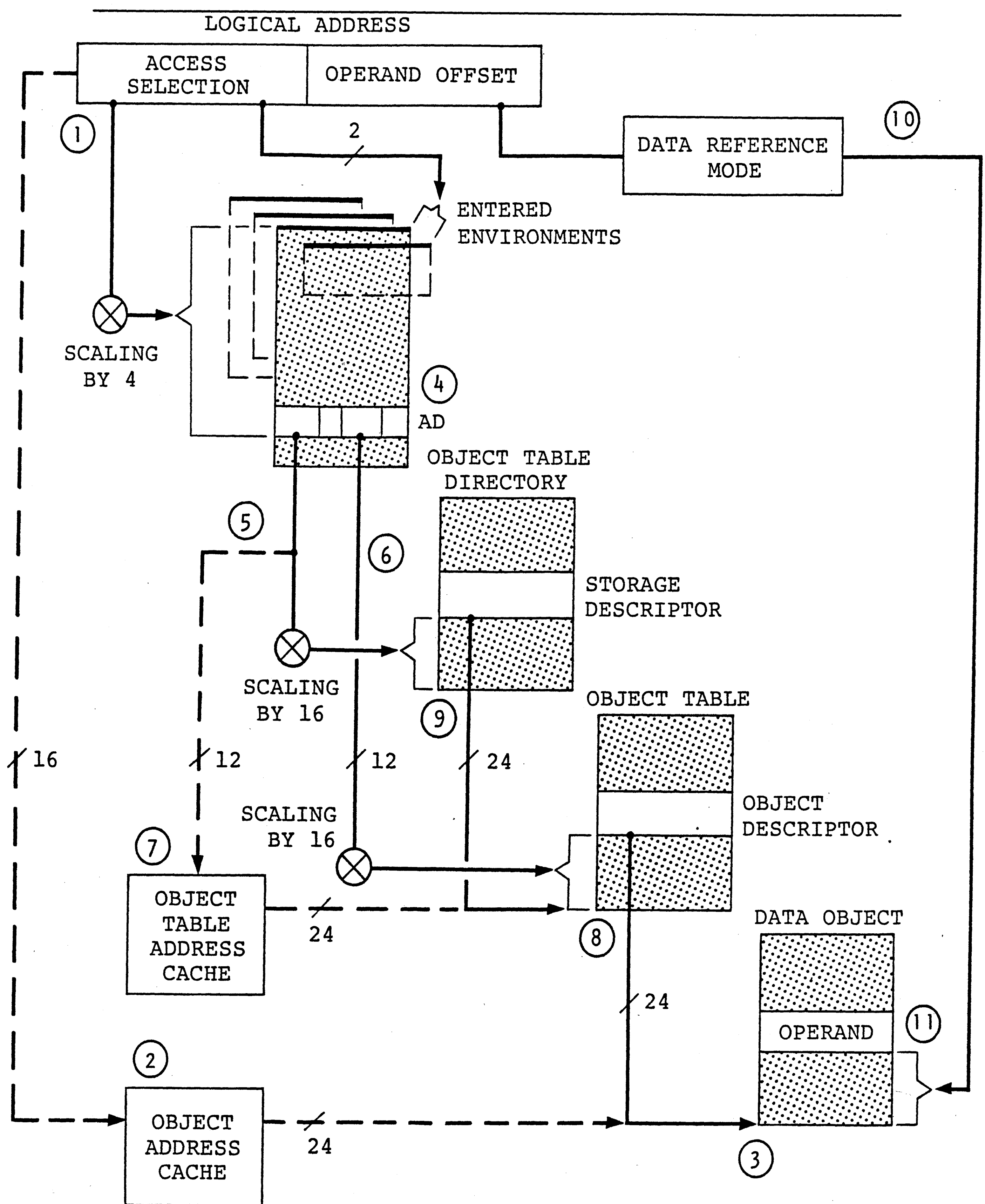


Figure 14. Physical Address Generation/Translation.

the base addresses of all of the object tables in the system (a maximum of 4096 are allowed). Object tables represent the second level of the mapping process and contain descriptors for all of the objects associated with their respective tables. There is roughly a one-to-one correspondence between processes and object tables. The descriptors found within the object tables contain a 24-bit physical base address, length, type, and other pertinent information for their respective objects.

As noted, the logical addresses that get translated are the access descriptors. Besides containing rights (i.e., read, write, copy, etc.) information, the access descriptors also contain two twelve bit index fields. The first field provides the index into the object table directory to select a descriptor for an object table; the second indexes into the specified object table to select the descriptor for the desired object. The above addressing scheme provides a total virtual address space of 2^{40} bytes, this comprises the 2^{12} object tables which can each contain 2^{12} descriptors for objects that can be up to 2^{16} bytes in length. However, at any one instant of time a process's logical address space is limited to 2^{32} bytes. This is because there are only four (2^2) addressing environment registers which hold access descriptors for access segments. These segments can contain up to 2^{14} access descriptors for objects, and the objects can have maximum lengths of 2^{16} bytes. The two level translation scheme facilitates relocation of objects even though there can be up to 2^{24} objects in the address space of the machine.

The translation procedure is traced in Figure 14 by following the circled numbers (see [5]):

- The access selector (1) is used to search the object cache (2).
  - **If a match occurs:** (Implying the data object address (3) has been previously cached.) The 24 bit physical address from the object cache entry is used to locate the data object in which the operand resides.
  - **If no match occurs:** The access selector is used in the normal way to locate an access descriptor in the current access environment (4). The specified access selection mode of the current data reference can entail a recursion of physical address generation for each instance of an access selector in the access selection mode.
    - The directory index (5) in this access descriptor is used to search the object table cache (7).
      - **Match occurs:** (Implying the indexed object table address (8) has been previously cached.) The 24 bit physical address from the object table cache entry is used to locate the indexed object table.
      - **No match occurs:** The object table is located normally through the object table directory (9) by using the directory index (5). When this is the case, the 24 bit physical address for the object table is loaded (with other information) into an appropriate entry in the object table cache.
    - The segment index (6) from the access descriptor (4) is used to index into the object table (8) to the object directory for the selected data object (3). This object directory is then used to provide a 24 bit physical base address for the data object. When this is the case, the 24 bit physical base address is loaded (with other information) into an appropriate entry in the object cache (2).
- The operand offset is calculated using the specified data reference mode (see above table). This calculation itself can entail a recursion of physical address generation for each instance of an access selector in the data reference mode.
- The calculated operand offset (10) is added to the physical base address of the data object (3) to obtain the final physical address of the first byte of the operand (11).

One can see from Figure 14 that several memory references may be required to retrieve a piece of data if its address is not available in the cache. The exact number, of course, depends on the particular combination of access selection and offset reference modes used, and on the effectiveness of the caching strategy.

IV. CONCLUSION

In this paper we have characterized object-based computer architectures in the context of the Ada programming language. This was done by illustrating key concepts with examples drawn from hardware and software systems. A case was made for object-based systems reducing system development costs and providing a very secure execution environment. These benefits require the use of elaborate addressing mechanisms which significantly increase address generation/translation times. Consequently, the performance of such systems becomes heavily dependent upon efficient address generation and effective address translation bypass schemes. Quantifying the trade-off between execution time on the one hand, and the granularity of protection and the dynamic support for data and program abstraction on the other hand is the first step in evaluating object-based computer architectures. Development of the techniques necessary for this quantification and the evaluation of any changes in the architecture that may result present important research issues.

V. REFERENCES

[1] R. A. Volz, T. N. Mudge and D. A. Gal, "Using Ada as a Robot System Programming Language," *Proceedings of the 13th International Symposium on Industrial Robots and Robots 7 Conference*, April 1983, pp. 12-42 thru 12-57.

[2] T. N. Mudge, R. A. Volz and D. E. Atkins, "Hardware/Software Transparency in Robotics Through Object Level Design," *Proceedings of the Society of Photo-optical Instrumentation Engineers Technical Symposium West*, SPIE Vol. 360, August 1982, pp. 216-223.

[3] M. Shaw, "The Impact of Abstraction Concerns on Modular Programming Languages," *Proceedings of the IEEE*, Vol. 68, No. 9, September 1980, pp. 1119-30.

[4] P. J. Denning "Fault Tolerant Operating Systems," *Computing Surveys*, Vol. 8, No. 4, December 1976, pp. 359-89.

[5] *iAPX 432 General Data Processor Architecture Reference Manual, Revision 3*, 171860-003, Intel Corporation, Santa Clara, California, 95051, 1982.

[6] *Reference Manual for the Ada Programming Language*, 171869-002, Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051, 1981.

[7] T. N. Mudge, G. D. Buzzard, D. J. Verhaeghe, J. Hill and D. C. Winsor, "Object-based Computer Architectures," *Proceedings of the 1983 Conference on Information Sciences and Systems*, The Johns Hopkins University, March 1983, pp. 733-741.

[8] E. I. Organick, M. P. Maloney, D. Klass and G. Lindstrom, "Transparent Interface Between Software and Hardware Versions of Ada Compilation Units," *University of Utah Report UTEC-83-030*, Department of Computer Science, June, 1983.

[9] *IBM System/38 Functional Concepts Manual*, GA21-9330-1, IBM Corporation, 1982.

[10] D. M. England, "Capability Concept Mechanism and Structure in System 250," *International Workshop on Protection in Operating Systems*, IRIA, Rocquencourt, August 1974, pp. 63-82.

[11] R. Fabry, "Capability-based Addressing," *CACM*, Vol. 17, No. 7, July 1974, pp. 403-12.

[12] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*: A Modular Multi-Microprocessor," *AFIPS Conference Proceedings*, Vol. 46, 1977 National Computer Conference, pp. 637-43.

[13] Wulf, W. A. and Bell, C. G., "C.mmp - A Multi-Mini-Processor," *AFIPS Conference Proceedings*, Vol. 41, part II, FJCC 1972, pp. 765-77.

[14] Needham, R. H. and Walker, R. D. H., "The CAP computer and its protection system," *ACM 6th Symposium on Operating System Principles*, 1977.

[15] A. K. Jones and E. F. Gehringer, (eds.), "The CM* Multiprocessor Project: A Research Review," *Carnegie-Mellon University Report CMU-CS-80-131*, Department of Computer Science, July 1980.

[16] J. G. P. Barnes, *Programming in Ada*, Addison-Wesley Publishers, London,1982.

[17] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[18] P. Wegner, "On the Unification of Data and Program Abstraction in Ada," *ACM Conference Record of the 10th Annual ACM Symposium on Principles of*

Programming Languages, January 1983, pp. 256-64.

[19] *Reference Manual for the Intel 432 Extensions to Ada*, 172283-001, Intel Corporation, 3065 Bowers Avenue, Santa Clara, California, 95051, 1981.

[20] T. A. Linden, "Operating System Structures to Support Security and Reliable Software," *Computing Surveys*, Vol. 8, No. 4, December 1976, pp. 409-45.

[21] *iAPX 432 Object Primer*, 171858-001, Rev. B, Intel Corporation, 3065 Bowers Avenue, Santa Clara, California, 95051, 1980.

UNIVERSITY OF MICHIGAN
3 9015 02651 8608