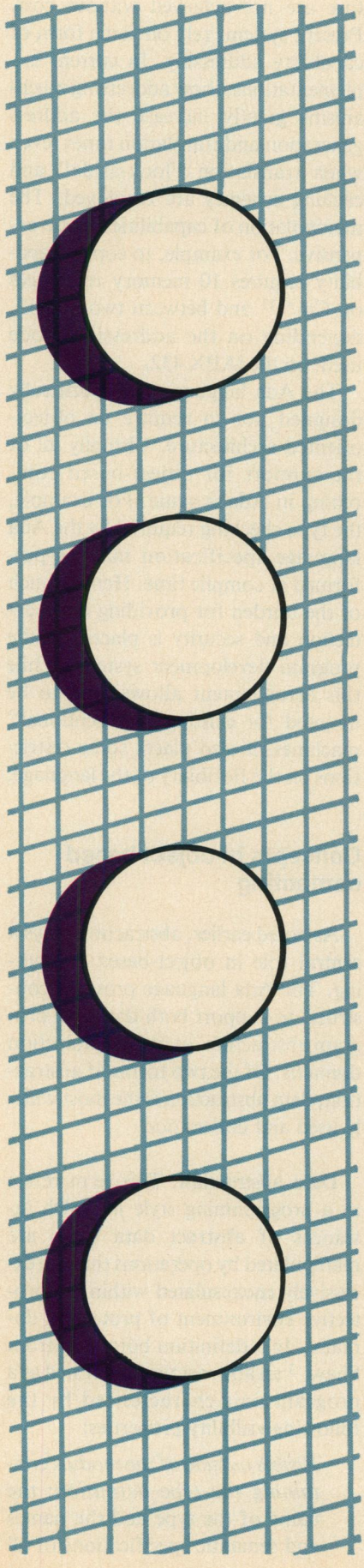


Object-Based Computing and the Ada Programming Language

G. D. Buzzard and T. N. Mudge
University of Michigan

Developments in several areas of computer science and engineering have coalesced during the past several years into a systems design methodology known as object-based computing. The primary benefit of this methodology is that it raises the level of abstraction available in the design process. Among the events that have encouraged the development of object-based computing are:

(1) advances in programming language design, such as program decomposition criteria and abstraction mechanisms;

(2) advances in operating systems design, particularly those which address data integrity and program security; and

(3) advances in computer architecture that allow direct hardware support for many operating system and language concepts.

Object-based computing is characterized by the extensive use of abstraction. Resources such as data, logical or physical devices, and, in some systems, program segments, may be represented by *abstract types*. Instances of abstract types form the objects in object-based computing. These objects are manipulated exclusively by operations that are encapsulated in a protective environment commonly referred to as a *type manager* or *protection domain*.

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

The first step in an object-based design philosophy is the identification of appropriate objects. The second step is the determination of operations on those objects. For example, consider a part-sorting system. The objects in such a system may include a camera, a robot, a frame (describing position), a set of parts, and bins to receive the parts. For robot objects, the set of available operations may include:

- INITIALIZE_ARM(robot_id)
- MOVE(robot_id, frame)
- OPEN(robot_id)
- CLOSE(robot_id)
- GET_FRAME(robot_id)

Each operation takes as a parameter *robot_id,* which uniquely identifies individual instances of the type robot, allowing a number of different robots to be used in the system. All instances of the same type share some common behavioral characteristics—in this case, the set of operations that is applicable to all robots.

Much of the growing interest in object-based computing is attributable to the Department of Defense's commitment to the Ada language project. Although Ada (DoD's proposed standard system implementation language for embedded systems) may not fit all definitions of an object-based language, it does incorporate key concepts. We will explain several concepts that are central to object-based computing and the extent to which they are supported by the Ada programming language. As part of the discussion, examples are given that have been

0018-9162/85/0300-0011$01.00 © 1985 IEEE

drawn from our use of Ada in programming a robot-based manufacturing cell.[1,2]

Object-based computing concepts apply to both hardware and software systems. In software systems, the term object-based describes software environments that incorporate the concepts of data abstraction,[3] program abstraction,[4] and protection domains.[5] Objects are represented as individually addressable entities that uniquely identify their own contents as well as the operations that may be performed on them. In the case of the robot object type, users have the ability to address specific robot objects individually and to manipulate their contents in a protected manner through operations, such as MOVE, that are explicitly defined for use on robot objects. Users need not be concerned with such details as object representation.

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. The second goal is to implement software systems that resist both accidental and malicious corruption attempts. Protection domains, which are described in the next section, are used to achieve the latter goal. As will be shown, support for both of these goals is provided by the Ada language.

Reductions of total life-cycle software costs in Ada are aided by the abstraction mechanisms provided in the language. The high level of abstraction that can be attained helps to increase programmer productivity by permitting the construction of generalized, reusable software units such as the robot controller described earlier. Also, once the interfaces of the abstractions are defined, program development for the implementation of each abstraction can proceed independently and in parallel, since the implementation details remain hidden (abstracted) from the view of the rest of the system. Abstraction aids software maintainability as well, by containing the effect of all changes, except those involving the interfaces, within the module housing the abstraction.

In hardware, the term object-based refers to the architectural support provided for the abstractions mentioned above. An essential goal in the development of such systems is to provide an efficient execution environment for the software system. Among the commercially available computer systems that provide support for object-based computing are the IBM System/38, [6] the Intel iAPX 432,[7] and the Plessy PP250.[8] Many of the ideas incorporated in the commercial systems were based on the results of several university projects (a good history of this work is given in Levy[9]). The most recent of these projects include Cm*[10] C.mmp,[11] and CAP.[12] A common thread running through all of these machines is the use of capability addressing techniques to implement secure protection domains, which can then be appropriately structured to provide data, program, and device abstractions.

The Ada language provides constructs to support both data and program abstraction, as well as protection domains.

Object-oriented machines are particularly well suited to applications that have stringent requirements for data security and program integrity in a dynamic environment. The high degree of abstraction provided by these machines also facilitates the interconnection of several processors into 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. As a further example, Cm* combines both tightly coupled multiprocessing and distributed networking concepts in one system.

As might be expected, the advantages of an object-oriented architecture are not achieved without cost. 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. The manipulation of capabilities is also expensive. For example, to copy a capability requires 10 memory references on Cm*,[13] and between two and 12, depending on the addressing mode used, on the iAPX 432.

The Ada language was specifically designed not to require an object-oriented architecture. Virtually all of the support for object-based computing in Ada is static. For example, the type checking required by the Ada language specification may be performed at compile time. Hence, much of the burden for providing error detection and security is placed on the program development system. While this arrangement allows Ada to be targeted for existing "conventional" machines, it also places some restrictions on the flexibility of the language.

Concepts in object-based computing

As noted earlier, abstraction plays a central role in object-based computing. The Ada language provides constructs to support both data and program abstraction, as well as protection domains. Of the two forms of abstraction, data abstraction is the most widely used and understood.

Data abstraction. This term refers to a programming style in which instances of abstract data types are manipulated by operations that are exclusively encapsulated within the protective environment of protection domains. In a definition borrowed from Shaw, [3] an abstract type is defined as a program unit characterized by the following visibility properties:

- *Visible outside of the module containing the type definition*: the name of the type and the names and semantic specifications of all

visible operations (procedures and functions) that are permitted to use the representation of the type. Some languages (e.g., Ada) also include formal specifications of the values that variables of this type may assume, and of the properties of these operations.

- *Not visible outside of the module containing 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.[14] The Ada package effectively places a wall around a group of declarations and permits access only to those declarations that 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 that may be necessary to enforce type consistency across compilation boundaries (Ada supports the notion of programs that comprise several separate compilation units). The body of the package contains the hidden implementation details.

Consider, for example, the Ada package ROBOT, whose specification is shown in Figure 1. This package can be used to abstract (hide) the implementation details of the type ROBOT_ARM, which has a physical device (a robot arm) associated with it. The operations available for manipulating this device, and even its existence, will be hidden by the abstraction (during program development, the physical robot arm may be represented by simulation code). Thus the applications programmer is freed from having to consider anything other than generating the code necessary to control the actions of an abstract device—the robot arm. Furthermore, the complete logical description of this abstract device is provided by the package specification, which is delimited by "**package** ROBOT **is**" and "**end** ROBOT;". (Ada reserved words are shown in bold, while user-defined and predefined package names, procedure names, function names, types, and variables are shown in upper case.)

The types defined in the package specification are as follows:

- ROBOT_ARM—the abstracted data type;
- ARM_MODEL—an enumeration type listing all of the robot arm models recognized by this system (two common robot models are shown, ASEA and PUMA); and
- FRAME—a 4×4 matrix that represents a homogeneous transformation. It represents 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.

The procedures are as follows:

- INITIALIZE_ARM—initializes an instance of the type ROBOT_ARM and moves the arm to a known starting position. The output of the procedure is of type ROBOT_ARM, and the input is of type ARM_MODEL.
- MOVE—takes as input a ROBOT_ARM and a FRAME and moves the arm and its hand to the position and orientation corresponding to the transformation given by FRAME. The updated arm is output, and its "hidden" state is changed to reflect its new position and orientation.
- OPEN and CLOSE control the arm's gripper;
- GET_FRAME is a function that provides controlled access to the hidden state, returning a value of type FRAME.

As noted earlier, the abstracted data type is ROBOT_ARM. It is the intent of the package ROBOT that this type be known only through the subprograms (procedures and functions) mentioned above and declared in the visible part of the package specification. The visible or public part of the package specification extends up to the reserved word "private." Hidden from public view within the package body are all of the other procedures, functions, and data structures that are necessary to effect movement of the physical robot arm and to update its representative data structure.

```
package ROBOT is
   type ROBOT_ARM is limited private;
   type ARM_MODEL is (ASEA, PUMA, . . .);
   type FRAME is array (1 . . 4, 1 . . 4) of FLOAT;

   procedure INITIALIZE_ARM (X: out
      ROBOT _ARM; KIND: in ARM_MODEL);
   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--not visible
   type ROBOT_ARM is record
         • -- Complete record contains
         • -- component of type FRAME
         • -- and ARM_MODEL
end ROBOT;
```

Figure 1. Specification for the package ROBOT.

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 subprograms 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. Moreover, since assignments to limited private types are also not allowed outside of their defining domain, the possibility of having inconsistent representations for the same robot arm (e.g., the logical data structure not reflecting the correct location of the physical robot) is eliminated.

In order to use the ROBOT package, the user program must first create instances of the necessary types in its declarative part, as follows:

```
ASEA:    ROBOT.ROBOT_ARM;
ARM:     ROBOT.ARM_MODEL
           := RBT;
TRANS:   ROBOT.FRAME;
```

Notice the use of explicit qualification ("ROBOT.") to name the defining domain for the types ROBOT_ARM, ARM_MODEL, and FRAME. This terminology informs the compiler that, for instance, the variable RBT is to be of type ROBOT_ARM, which is defined in the package ROBOT.

In the user program, subprograms from the ROBOT package to initialize and move a robot may be invoked as follows:

```
ROBOT.INITIALIZE_ARM(RBT, ARM);
    • -- compute a value for TRANS,
    • -- the homogeneous transform
    • -- describing
    • -- the movement required.
ROBOT.MOVE(RBT, TRANS);
```

The first parameter for both of the procedure calls, RBT, provides the logical representation of the robot that is to be manipulated. The second parameter of INITIALIZE_ARM specifies that RBT be the logical representation of an ASEA robot. The only way that any operations on the data structure RBT may be performed is by passing it as a parameter to a subprogram that appears in the visible part of the package ROBOT. Even though the user program has full access to the type FRAME, which is a component (not shown) of the limited private type ROBOT_ARM, it still may not directly manipulate the FRAME component of ROBOT_ARM. That is, external packages are prevented from directly manipulating components of hidden record types (private or limited private), regardless of the visibility of the components' types. The data structures, excluding those that are explicitly declared private or limited private, along with the procedures and functions that appear in the public part of the package specification, are directly available for use by packages external to ROBOT. Figure 2 provides a logical conceptualization of how the package ROBOT is used to move an instance of an ASEA robot. The same "black box" package can also be used to move any other valid instance of ROBOT_ARM.

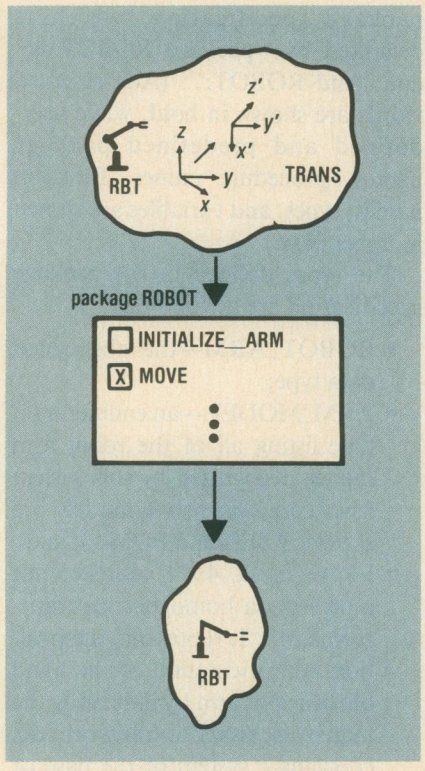


Figure 2. Logical view of data abstraction.

Through the use of packages and private types, the Ada language provides easy-to-use constructs to support data abstraction. As we have seen, data abstraction is a powerful tool for both program development and error confinement. However, it is not the only form of abstraction provided by the Ada language. Program abstraction provides a different and, in some senses, a more powerful conceptual view.

Program abstraction. Programs and subprograms provide another common level of abstraction. Program abstraction enables operations on implicit objects. In addition to hiding the representation of and access to an object, the object's existence is also hidden. The result is a more complete form of hiding and, usually, a more concise interface than data abstraction.

Program abstraction in Ada is realized through generic package or subprogram instantiation. Ada allows the declaration of generic program units that serve as templates for packages or subprograms from which actual pakcages or subprograms can be obtained. Generic program units may have actual parameters that provide instantiation-specific details of the template. These parameters can be data objects, types, or subprograms. Thus, the parameter associations allow generic units to do the following: directly manipulate data objects provided by the user; create and manage data structures or logical devices corresponding to an arbitrary type supplied by the user; and use abstract, instantiation-dependent subprograms made available by the user.

Through their types, the formal parameters specify the expectations that the generic unit makes about the actual parameters that will be supplied by the user of the generic unit at instantiation time. For example, a generic package that manages stacks might have a formal data object parameter of type POSITIVE that indicates the maximum size of the stack. This formal parameter will be matched by any actual parameter whose value at instantiation time is an integer greater than or equal to one. The same generic package may also have a formal type parameter of type private that allows the user to specify the type of elements that are to be handled by the instantiation of the stack package. This formal parameter will be matched by any actual parameter for which, at least, the (in)equality tests and assignment are defined; these are the only operations allowed on data objects whose types are private.

Generic formal subprogram parameters, when used, allow the user to provide—or select from a library— subprograms for use within the generic unit. In the case of a generic terminal handler, for instance, subprogram parameters for SEND and RECEIVE

procedures would free the programmer from having to accommodate the differing communication protocols that exist for various terminal types. Matching rules for generic formal subprogram parameters are similar to those for nongeneric subprogram parameters. In effect, the generic formal parameters provide a written "contract" between the writer of the generic template and the writer of any program that is designed to use this template. The terms of this contract are rigidly enforced by the type-checking facilities present in the compiler.

Generic units allow a higher level of abstraction than data abstraction, because the abstracted data type may be completely hidden within an instance of the generic package body. The hidden data structure is accessed through internal package variables that are nonlocal to the subprograms within the package. Manipulation of the data structure occurs as a controlled side effect—strictly contained within the package body—of the requested operation. Thus generic units support a programming style in which the specified (visible) operations either directly or indirectly transform a hidden internal state that 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. The external packages are no longer required to supply the type ROBOT_ARM for each operation that they invoke on the arm. Instead, the packages specify the element of the enumeration type ARM_MODEL that corresponds to the arm that they wish to use when they instantiate the generic package ROBOT (the generic package extends from the keyword **generic** to **end** ROBOT;). The generic template is filled in based on the value of the generic formal parameter ARM at the time of instantiation. Hence, the statements:

```
package RBT__ASEA is new ROBOT(ASEA);
package RBT__PUMA is new ROBOT(PUMA);
```

would create instances of the package ROBOT specifically for an ASEA robot and a PUMA robot, respectively. A new instance of the package would be created for each robot in the target system. Recall that in data abstraction, only one instance of the package existed. The logical view of a system consisting of one ASEA robot and one PUMA robot is illustrated in Figure 4.

Implementations for both program and data abstraction require that certain procedure bodies contain conditionally executed code segments to account for the differences between the numbers of links and the type of joints used in the different robots, and to allow individual robots to communicate with their respective device drivers. An alternative solution available in program abstraction is to add subprogram parameters to the formal part of the generic package. The actual subprograms, which could be made available to the applications programmer in the form of a library, would provide the necessary abstractions for manipulating their corresponding devices. The user would then select the appropriate set of actual parameters for each specific generic instantiation.

```
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.

The biggest difference between data and program abstraction lies in the program unit that possesses the instance of the abstracted data type. In data abstraction, the type is possessed by a program unit that is external to the unit that manages the type; hence, the unit that manages the type is commonly referred to as an external type manager. In program abstraction, the type is maintained within the unit that manages it; hence, the managing unit is commonly referred to as an internal type manager. External type managers

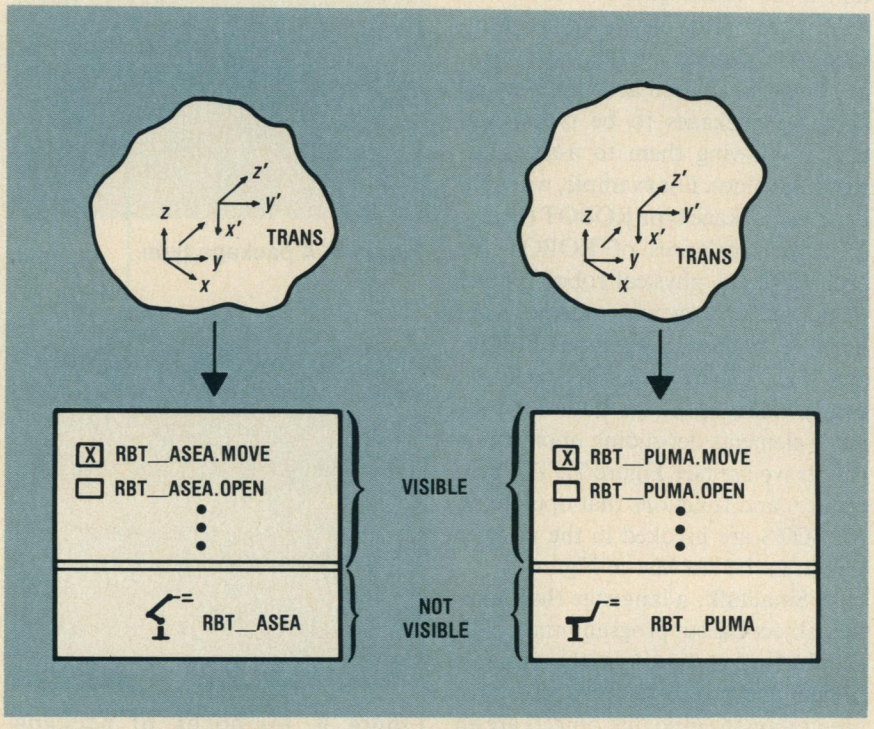


Figure 4. Logical view of program abstraction.

support a functional programming style in which the type to be manipulated is passed as a parameter to a subprogram that performs its operation without side effects.

The chief advantage of this programming style is that it can be modeled more readily using well-known mathematical techniques, thus opening up greater possiblilities for correctness proof methods. In Ada, it is also the case that data objects created at run-time by the dynamic instantiation of types can be handled by external type managers, whereas the instantiation of internal type managers and the creation of their enclosed data objects is static—that is, it must occur at compile time. It can be argued, however, that internal type managers provide a much more realistic model of the real-world objects that the computer program is meant to manipulate, and that the protection against errors both at run-time and during program development is much greater. Nongeneric packages may also function as internal type managers. However, in doing so, generality is lost, since the abstraction would be available for only one instance.

Intel has provided a powerful extention to the Ada language for the iAPX 432 processor that allows the programming of dynamic internal type managers. This has been accomplished by allowing packages to be types, and hence, allowing them to assume assignable values. For example, we could define a package type ROBOT (Figure 5), create instances of ROBOT for each different physical robot (Figure 6), declare an array whose elements are instances of the package type ROBOT, and then, during program execution, assign values (package bodies) to the array elements depending upon which robots we use (see Figure 7). The procedures and functions that operate on ROBOTS 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.[15] It has been proposed that these concepts be merged together in Ada as well.[4,16] In fact, Intel has already taken an initial step in this direction with their "package type" extension to the Ada language mentioned above.[17] Extensions such as package types and those described in Wegner[4] and Jessop[16] combine the dynamic flexibility of data abstraction with the conceptual and protection benefits of program abstraction, resulting in a powerful universal abstraction mechanism. The availability of such a mechanism relieves the software designer of the restrictive choice between adopting a program-oriented or data-oriented software design methodology—a choice that must occur very early in the design process.

Protection domains. Protection domains, and the inherent security that they provide, constitute another key concept. The basis for secure, error-tolerant execution environments lies in the principle of system closure.[5] This principle 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.[18] Briefly stated, a protection domain is an environment or context that defines the set of access rights and operations that are currently available to a specific user for objects contained within the domain. The concepts embodied in protection domains are similar to those underlying Ada packages.

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. By implication, any interactions with 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 include the binding of processes to processors, or the assignment of memory to currently executing contexts.

These controls ensure that when the resource units are released or preempted, all of the information contained within the unit is returned to a null state. Information is thus prevented from "leaking" out of a protection domain, even if it is left in an area that eventually becomes accessible to other users. Error confinement also aids the program debugging process, since bugs will be located in modules where errors are detected. Program maintenance benefits as well, since the protection domain defines the maximal set of modules that can be affected by modifying 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 state. Reconfiguration facilities attempt to restore the system to an operable state by removing the failed component—hardware

```
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 RBT_ASEA is constant ROBOT;
package body RBT_ASEA is
  •
  •
  •
end RBT_ASEA;

package RBT_PUMA is constant ROBOT;
package body RBT_PUMA is
  •
  •
  •
end RBT_PUMA;
```

Figure 6. Instances of package types.

or software. If the reconfiguration attempt is successful, the system is restarted.

Perhaps the most elegant mechanism for implementing protection domains is capability addressing. While much can be done at compile time in languages such as Ada to enforce the concept of protection domains, there are many cases where the dynamic enforcement of access rights provided by capability addressing is useful. As an example, consider a system in which it is desirable to grant different rights (perhaps the ability to invoke a different set of operations) to the various users of a given object based on information presented at run-time. Capability addressing permits users to dynamically determine whether they have indeed been granted the right to perform a requested operation. Determinations such as this are not possible at compile time. Compile-time protection enforcement also lacks the ability to support the detection of and recovery from access failures in the run-time system.

```
declare
  type ROBOT_ARRAY is array (1..LAST)
    of ROBOT;
  ROBOTS: ROBOT_ARRAY;
  NEW_POSITION: FRAME;
begin
  case ARM is
      when ASEA  =>
        ROBOTS(I) := RBT_ASEA;
      when PUMA  =>
        ROBOTS(I) := RBT_PUMA;
          •
          •
          •
  end case;

  ROBOTS(I).MOVE(NEW_POSITION);
      •
      •
      •
```

Figure 7. Assignment of package type.

A capability can be thought of as the name of an object. An object cannot be accessed—nor its existence determined—unless its name is known. 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 domain in which the capability is defined is the restriction of these rights. Capabilities are created along with their respective objects. The initial control of the capability, and hence of the object, belongs solely to the defining domain. Consider the case of a user—**package** USER in Figure 8—that selects the package ROBOT of Figure 1 in a system employing capability addressing. The defining domain for a variable of type ROBOT_ARM is ROBOT. In the case of pass-by-reference parameter association, the rights of the capability given to the instantiating context are restricted to "copy" because ROBOT_ARM is a limited private type. Within the defining domain, the capabilities may be amplified as needed. 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 a parameter of mode **in out**, requiring both read and write rights. In case of GET_FRAME, ROBOT_ARM is an **in** parameter, and thus requires only read rights.

The use of capabilities is not dependent on the particular method of parameter association used. Rather, in the case of paramaters passed by reference, the capability is amplified at the

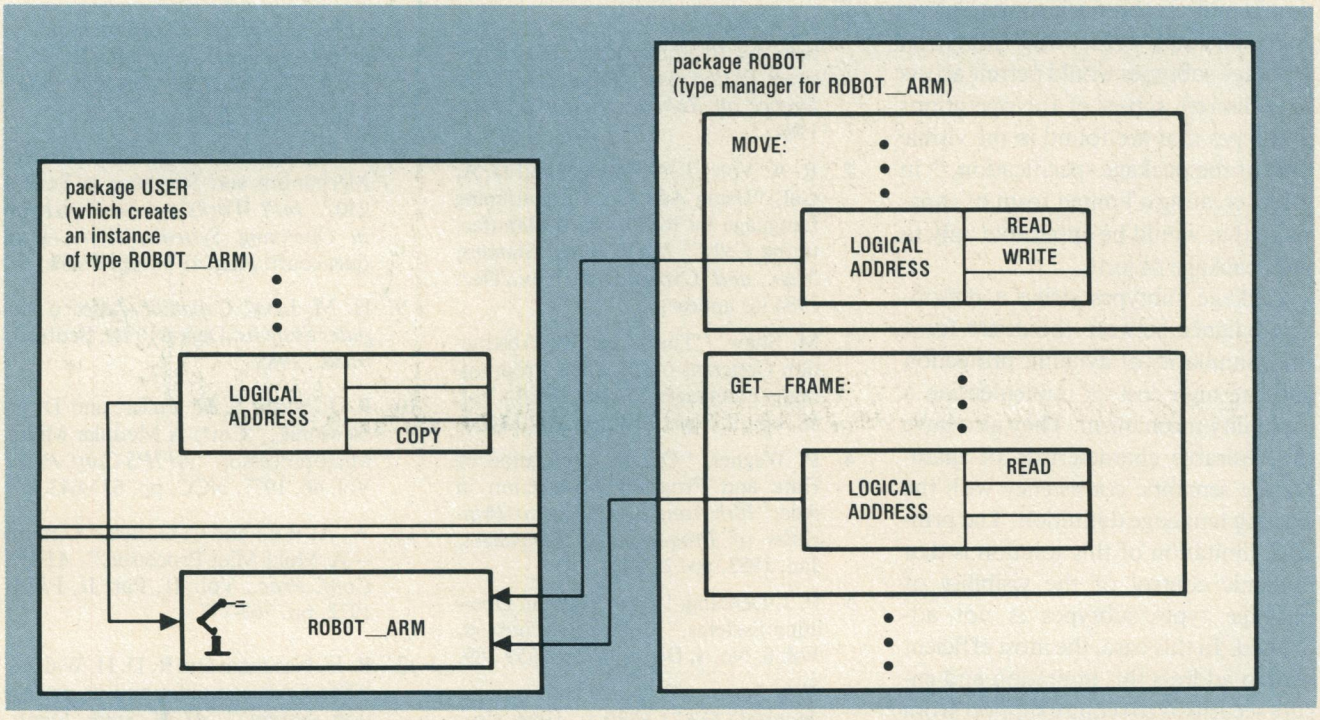


Figure 8. Capabilities and protection domains.

subprogram interface to allow the appropriate access rights. This is permitted because amplification occurs within the defining domain for the parameter type. In the case of parameters passed by value, the new instance of the parameter type is created with the appropriate access rights, and the parameter values are then copied—again, this is permitted since it also occurs within the defining domain.

Though the Ada language specifically avoids any mention of capability addressing, constructs that support many of the concepts embodied in protection domains are provided. They include packages and private types, and their associated compile-time type checking and exception handlers. The latter permit users to detect and categorize errors, and to reconfigure and restart the system. Protection domain concepts such as the control of information "leakage" lie below what is normally considered to be the language level. The major shortcoming of Ada with respect to support for protection domains lies in the fact that all users of objects external to the defining domain are treated equally. That is, Ada provides an "all or none" type of protection mechanism. A possible extension to Ada that would address this problem is to allow both package types and subtypes. Package subtypes would permit access to a discrete subset of the operations and types that are found in the visible part of the package specification,[6] in effect creating a limited form of capability that would be applicable only to Ada package objects.

Package subtypes strike a reasonable balance between the desire for a fine granularity of dynamic protection and the high cost of implementing a capability mechanism. They also have the desirable characteristic of maintaining semantic consistency with the existing language definition. The principal limitation of this solution is that dynamic control of the visibility of package types subtypes is not addressed. In this case, the most efficient way to address this limitation and enhance security is to require support from the underlying hardware.

Ada provides good support for both data and program abstraction. However, in an effort to allow Ada to be used efficiently on existing machines with conventional architectures, the language has been designed to require little dynamic checking. This effect is most noticeable in the limited support provided for protection domains and the lack of a unified abstraction mechanism that would include features such as package types. However, these features are usually not required in the application area for which Ada was primarily targeted—embedded systems. Current embedded systems typically have many static attributes: configurations are not modified at run-time; program implementations are not changed; and devices are not taken on- and off-line. Hence, within its intended application area, the Ada language is suitable for object-based computing on conventional architectures.

References

1. R. A. Volz and T. N. Mudge, "Robots are (Nothing More Than) Abstract Data Types," *Proc. SME Conference on Robotics Research,* Aug. 1984.
2. R. A. Volz, T. N. Mudge, and D. A. Gal, "Using Ada as a Programming Language for Robot-based Manufacturing Cells," *IEEE Trans. Systems, Man, and Cybernetics,* Nov./Dec. 1984 (to appear).
3. M. Shaw, "The Impact of Abstraction Concerns on Modular Programming Languages," *Proc. IEEE,* Vol. 68, No. 9, Sept. 1980, pp. 1119-1130.
4. P. Wegner, "On the Unification of Data and Program Abstraction in Ada," *10th Annual ACM Symp. Principles of Programming Languages,* Jan. 1983, pp. 256-264.
5. P. J. Denning, "Fault Tolerant Operating Systems," *Computing Surveys,* Vol. 8, No. 4, Dec. 1976, pp. 359-389.
6. *IBM System/38 Functional Concepts Manual,* GA21-9330-1, IBM Corporation, Rochester, Minn., 1982.
7. *iAPX 432 General Data Processor Architecture Ref. Manual,* Rev. 3, 171860-003, Intel Corporation, Santa Clara, Calif., 1983.
8. D. M. England, "Capability Concept Mechanism and Structure in System 250," *Int'l Workshop on Protection in Operating Systems,* IRIA, Rocquencourt, Aug. 1974, pp. 63-82.
9. H. M. Levy, *Capability-based Computer Systems,* Digital Press, Bedford, Mass., 1983.
10. R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*: A Modular Multi-Microprocessor," *AFIPS Conf. Proc.* Vol. 46, 1977, NCC, pp. 637-643.
11. W. A. Wulf and C. G. Bell, "C.mmp—A Multi-Mini-Processor," *AFIPS Conf. Proc.,* Vol. 41, Part II, FJCC 1972, pp. 765-777.
12. R. H. Needham and R. D. H. Walker, "The CAP computer and its protection system," *ACM Sixth Symp. Operating System Principles,* 1977.

13. 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.

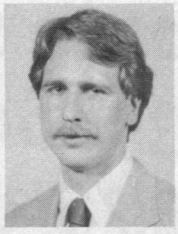
14. J. G. P. Barnes, *Programming in Ada,* 2nd ed., Addison-Wesley, London, 1984.

15. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, Reading, Mass., 1983.

16. W. H. Jessop, "Ada Packages and Distributed Systems," *Sigplan Notices,* Vol. 17, No. 2, Feb. 1982, pp. 28-36.

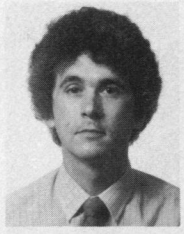
17. *Reference Manual for the Intel 432 Extensions to Ada,* 172283-001, Intel Corporation, Santa Clara, Calif., 1981.

18. T. A. Linden, "Operating System Structures to Support Security and Reliable Software," *Comp. Surveys,* Vol. 8, No. 4, Dec. 1976, pp. 409-445.



Gregory D. Buzzard received the BS and MS degrees in electrical engineering from the University of Michigan, Ann Arbor, in 1981 and 1982, respectively. He is currently pursuing the PhD in electrical engineering in the area of architectural support for intertask communication in distributed systems.

Buzzard is a Kodak Fellow and a member of the University's Robotics Research Laboratory. He is also a member of Eta Kappa Nu, Tau Beta Pi, ACM and the IEEE Computer Society.



Trevor Mudge received the BSc degree in cybernetics from the University of Reading, England, in 1969, and the MS and PhD degrees in computer science from the University of Illinois, Urbana, in 1973 and 1977, respectively. He has been with the Department of Electrical Engineering and Computer Science at the University of Michigan since 1977 and currently holds the rank of associate professor. His research interests include computer architecture, programming languages, VLSI design, and computer vision.

Questions concerning this article can be addressed to the authors at the Center for Robotics and Integrated Manufacturing, University of Michigan, 2514 E. Engineering Bldg., Ann Arbor, MI 48109.