THE ADA DEVELOPMENT METHOD (ADM):
An object-oriented software development method for the entire development cycle

17 June 1989

presented at the
Summer 1989 SIGAda Conference in Ottawa
by

Donald G. Firesmith, President
Advanced Software Technology Specialists (ASTS)
3418 Broadway
Fort Wayne, IN 46807, USA
(219) 456-9260

Abstract: This paper describes the Ada Development Method, a recursive Ada- and object-oriented software development method that 1) covers all of the software activities of DOD-STD-2167A [1] (i.e., Software Requirements Analysis, Preliminary Design, etc.) and 2) supports the design of dynamic behavior and process abstraction. It describes the method-specific graphics and discusses the method's management implications.

Keywords: Ada, Call_Rendezvous Chart, Library Diagram, Object-Oriented Development, Object Diagram, OOD, recursion

INTRODUCTION

Object-Oriented Development/Design (OOD) is not a software development method, but rather a large and growing class of related recursive, Ada-oriented software development methods [2,3,5,6,7,8,9,10,11,12] based on object abstraction. Unfortunately, this *embarrassement de riches* has presented the Ada manager and developer with a difficult choice, exacerbated by the fact that most OOD methods suffer from the same set of problems: they tend to

ignore the critical impact of recursion, they are typically restricted to the Preliminary Design activity and do not support Software Requirements Analysis or Testing, they supply inadequate graphics for exception handling and dynamic behavior, and they tend to ignore tasking and the design of dynamic behavior. The **Ada Development Method (ADM)** is an attempt to provide a single, unified Ada-oriented software development method to answer these important limitations. This paper will describe the major concepts underlying ADM, the method-specific graphics used by the method, the individual steps of the method, and important management implications of the method.

MAJOR CONCEPTS

The concepts of abstract object, class, recursion, assembly, and subassembly form the foundation of ADM and must be known in order to understand the individual steps of the ADM development process. The concepts of abstract object and class come originally from the object-oriented programming, systems, languages, and applications (OOPSLA) community, require modification for Ada (e.g., Ada has only limited inheritance and does not support dynamic binding), and are not yet completely standardized in the Ada community. The concept of recursion (with regard to software development methods) is critical because it leads to new development cycles different from the classic waterfall life-cycle. The concepts of assembly and subassembly are not unique to ADM, but are useful to all recursive methods that incrementally develop software.

An **ABSTRACT OBJECT** is:

- An abstraction of a single physical or conceptual entity (e.g., hardware device, person, purpose) from the real-world, requirements specification, or problem domain. This is the abstract object of software requirements analysis and early static architectual design.
- The associated software blackbox (e.g., abstract state machine package) that controls, emulates, implements, models, simulates, stimulates, or tracks it.

An abstract object also:

- Is an instance of some (possibly annonomys) class.
- Has a sharply defined boundry and interface.
- Therefore exhibits both an outside (i.e., user) and inside (i.e.,

developer) view.

- Is uniquely identifiable (typically with a noun or noun phrase that is meaningful from the user (rather than the developer) viewpoint.
- Encapsulates the following resources:
  - Data (including state information).
  - Operations (that may impact the encapsulated data).
  - Exceptions (usually associated with the operations).
- Is influenced by its history as well as outside influences, and thus has state in the mathematical sense.
- Is completely characterized (from the user viewpoint) by its exported operations and exceptions. The structure of the encapsulated data is considered an implementation detail, is therefore hidden from the user of the abstract object, and is protected by information hiding.
- Almost always impacts other abstract objects only by invoking their visible exported operations (i.e., data is protected by being hidden and is neither directly accessable nor shared).
- Is abstract (i.e., exports operations, exceptions, and possibly Ada variables and constants at a high level of abstraction while hiding implementation details.
- Should be reusable and therefore complete (i.e., export all operations, both primitive and compound, and exceptions needed by the intended user. This supports maintainability, simplifies configuration management, and significantly reduces total source and object code size.
- Is typically implemented in Ada as an Abstract State Machine package or task.

Note that:

- Abstract objects (e.g., packages and tasks) therefore contain Ada objects (i.e., data variables or constants) and are thus not the same.
- Object abstraction includes data, functional, and process abstraction.

A **CLASS** is:

- A set of instances of such abstract objects, all having the same characteristics (i.e., attributes, operations, exceptions) and conforming to the same rules.
- The associated software blackbox (e.g., Abstract Data Type package, generic package).

RECURSION (with regard to software development methods) is the repetition of the same steps to generate new product at the next lower level of abstraction. This is to be compared with iteration which is the repetition of the same steps on the same product, typically to correct errors.

An **ASSEMBLY** is the total set of all Ada programming units developed during all recursive repetitions of the software development method when applied to a specific set of coherent software requirements (i.e., requirements that specify a single well-defined problem). An assembly in Ada is typically a single Ada program. This is the software whose static structure is documented on an Assembly Library Diagram (ALD). See Figure 1. Depending upon when the recursive method is initiated, an assembly can be either a DOD-STD-2167A system, subsystem/segment, Computer Software Configuration Item (CSCI), or Computer Software Component (CSC). See Figure 2.

A **SUBASSEMBLY** is the set of those Ada programming units developed during only a single non-recursive pass through the recursive software development method. This is the amount of software whose static structure is documented on either a Subassembly Library Diagram (see Figures 3 and 4) or traditional Booch Diagram [3] (See Figure 5) or PAMELA 2 (or SCOOP 3) Library Graph [7,8] (see Figure 6). A Higher-Level Subassembly contains at least one other subassembly, and is roughly equivalent to what has been called a Rational Subsystem. A Lowest-Level Subassembly contains only library units. A subassembly should be mapped to a CSC under DOD-STD-2167A, (see Figure 2) and can be thought of as Higher-Level CSCs (HLCSCs) and Lowest-Level CSCs (LLCSCs), although these abbreviations are different than those of TLCSC and LLCSC found in the superseded DOD-STD-2167.

ADM GRAPHICS

Almost all Ada software development methods are highly graphical in nature, and ADM is no exception to the observation that graphics are far superior to Ada PDL or code when used to document multiple Ada units and their relationships. ADM incorporates the following 10 graphics (in order of production) for software requirements analysis and design:

- External Objects Diagram (EOD).
- Assembly Library Diagram (ALD).
- Subassembly Object Semantic Net (SOSN).
- Subassembly Object_Operation Diagram (SOOD).
- Subassembly Object_Data/Control Flow Diagram (SOD/CFD).
- Object State Transition Diagram (OSTD) or Table (OSTT).
- Library Unit Data/Control Flow Diagram (LUD/CFD).
- Subassembly Library Diagram (SLD).
- Call_Rendezvous Chart (CRC).
- Timing Diagram.

The **EXTERNAL OBJECTS DIAGRAM (EOD)** is the context diagram of ADM and is used to document the interfaces between an assembly and its external world. It identifies the assembly, the external entities (e.g., hardware objects, other assemblies) with which it interfaces, and the data and control flows between the assembly and these external entities. See Figure 7.

The **ASSEMBLY LIBRARY DIAGRAM (ALD)** is used to document the static structure of the assembly in terms of its subassemblies and the recursion relationship between them. See Figure 1. It is also an important management tool for scheduling, staffing (one small software development team is typically assigned to each subassemly), and work breakdown as well as a configuration management tool that contains the same information (and more) as the DOD-STD-2167A software organization diagram. See Figure 2.

The **SUBASSEMBLY OBJECT SEMANTIC NET (SOSN)** is used to identify the subassembly abstract objects and classes and the relationships between them. See Figure 8. It provides much the same information as the information model of Object-Oriented Systems Analysis [9].

SUBASSEMBLY OBJECT_OPERATION DIAGRAM (SOOD) is used to

SUBASSEMBLY OBJECT_DATA/CONTROL FLOW DIAGRAM (SOD/CFD) is used to

OBJECT STATE TRANSITION DIAGRAM (OSTD) OR TABLE (OSTT) is used to

LIBRARY UNIT OBJECT_DATA/CONTROL FLOW DIAGRAM (LUD/CFD) is used to

SUBASSEMBLY LIBRARY DIAGRAM (SLD) is used to

CALL_RENDEZVOUS CHART (CRC) is used to

TIMING DIAGRAM is used to

THE METHOD

At the highest level, ADM consists of the following main steps:

1) IDENTIFY ASSEMBLIES AND BUILDS.
2) SCHEDULE AND STAFF EACH ASSEMBLIES DEVELOPMENT.
3) FOR EACH BUILD:
3.1) Recursively develop the relevant subassemblies of the relevant assemblies.
3.2) Release the build software and documentation to the project Software Configuration Management (SCM) organization.
3.3) Independently test the build software.

Step 3.1 consists of the following steps are repeated recursively to develop each subassembly:

1) SUBASSEMBLY REQUIREMENTS ANALYSIS AND DESIGN PHASE

1.1) Initiate Subassembly Development.
1.1.1) Schedule the milestones of subassembly development.
1.1.2) Staff the subassembly software development teams.
1.2) Analyze the Subassembly Requirements.
1.2.1) Store the subassembly requirements in the subassembly Software Development File (SDF).
1.2.2) State the subassembly objective. If initial subassembly, the objective of the entire assembly. If a child subassembly, the objective is the objective of the stubbed operation whose recursion led to the creation of the subassembly. The primary purpose of this step is to help the developers use object abstraction to identify the relevant abstract objects and classes at this level of abstracton.
1.2.3) Review the subassembly requirements. The developers prepare to identify the relevant abstract objects and classes

1.2.4) If initial subassembly, develop the External Objects Diagram (EOD). This is the context diagram of ADM. See Figure 7.

1.2.5) Use object abstraction to identify all subassembly abstract objects and classes. This step requires practice and experience.

1.2.6) Develop the Subassembly Object Semantic Net (SOSN). This documents the subassembly's objects, classes, and their relationships. The SOSN also helps the developer determine the exported operations of the subassembly objects and classes. See Figure 8.

1.2.7) Develop the Subassembly Object_Operation Diagram (SOOD). This is the main external view of the objects, classes, and their most important relationships (i.e., the operations they require of one another. See Figure 9.

1.2.8) Develop the Subassembly Object Data/Control Flow Diagram (SOD/CFD). This diagram shows the data and control flows between the objects and classes. See Figure 10.

1.2.9) Analyze and organize the subassembly requirements. If initial subassembly, these consist of all assembly requirements. If a child subassembly, these are those requirements allocated to the stubbed operation requiring recursion that led to the creation of the current child subassembly. The requirements are sorted first by abstract object and class, and then by operation.

1.2.10) Develop the State Transition Diagram or Table for each abstract object or class with complicated states or transitions. This step will help the developer determing the object 's encapsulated data and operations. See Figure 11.

1.2.11) Develop the Library Unit Data/Control Flow Diagram for each abstract object, class, and other library unit. Note that all library units will not be abstract objects and classes. See Figure 12.

1.3) Develop the Subassembly Logical Design. This step consists of developing, compiling, and debugging Ada PDL/code for the main subprogram (if initial subassembly) or for the stubbed operation (if a child subassembly). This PDL will not contain all exception handlers yet because the developers will not yet know what exceptions will be reased by units at lower levels of abstraction.

1.4) Iterate back to upgrade the subassembly requirements graphics as necessary based on the Subassembly Logical Design.

1.5) Analyze all subassembly abstract objects and classes. For each object and class:

1.5.1) Analyze and determine the main (e.g., exported) Ada objects and types in the context of their encapsulating abstract object or class.

1.5.2) Analyze and determine the main (e.g., exported) operations in the context of their encapsulating abstract object or class and the type of Ada object on which they will operate. Add additional operations as necessary to promote reuse, simplify use, and decrease user code size.

1.5.3) Develop the logical design for each operation using compilable Ada PDL/code.

1.5.4) Identify the associated developer-defined exceptions that may be raised by the abstract objects and classes.

1.6) Analyze any higher-level subassemblies (i.e., subassemblies that will contain subassemblies) in terms of their interfaces.

1.7) Develop the draft Subassembly Library Diagram (SLD) to document the static architecture of the subassembly in terms of the subassembly library units and their dependencies. See Figures 3 and 4.

1.8) Augment the Subassembly Logical Design with the necessary exception handlers using Ada PDL/code.

1.9) Identify other auxilliary library units (e.g., object relationship packages, types and constant packages) as necessary. Remember that even on OOD projects, Ada programs typically contain a small number of library units that do not implement abstract objects and classes.

1.10) Update the Subassembly Library Diagram (SLD) with these auxilliary library units as necessary.

1.11) Determine opportunities for reuse now that the subassembly's main library units have been identified, but not yet coded.

1.12) Code and compile the specifications of all known subassembly (package and generic package) library units.

1.13) Design and document the subassembly dynamic behavior.

1.13.1) Identify all hidden subprograms and tasks in the subassembly, additional library unit level data and control flows, subprogram calls, task rendezvous, rendezvous directions, rendezvous controls (e.g., guards, delays, selects, etc.), intermediate or third-party tasks, process abstraction packages and generic packages (e.g., buffers, pumps, relays, transporters), etc.

1.13.2) Develop one or more Subassmbly Call_Rendezvous Charts (CRCs) to document this dynamic design. See Figure 13.

1.13.3) Determine additional opportunities for reuse now that

additional library units have been identified.

1.13.4) Code and compile the specifications of the additional library units (e.g., process abstraction packages and generic packages).

1.13.5) Code and compile the initial bodies of the subassembly library units including:

1.13.5.1) Subprogram specifications using stubs and the separate clause.

1.13.5.2) Task specifications.

1.13.5.3) Task bodies using stubs and the separate clause.

1.13.6) Update the Subassembly Library Diagram (SLD) with the additional library units and dependencies.

1.13.7) Develop Subassembly Timing Diagrams (STDs) as necessary to document the expected timing of subprogram calls and task rendezvous. See Figure 14.

1.13.8) Use tools (e.g., based on Petri Net analysis) as necessary to analyze complex tasking design.

1.14) Make additional design decisions based on software engineering considerations.

1.14.1) Determine the operations requiring recursion. These are the complex subprograms and task entries that must remain temporarily stubbed because they will depend upon as yet unidentified resources in a child subassembly (i.e., at the next lower-level of abstraction).

1.14.2) Allocate unmet requirements to the resulting child subassemblies as necessary.

1.15) **Perform the peer-level Subassembly Requirements and Design Inspection** of all intermediate products produced during steps1 through 1.14.2 and stored in the Subassembly Software Development File (SDF).

1.16) If necessary, initiate recursion on all operations that must be stubbed (see step 1.14.1).

2) SUBASSEMBLY CODE AND TEST PHASE

2.1) Code and compile all subprogram and task bodies not requiring recursion. Use separate subunits corresponding to the already existing stubs (see step 1.13.5).

2.2) Plan subassembly testing.

2.3) Design and code subassembly test software.

2.4) Peform initial (library) unit testing of the subassembly

software.

2.5) Integrate the subassembly software.

2.6) **Perform the peer-level Subassembly Code and Test Inspection** of all intermediate products produced during steps 2 through 2.5 and stored in the Subassembly Software Development File (SDF).

2.7) Place the subassembly software into the Assembly Ada Library. This software is now under developer configuration control.

2.8) Integrate the subassembly into the growing assembly.

2.9) Update the Assembly Library Diagram (ALD) with the current subassembly. See Figure 1.

2.10) Use tools to update the deliverable documentation.

MANAGEMENT IMPLICATIONS

BIBLIOGRAPHY

[1] *Defense System Software Development, DOD-STD-2167A*, Department of Defense, 29 February1988.

[2] Berard, Ed, *Object-Oriented Design Handbook for Ada Software*, EVB Software Engineering, Inc., 1985.

[3] Booch, Grady, *Software Engineering with Ada, Second Edition*, Benjamin/Cummings Publishing Co., 1986.

[4] Buhr, Dr. R. J. A., *Systems Design with Ada*, Prentice-Hall, 1984.

[5] Bulman, David M., *Model-Based Object-Oriented Development*, Pragmatics, Inc., 1989.

[6] Cherry, George W., *PAMELA 2: An Ada-Based Object-Oriented Design Method*, Thought Tools, Inc., 1988.

[7] Cherry, George W., *Software Construction with Object-Oriented Pictures* , Thought Tools, Inc., 1988.

[8] Seidewitze, Ed, and Stark, Mike, *Generalized Object-Oriented Software Development*, NASA Goddard SW Engineering Laboratory, 1986.

[9] Shlaer, Sally and Mellor, Stephen J., "An Object-Oriented Approach to Domain Analysis", Project Technology, Inc., 1989.

[10] Shlaer, Sally and Mellor, Stephen J., *Object-Oriented Systems Analysis*, Yourdon Press, 1988.

[11] Shumate, Ken, *Understanding Ada with Abstract Data Types, Second Edition*, John Wiley & Sons, 1989.

[12] Vidale, Dr. R. F., "Extending Object-Oriented Ada Design Methodology", GTE Government Systems Corporation, 1986.

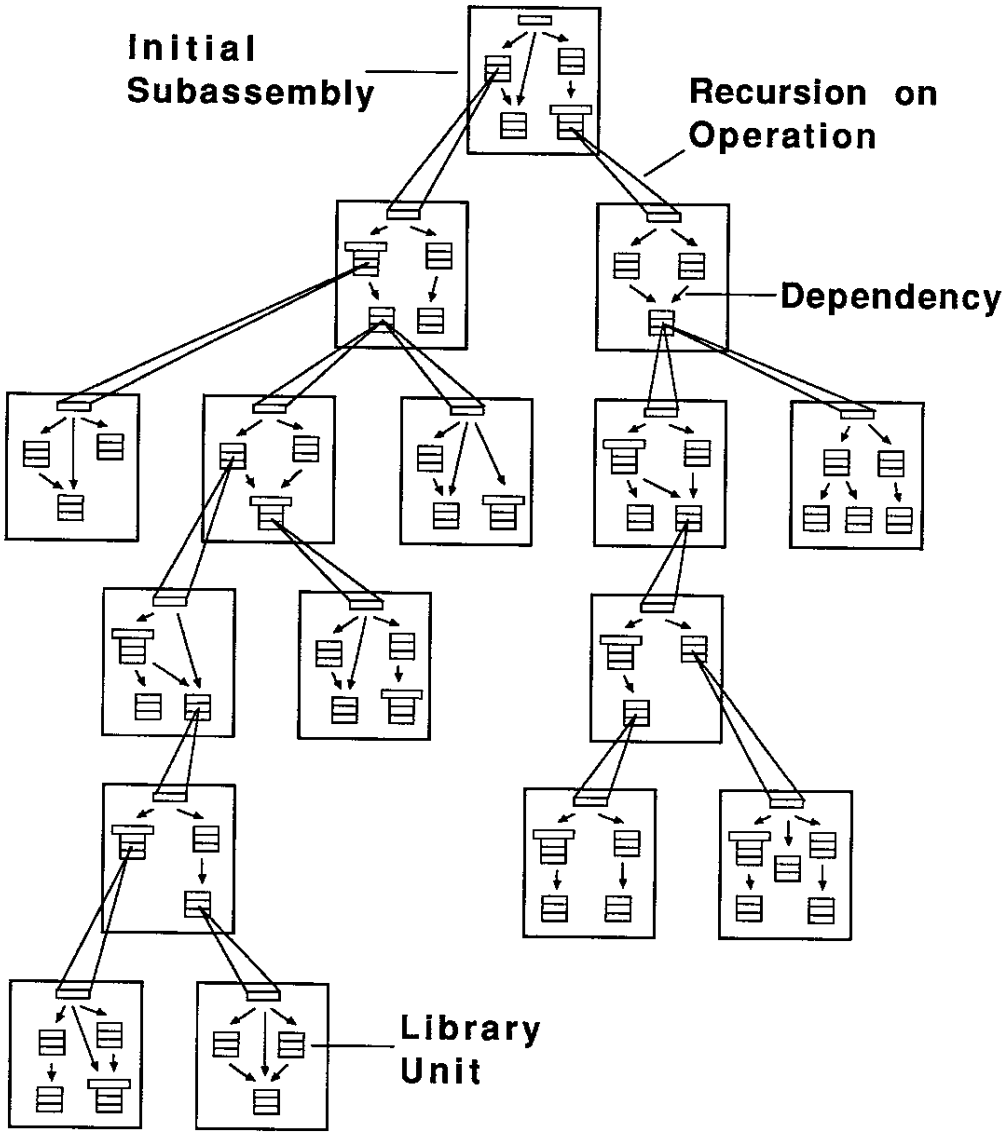

FIGURE 1: ASSEMBLY LIBRARY DIAGRAM (ALD)

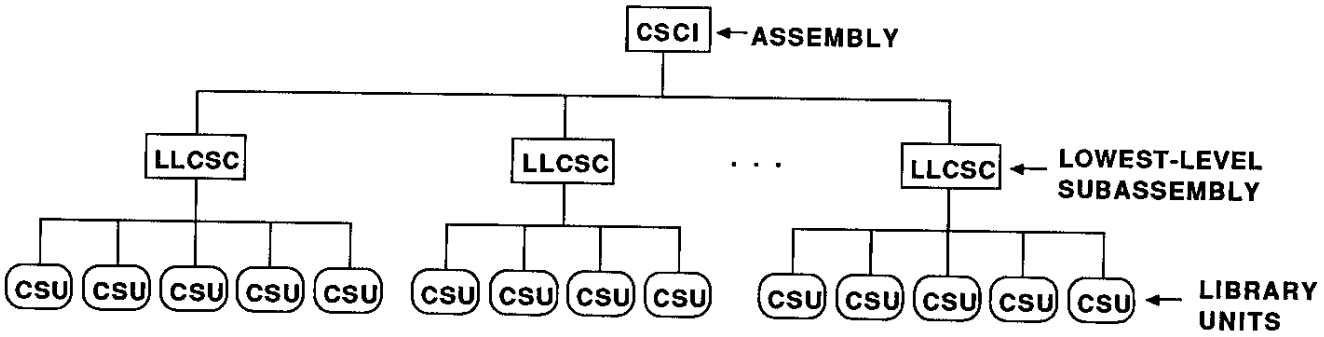

FIGURE 2: DOD_STD_2167A SOFTWARE ORGANIZATION

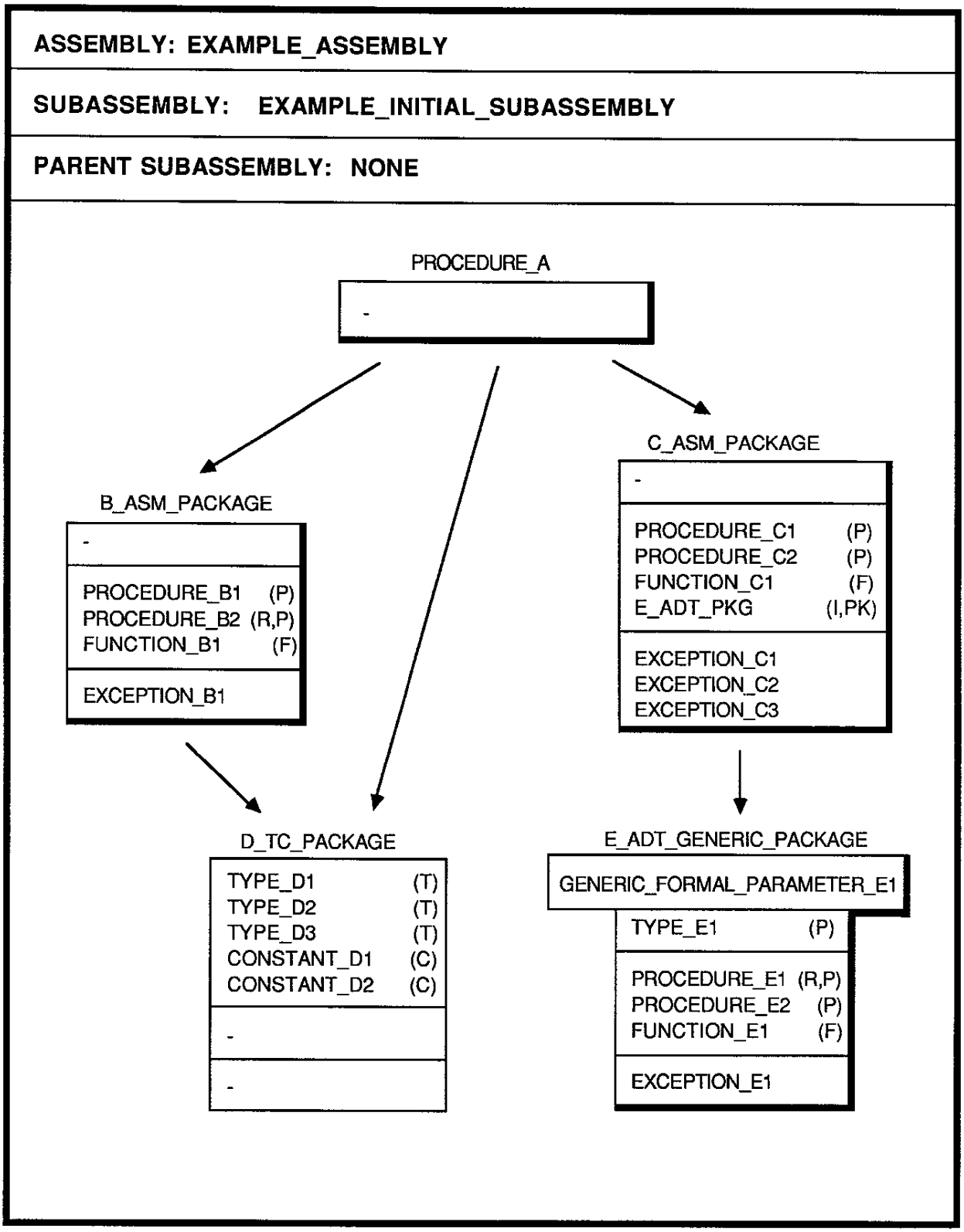

FIGURE 3: SUBASSEMBLY LIBRARY DIAGRAM (SLD)

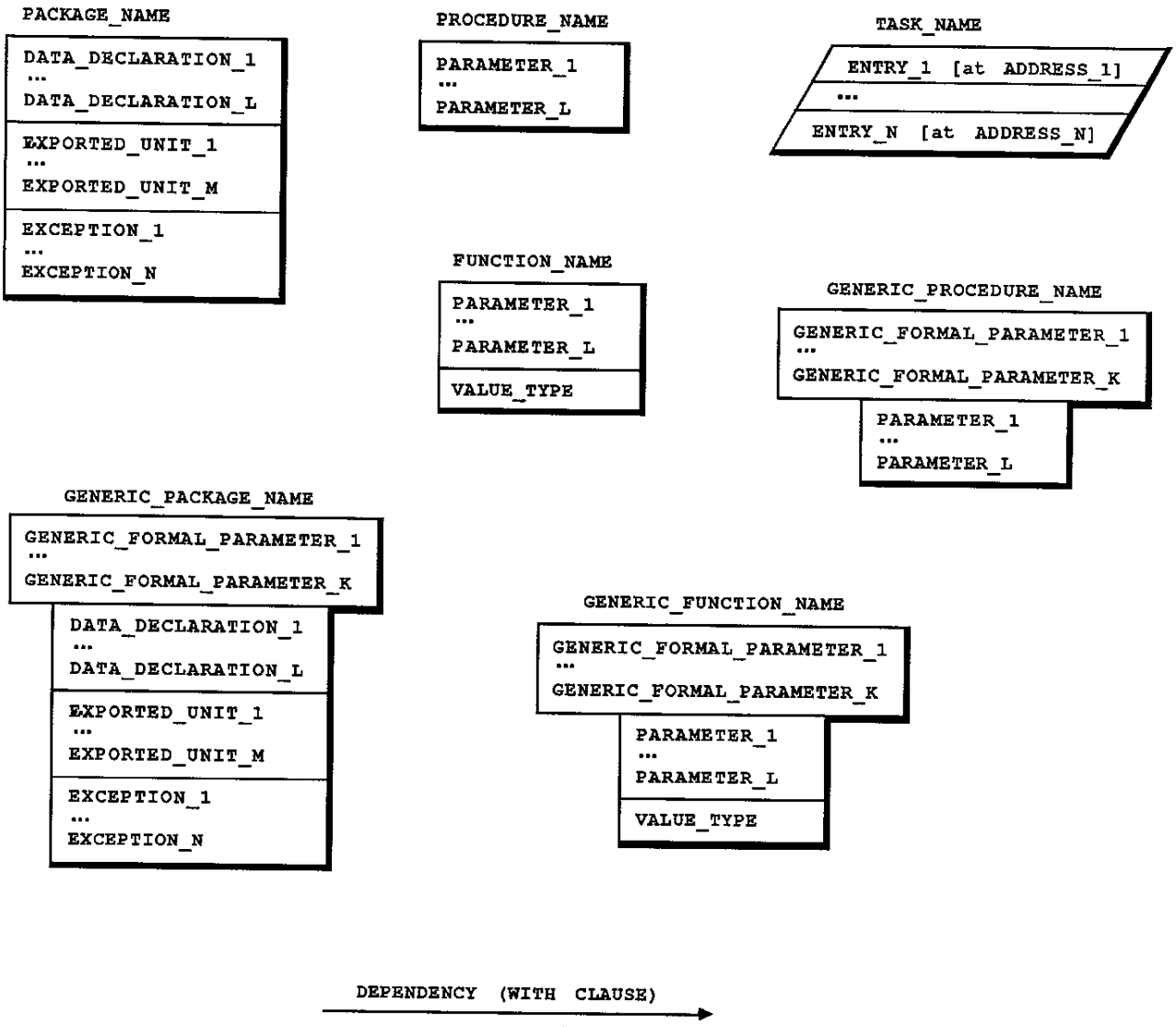

FIGURE 4: ADA LIBRARY GRAPH ICONS

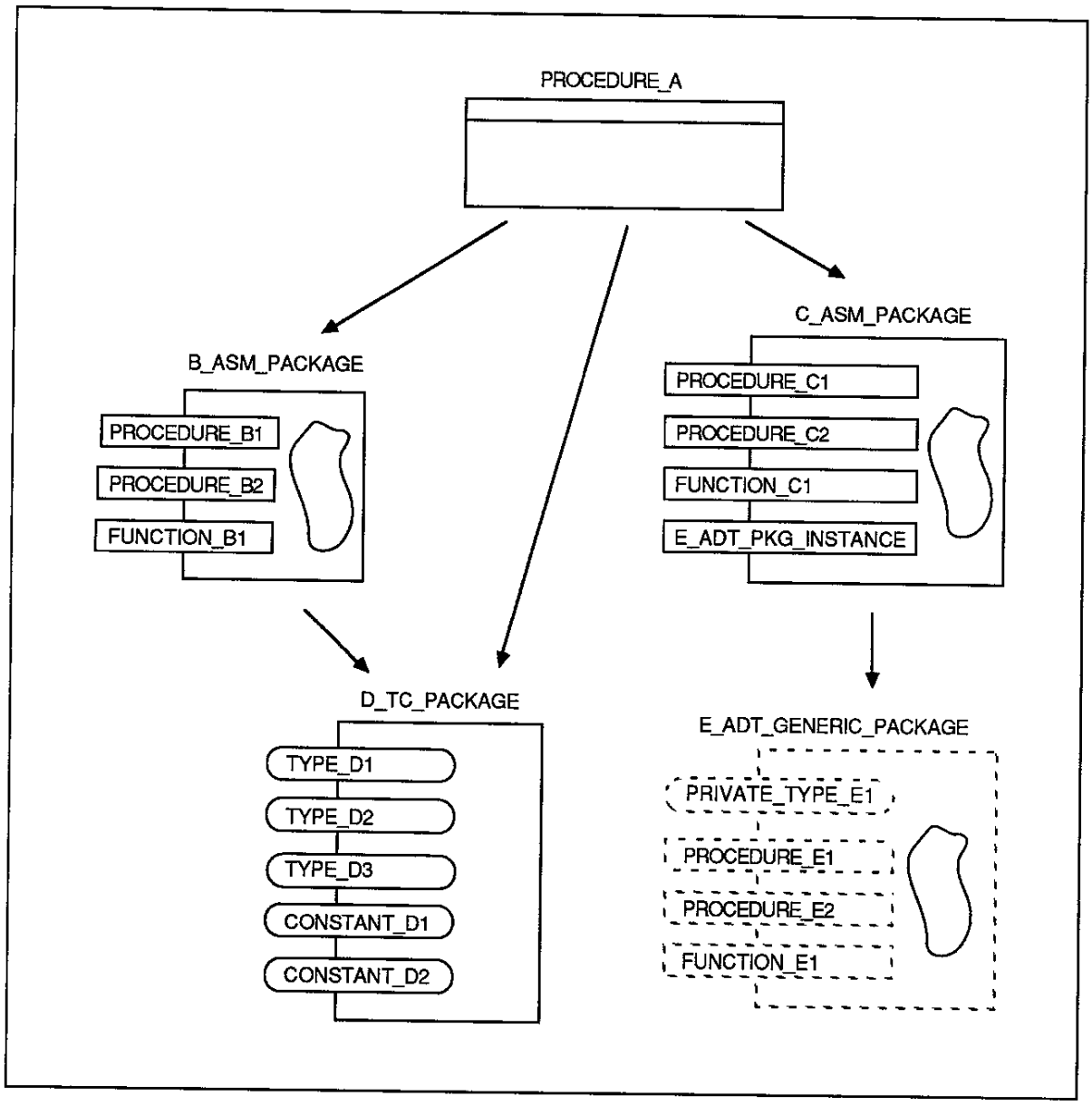

FIGURE 5: BOOCH DIAGRAM

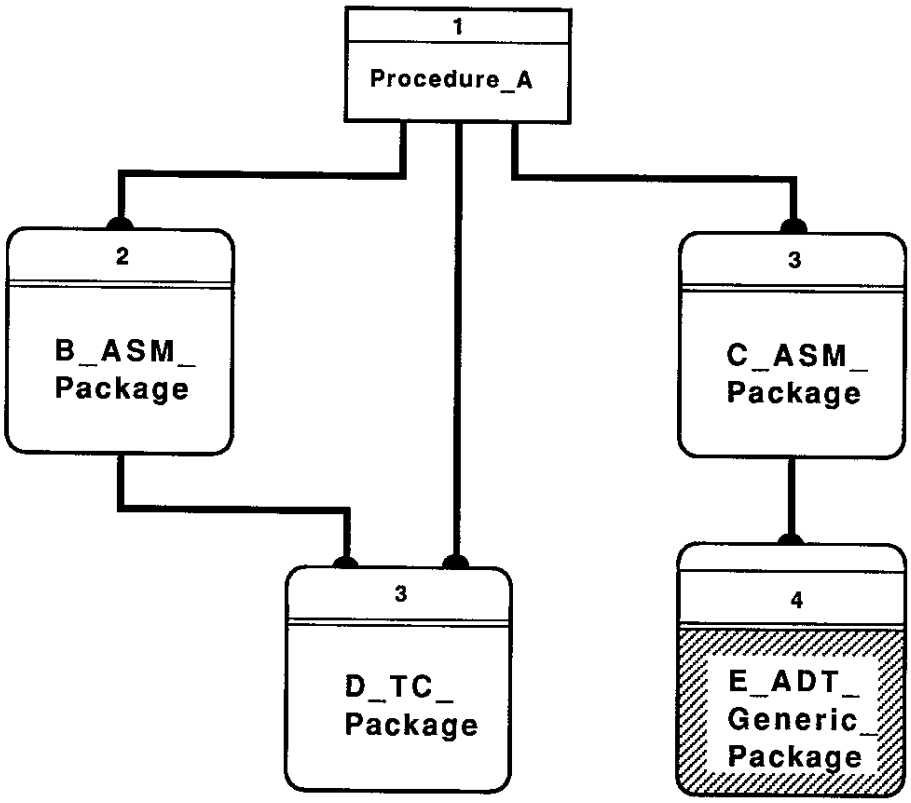

FIGURE 6: SCOOP3 LIBRARY GRAPH

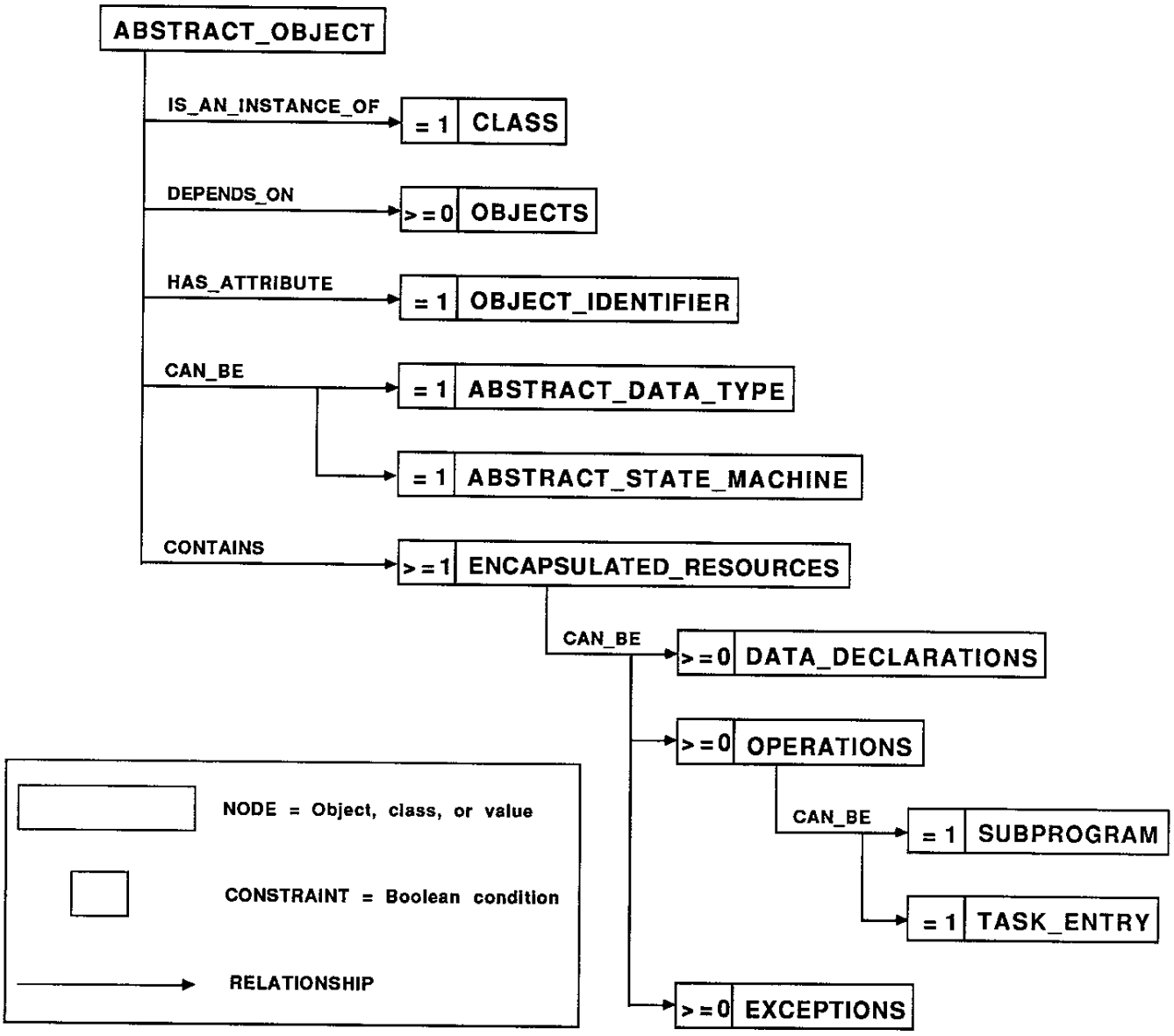

FIGURE 8: SUBASSEMBLY OBJECT SEMANTIC NET

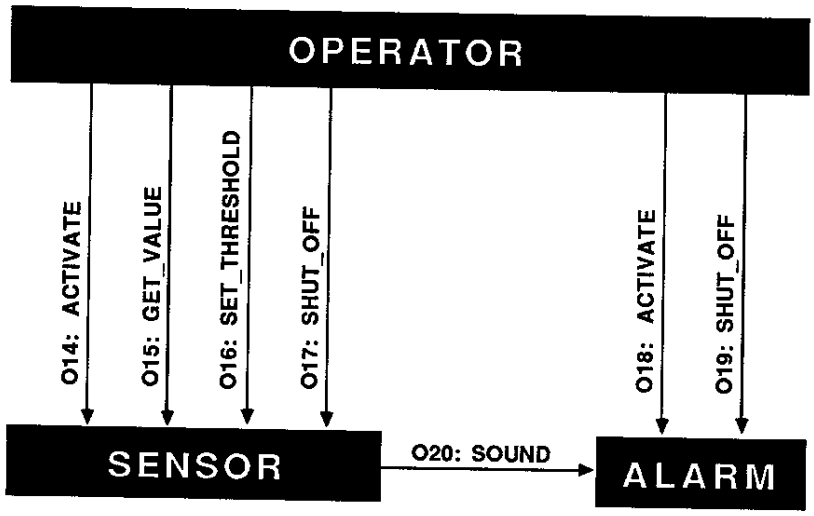

FIGURE 10: SUBASSEMBLY OBJECT_OPERATION DIAGRAM

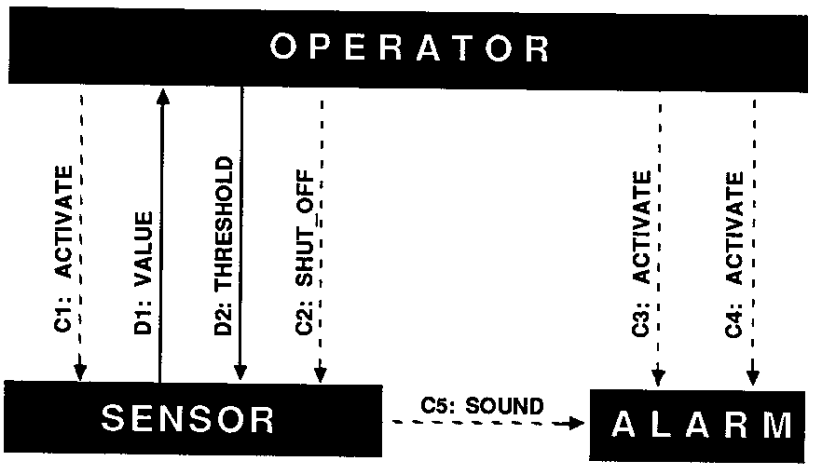

FIGURE 11: SUBASSEMBLY OBJECT DATA/CONTROL FLOW DIAGRAM

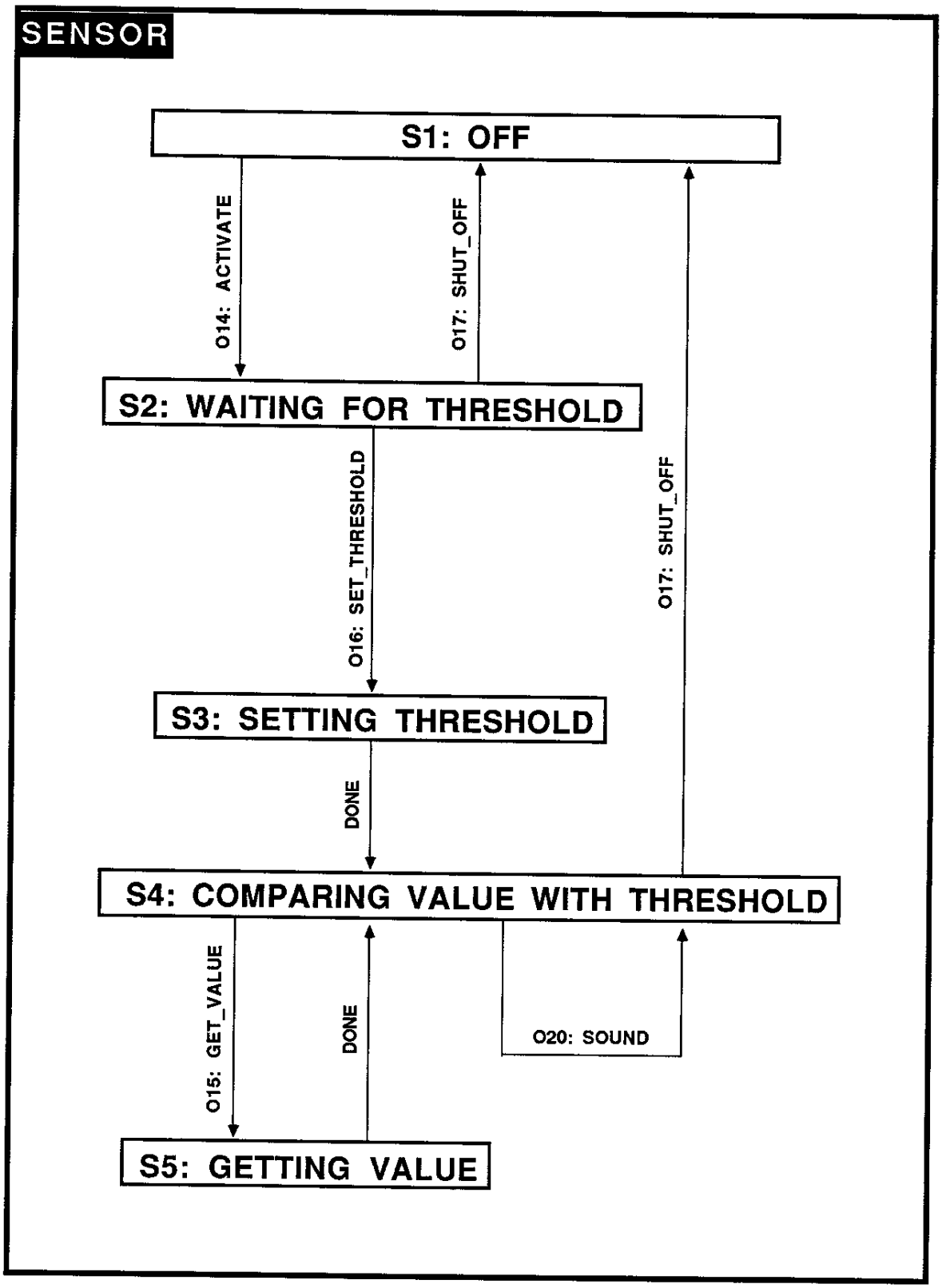

FIGURE 12: OBJECT STATE TRANSITION DIAGRAM

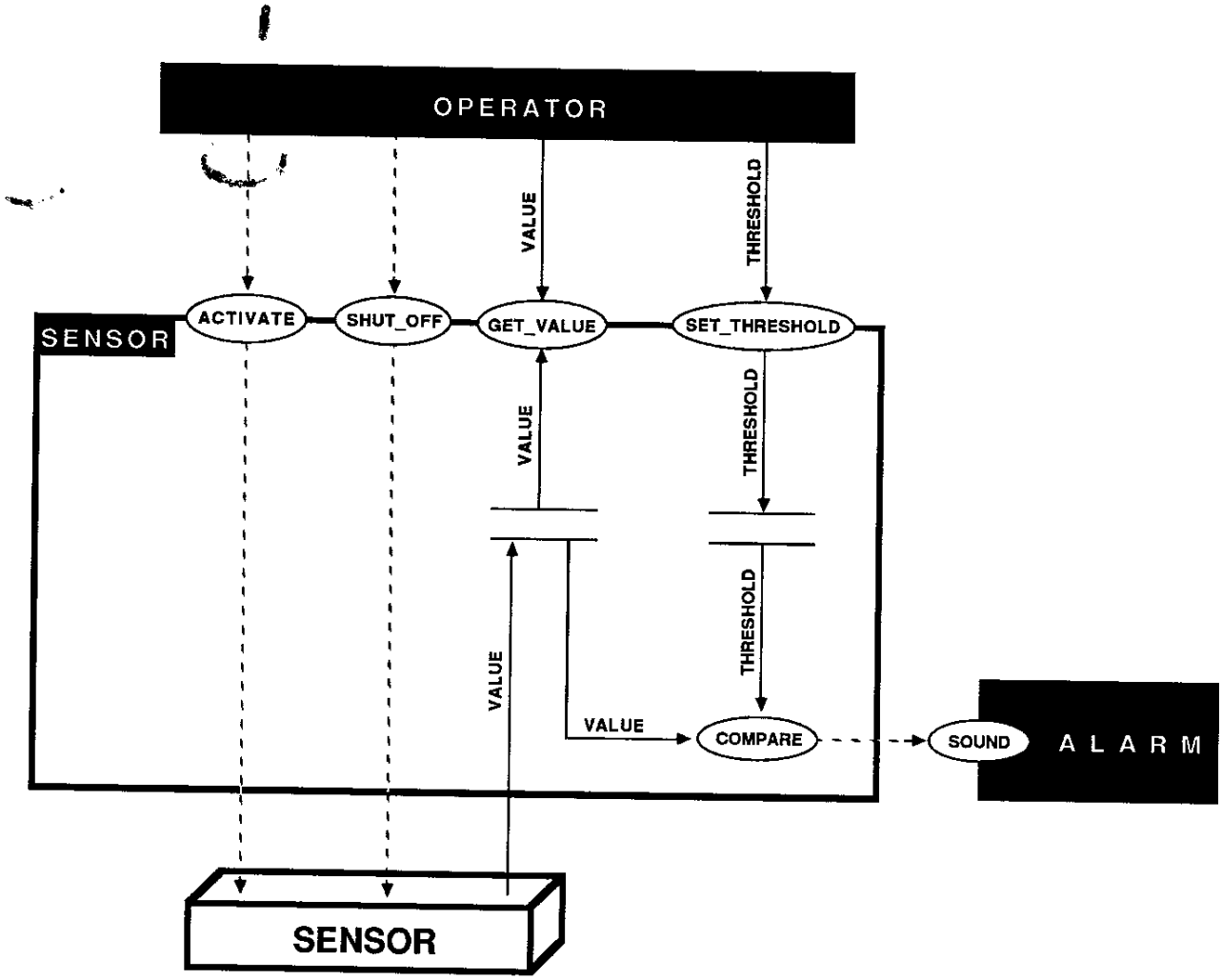

FIGURE 13: LIBRARY UNIT DATA/CONTROL FLOW DIAGRAM (LUD/CFD)