DONALD FIRESMITH

NATIONAL CONFERENCE PROCEEDINGS

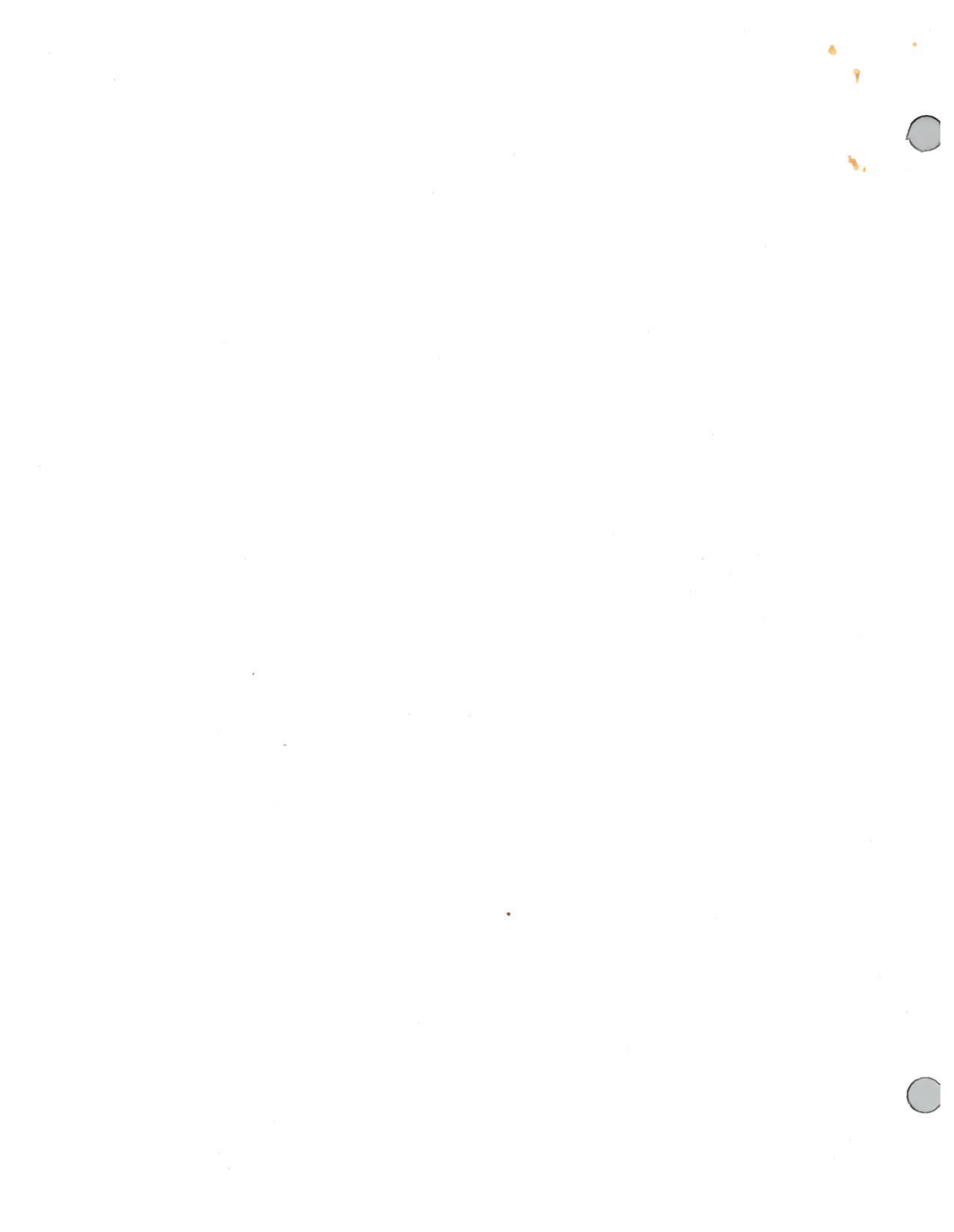
METHODOLOGIES and TOOLS for REAL TIME SYSTEMS

NOVEMBER 14 – 15, 1988

National Clarion Hotel
Arlington, Virginia

Presented by

The National Institute for Software and Productivity
Washington, D.C.



METHODOLOGIES and TOOLS for REAL TIME SYSTEMS

Mack W. Alford
Conference Chairman

TABLE of CONTENTS

Section	Subject	Author
A	**Keynote Address:** *A Perspective on Methodologies and Tools for Real-Time Systems*	Anthony I. Wasserman, Ph.D. Interactive Development Environments
B	*Object-Oriented Systems Engineering*	Mack W. Alford Ascent Logic Corporation
C	*Object-Oriented Systems and Software Engineering Using ESML*	Paul B. Carpenter Honeywell, Inc.
D	*Scenario-Oriented Systems Engineering*	Michael S. Deutsch, Ph.D. Hughes Aircraft Company
E	*Development Environments: Underlying Methodologies, Tools and Training*	Raymond W. Wolverton Hughes Aircraft Company
F	*Using PC-based Tools to Reduce Large Scale Software Development Costs*	Thomas R. Pierpoint Meridian Software Systems, Inc.
G	*An Integrated Tool Set for Host and Target Portable Real-Time Software Development*	P.K. Rowe Multi-Processor Toolsmiths, Ltd.
H	*An Object-Oriented Requirements Specification Method for Ada*	Sidney C. Bailin, Ph.D. Computer Technology Assoc. Inc.
I	*Implementing the DOD-STD-2167 and 2167A Software Organization*	Lewis Gray, Ph.D. TRW FSD
J	*A Successful Use of Ada with the Yourdon Design Method in an Embedded Guidance System*	William L. Miller Rockwell International
K	*Selecting and Using Methods and Tools for Real Time Systems*	R. Pethia, Ph.D; W. Wood, Ph.D. The Software Engineering Institute
L	*AFATDS: Productivity Improvements Using Ada and Object-Oriented Development*	Donald G. Firesmith Magnavox
M	*An Evaluation of CASE Products for Time-Critical Systems: TAGS, Design Aid and Teamwork*	Henry G. Stuebing Naval Air Development Center
N	*A User's Report on the Statemate, Teamwork, PROMOD, TeKCASE and IDE CASE Products*	Homa Taraji Aerospace Corporation
O	*Experience with AdaGraph in the Granite Sentry Program*	Major Eugene C. Bounds U.S. Air Force, SPACECOM
P	*Selecting and Applying CASE Products*	John R. Vosburg Booz•Allen, Hamilton & Co.
Q	*The STARS Program--FY'89*	E. Wald Naval Research Laboratory

AFATDS EXPERIENCE

PRESENTED AT THE NISQP CONFERENCE

METHODOLOGIES AND TOOLS

FOR

REAL-TIME SYSTEMS

NOVEMBER 15, 1988

BY

Donald G. Firesmith

ADVANCED SOFTWARE TECHNOLOGY SPECIALISTS

3418 BROADWAY

FORT WAYNE, IN 46807

(219) 456-9260

firesmit@ajpo.sei.cmu.edu

1) AFATDS Overview

- The Advanced Field Artillery Tactical Data System (AFATDS) is a very large automated command and control system designed for the US Army to support all levels of command from platoon to corps.

- AFATDS implements the full range of field artillery command and control functions as well as fire support, control, and coordination for cannon, rocket, missile artillery, mortars, air support, and naval gunfire.

- The system requirements vary from soft real-time functions to totally administrative functions.

- AFATDS employs an extremely high degree of automation that features advanced decision support tools, information presentation techniques, and concurrent foreground and background operations, etc.

- Contractor:

  Magnavox Electronic Systems Company

- Contact person:

  Mr. Harold (Skip) B. Carstensen,
  AFATDS Software Director,
  (219) 429-5272

- Contract award: May 1984

- AFATDS is a multi-phase project nearing completion of Concept Evaluation Phase (CEP).

- The main software development method was Object-Oriented Development (OOD).

- The general software development philosophy was:

  1. Follow OOD to implement the required capability
     (Get it working).

  2. If necessary, increase object code speed
     (Make it fast enough).

  3. If necessary, reduce the size of the object code
     (Make it small enough).

  If possible, rely on unit-level redesign (to fix obvious inefficiencies) and compiler optimizations to increase speed and decrease size. Do not hand-optimize code for the project compiler unless absolutely necessary and then only if the software is on a critical performance path.

- Host development system:
  - DEC 8800, 8650, 8600, and 11/780 with VAX/VMS
  - DEC Ada compiler and tools
- Target system:
  - Motorola MC68020-based (32 bit) workstation with touch-entry graphics display.
  - Telesoft TeleGen2
- Beta test site for:
  - Motorola M68020 and MMU chips
  - TI 802.5 token ring LAN chip
  - DEC Ada compiler and related tools
  - Telesoft TeleGen2 Ada VAX and 68K Cross Compiler and related tools
  - OOD
  - Ada and Ada PDL

- Training was primarily provided by EVB Software Engineering.

- The training was of high quality, but expensive and difficult to tailor for consistency with company standards, etc.

- AFATDS was Magnavox's first significant Ada project and first large software system project.

- "The AFATDS program ... is a sound technical success for Ada. ... Magnavox, the AFATDS contractor, has been able to use Ada throughout the design and development phases to support its software engineering methods. ... [However,] the development team has been brought to a total standstill on dozens of occasions by reviewers, [GAO] investigators, and IV&V (Independent Verification and Validation) contractors."
  Ralph E. Crafts, Editor
  Ada Strategies, March 1988

- AFATDS consists of a set of 7 major hardware components and the following 9 software configuration items:

  - Execution Environment:

    Operating System
    Data Management
    Information Management
    Communications Support
    Communications Interface

  - Applications Software:

    Fire Support Planning
    Fire Support Execution
    Movement Control
    Common Functions

- With the exception of less than 3 KSLOC of operating system and communication software, AFATDS was written completely in Ada.

- AFATDS can be configured from a single hardware component to a large center contained in several tactical vehicles.

AFATDS SOFTWARE SIZE

REL.	DATE	FILES	LINES	NCNBs	TSCs
4.04	25 FEB 88	7,553	2,517,594	1,175,498	409,982
4.03	16 DEC 87	7,428	2,495,749	1,160,332	401,933
4.02	03 SEP 87	6,691	2,320,000	1,061,487	370,469
4.01	MAY 87	5,109	1,806,101	819,792	278,085
3.0	JUL 86	?	?	494,000	?
2.0	APR 86	?	?	254,000	?
1.0	FEB 86	?	?	51,000	?

LINES = PHYSICAL LINES

NCNBs = NON-COMMENT, NON-BLANK PHYSICAL LINES

TSCs = TERMINAL SEMICOLONS (DETERMINED BY COUNTING ALL SEMICOLONS EXCEPT THOSE IN COMMENTS, QUOTED STRINGS, AND FORMAL PARAMETER LISTS -- NOTE: THIS IS NOT THE SAME AS ADA STATEMENTS)

AFATDS PRODUCTIVITY (NCNB SLOC)

	RELEASE	1	2	3	4.01	4.02	4.03	4.04	TOTAL
△ SIZE	EXEC. ENV.	?	?	?	?	20,907	-4,129	2,752	339,162
	APPLICATIONS	?	?	?	?	251,922	102,974	11,881	789,775
	AFATDS TOTAL	51,000	203,000	240,000	248,630	272,829	98,845	14,633	1,128,937 *
DAYS	EXEC. ENV.	N/A	N/A	N/A	4,245	1,730	1,136	588	13,769
	APPLICATIONS	N/A	N/A	N/A	7,019	5,307	3,179	1,589	24,461
	OTHERS	N/A	N/A	N/A	1,291	733	766	392	5,169
	AFATDS TOTAL	6,116	3,983	5,325	12,555	7,770	5,082	2,569	43,400
SLOC / DAY	EXEC. ENV.	?	?	?	?	12	-4	5	25
	APPLICATIONS	?	?	?	?	47	32	7	32
	AFATDS TOTAL	8	51	45	20	35	19	6	26

EXECUTION ENVIRONMENT - OPERATING SYSTEM (52 KSLOC)
DATA MANAGMENT (88 KSLOC) * INCLUDES 36.5 KSLOC SUBCONTRACTED
INFORMATION MANAGEMENT (140 KSLOC)
COMMUNICATIONS SUPPORT (52 KSLOC)
COMMUNICATIONS INTERFACE (44 KSLOC)

TOTAL (376 KSLOC)

APPLICATIONS SOFTWARE - FIRE SUPORT PLANNING (276 KSLOC)
FIRE SUPPORT EXECUTION (254 KSLOC)
MOVEMENT CONTROL (48 KSLOC)
COMMON FUNCTIONS (212 KSLOC)

TOTAL (790 KSLOC)

OTHERS - SOFTWARE MANAGEMENT
SOFTWARE TEST
RELEASE INTEGRATION AND TEST
SOFTWARE QUALITY ASSURANCE
DATA ITEM SECURITY SCHEME
SCREEN DESIGN SUPPORT

REUSABLE SOFTWARE NOT DEVELOPED ON THE PROJECT (I.E., GRACE, MATH PACKAGES) WAS NOT COUNTED!
REUSABLE SOFTWARE DEVELOPED ON THE PROJECT WAS ONLY COUNTED ONCE!

DAYS - 8 HOURS BILLED

SOURCE - MS. KAREN SIVLEY (DETAILED PAPER IN PROGRESS)

2) AFATDS Lessons Learned

- "Ada is being successfully used today in military programs, such as AFATDS." Report of the Defense Science Board Task Force on Military Software, September 1987

- Management support for Ada is essential. Innovative software development methods are less likely to be implemented without active management support at all levels.

- Low and mid-level technical managers must remain technically current on projects involving a new software development method, a new language, and a new mindset. They must not become bogged down in micro-scheduling and status reporting.

- Software engineers must be an integral part of the system engineering effort.

- Plan on spending more effort on requirements analysis and less effort on integration/testing than is typical on projects using classical methods and the "waterfall" life-cycle. According to Skip Carstensen, effort on a large Ada project can be allocated as follows: 55% to requirements analysis and design, 10% to coding, and 35% to testing and integration.

- Increased government/user participation benefits the project.

- Pertinent government and IV&V personnel must receive training in software engineering, the project software development method (if state-of-the-art), and Ada.

- Spend adequate time and money prior to project initiation to:

  1. Determine the appropriate software development method.

  2. Develop software standards and procedures (e.g., Ada coding standards).

  3. Train technical management and the developers in software engineering, the project software development method, the software development environment, and Ada.

- Because ongoing training in the project software development method and Ada (e.g., via Help Desks) is very cost-effective, designate adequate personnel to provide such expertise.

- Use an Ada-oriented software development method (e.g., OOD).

- OOD is not compatible with:

  1. Functional decomposition methods.
  2. Obsolete DoD development standards (e.g., DOD-STD-2167 and MIL-STD-1521) based on the classic "waterfall" life-cycle.

- OOD naturally produces a very large number (over 90%) of very small and simple Ada programming units (McCabe's metric less than 4).

- Such units cause far fewer problems during integration and test.

- With the use of Ada for design as well as coding, PDL's as such are no longer needed.

- Ada is not compatible with the DIDs associated with certain DoD standards (e.g., DOD-STD-2167, MIL-STD-483, and MIL-STD-490). Attempting to conform to such standards in a rigorous manner on a modern Ada project forces numerous convolutions and wastes a great deal of time. Such standards are obsolete and do not lend themselves to being adapted to modern software engineering practices. For example, one must document the inputs and outputs of the standard Ada "Calendar_IO" and "Text_IO" packages because these standards require the description in detail of all unit inputs and outputs.

- MIL-STD-483 and MIL-STD-490:

  1. Do not support either modern software engineering or OOD.

  2. Greatly decrease developer productivity without improving software quality.

- Proper (i.e., Ada- and method-oriented) Software Development Files (SDFs) are better than traditional C5s for documenting Ada designs.

- Evaluate each compiler vendor's commitment to supporting your project. How often will updates be provided? Will your input affect the vendor's priority with regard to fixes? How easy will it be to report problems, get responses? What will the vendor's response time be on getting a fixed product?

- Test and benchmark compilers, etc. prior to making a decision.

- Do not base your decisions on the verbal promise of a vendor.

- Designate someone to act as the contact person dealing with (and the expert on) compilers, tools, and their vendors.

- The lack of user friendly automated tools can significantly increase cost and schedule.

- Tools must support the project software development method.

- The use of beta site versions of immature compilers required more computing power and disk space than we imagined possible. For example, we needed fourteen 400 megabyte disks.

- The pragma INTERFACE feature of the Telesoft TeleGen2 compiler works extremely well with small assembly routines.

- Due to compiler immaturity, certain Ada constructs and procedures may be rewritten to provide the same capability with significantly less memory required. Style changes resulted in an overall reduction of 20 to 50 percent in object code size of certain units. Note that this is extremely compiler-dependent.

- Compiler optimization is critical. The use of an optimizer should provide at least 10 to 30 percent object code size reduction. One should be able to reduce the object code size of imported (i.e., "withed") packages another 5 to 15 percent with a smart linker. Note that this is also very compiler- and time-dependent.

- One of the most difficult management jobs when dealing with the Ada technology transition is achieving a cost-effective balance between sufficient:

  - Control through the use of sound engineering methods.

  - Flexibility to immediately adapt and apply the numerous lessons learned.

- Significant reuse is not only practical, but also necessary if one is to meet cost and schedule constraints on a very large project.

- AFATDS has proved the practicality of Ada reuse. The reuse of Ada software on AFATDS meant that the developers did not have to develop over 100,000 lines of deliverable code. Approximately 13 percent of the total delivered software was reused software for a savings of slightly under 190 man months or 2 calendar months off the overall project schedule.

- There are significant places in a major project where very similar functionality is being performed. Major portions of software developed can be reused with only minimal modification or addition.

- Based on our experience, the use of OOD made reuse easier.

- In order to make maximum use of the reusability that OOD provides, one would have to use OOD from the very beginning of the project. Because the DoD is entrenched in functional decomposition, one may be forced to wait until the CSCI level to initiate OOD.

- There is a strong tendency for developers to think that they can develop better software than that in the reuse libraries. The real reason for this is often considerably more psychological than technical due to developer pride and fear of the unknown.

- We can easily increase the amount of software being reused to 25 percent on the next similar project, and 50 percent reuse on future projects appears possible.

3) Information Sources

- "Experiences in Achieving Size and Performance Requirements on a Large Ada Project",
  Harold B. Carstensen, Jr.,
  Computers in Aerospace Conference,
  June 1987

- "Experiences in Delivering a Large Ada Project",
  Harold B. Carstensen, Jr.,
  Military Computing Conference,
  5 May 1987

- "A Real Example of Reusing Ada Software",
  Harold B. Carstensen, Jr.,
  Fourth National Conference on Methodologies and Tools,
  2 March 1987

- "The Management Implications of the Recursive Nature of Object-Oriented Development",
  Donald G. Firesmith,
  AdaEXPO/SIGAda Conference,
  7-11 December 1987

- "Transitioning Developers to Ada",
  Donald G. Firesmith,
  Second Annual ASEET Symposium,
  9-11 June 1987

- "The ETAS Central Processor: A Case Study in Reusable Software",
  Richard A. Howard,
  Tactical Communications Conference - 88,
  3-5 May 1988

- "Experience and Lessons Learned in Transporting Ada Software",
  Karen E. Sivley,
  Joint Ada Conference,
  16-19 March 1987

- "Development Software Configuration and Integration in a Large Ada Project",
  David H. Ternes,
  SIGAda Conference,
  9-11 December 1987

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) <u>Develop the Subassembly Logical Design</u>. 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) <u>Iterate back to upgrade the subassembly requirements graphics</u> as necessary based on the Subassembly Logical Design.

1.5) <u>Analyze all subassembly abstract objects and classes</u>. 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.

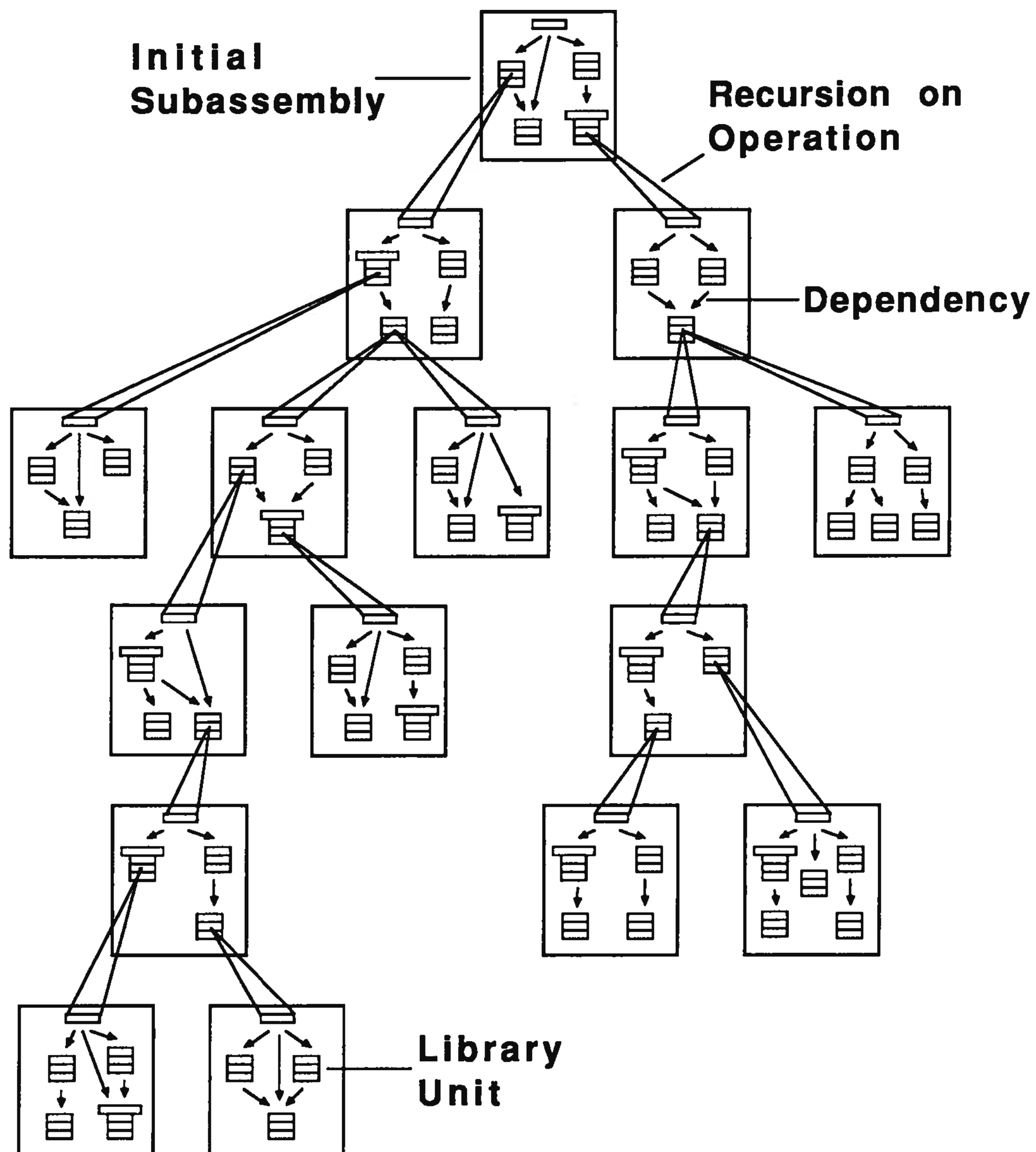


FIGURE 1: ASSEMBLY LIBRARY DIAGRAM (ALD)

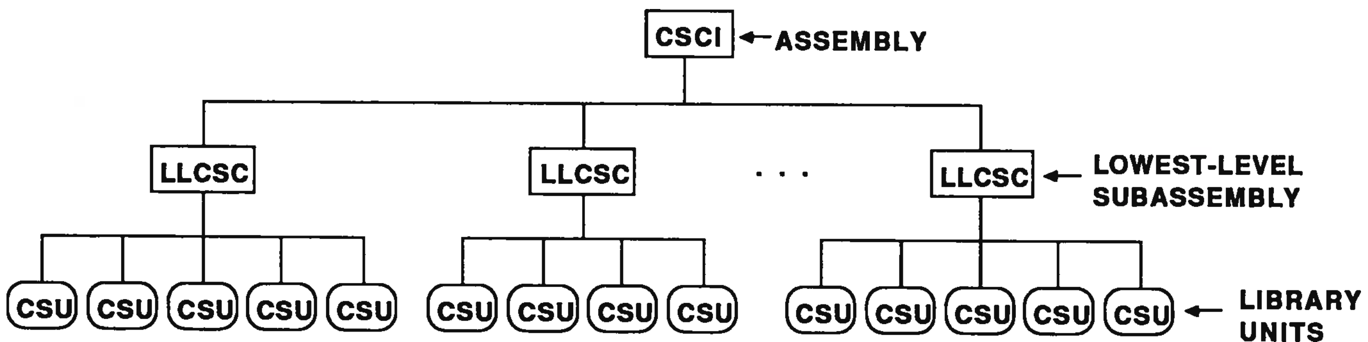


FIGURE 2: DOD_STD_2167A SOFTWARE ORGANIZATION

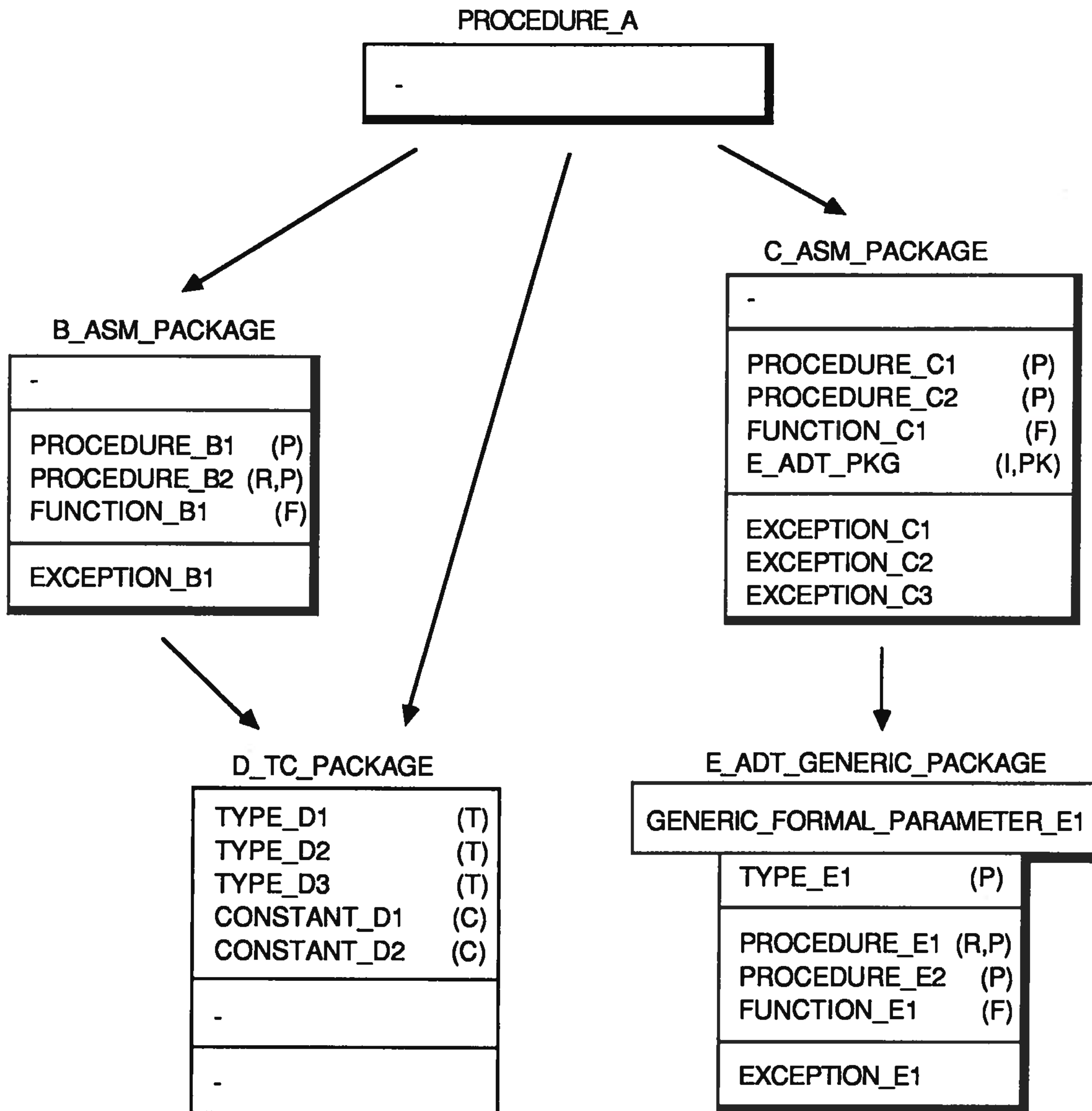


FIGURE 3: SUBASSEMBLY LIBRARY DIAGRAM (SLD)

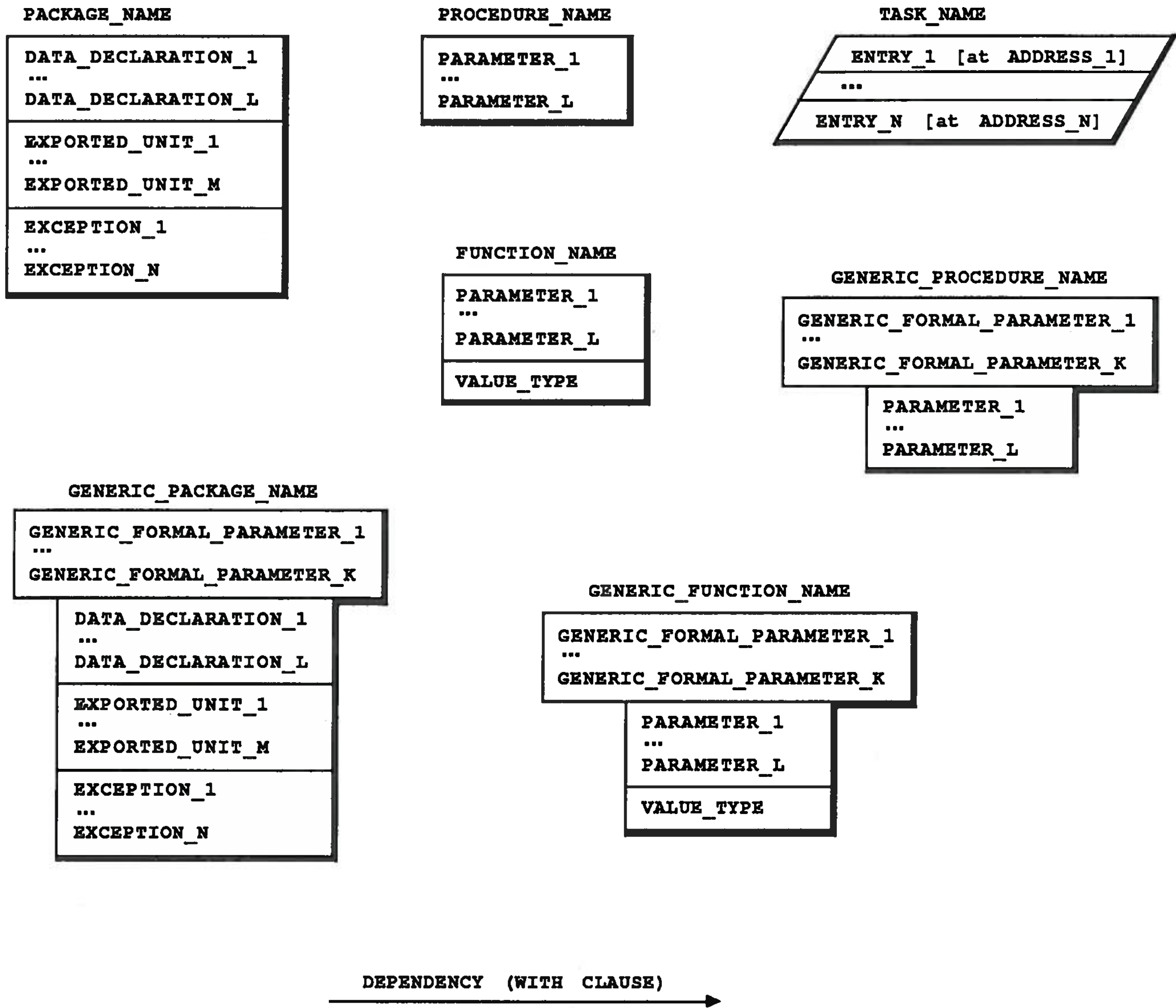


FIGURE 4: ADA LIBRARY GRAPH ICONS

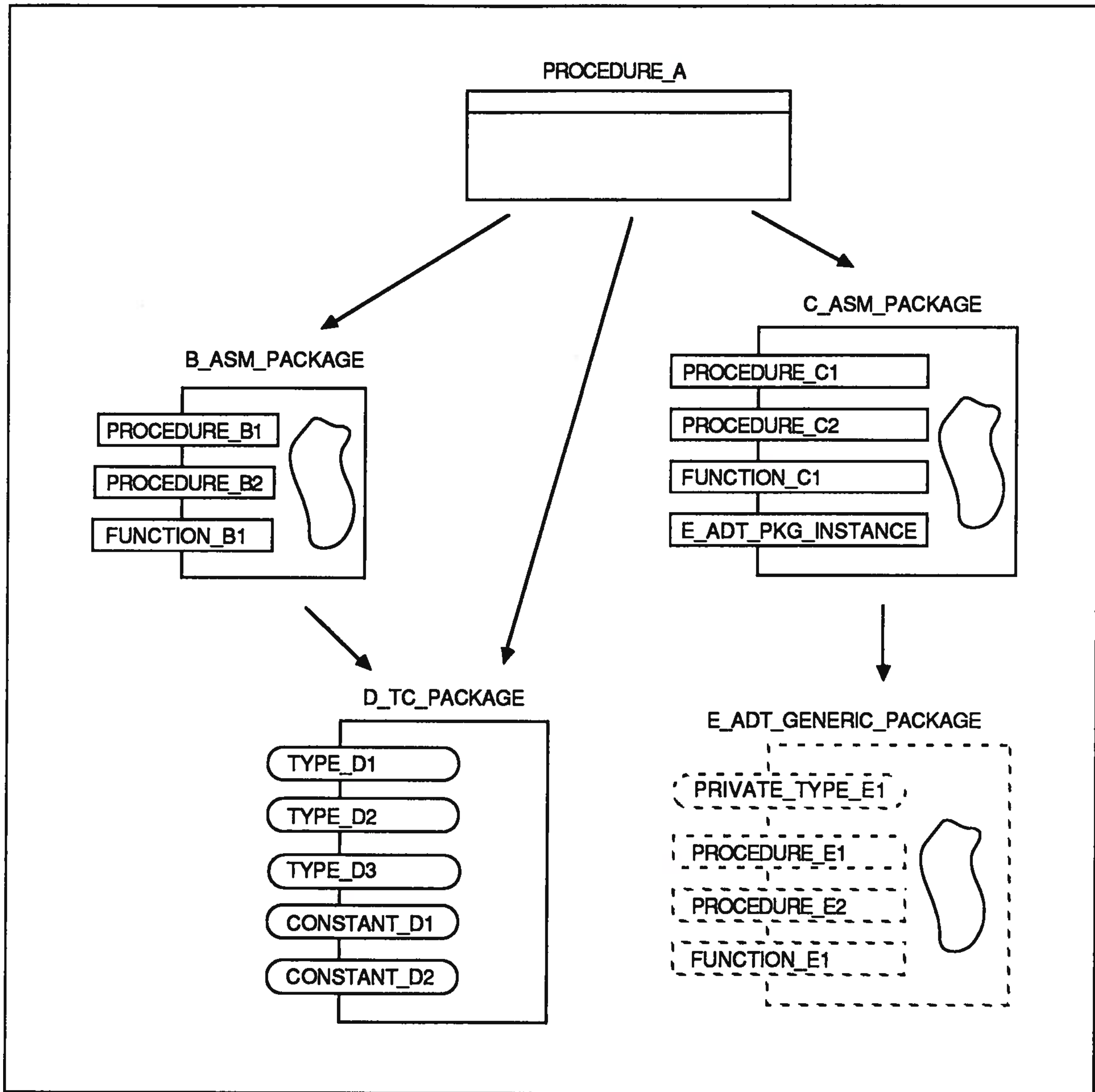


FIGURE 5: BOOCH DIAGRAM

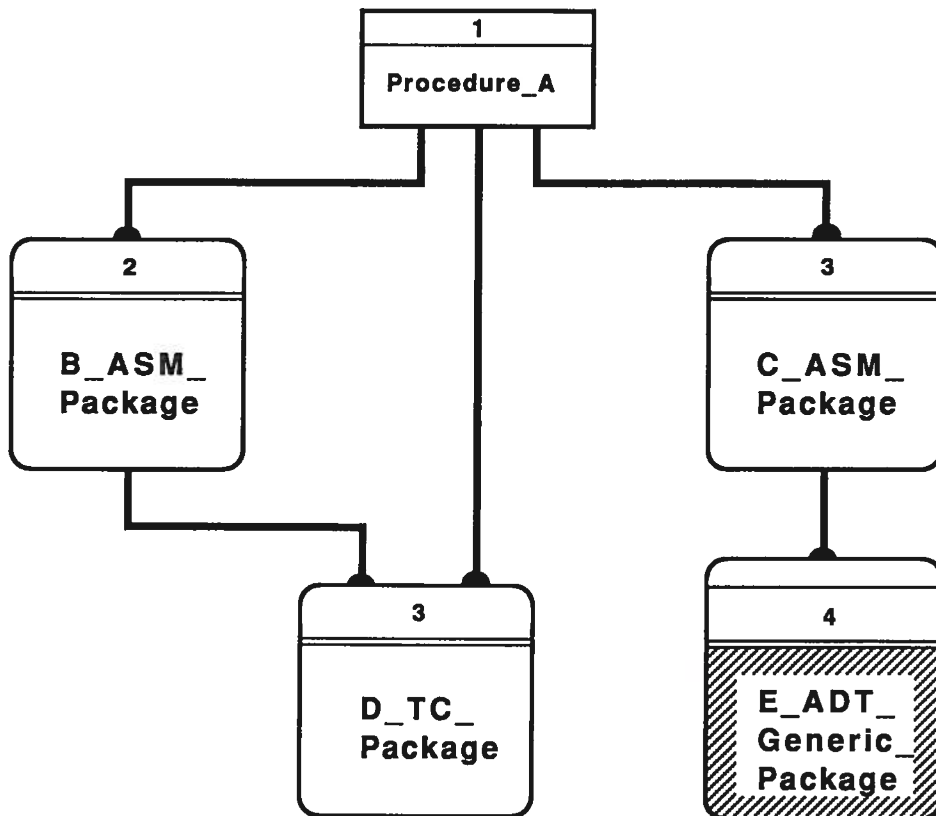


FIGURE 6: SCOOP3 LIBRARY GRAPH

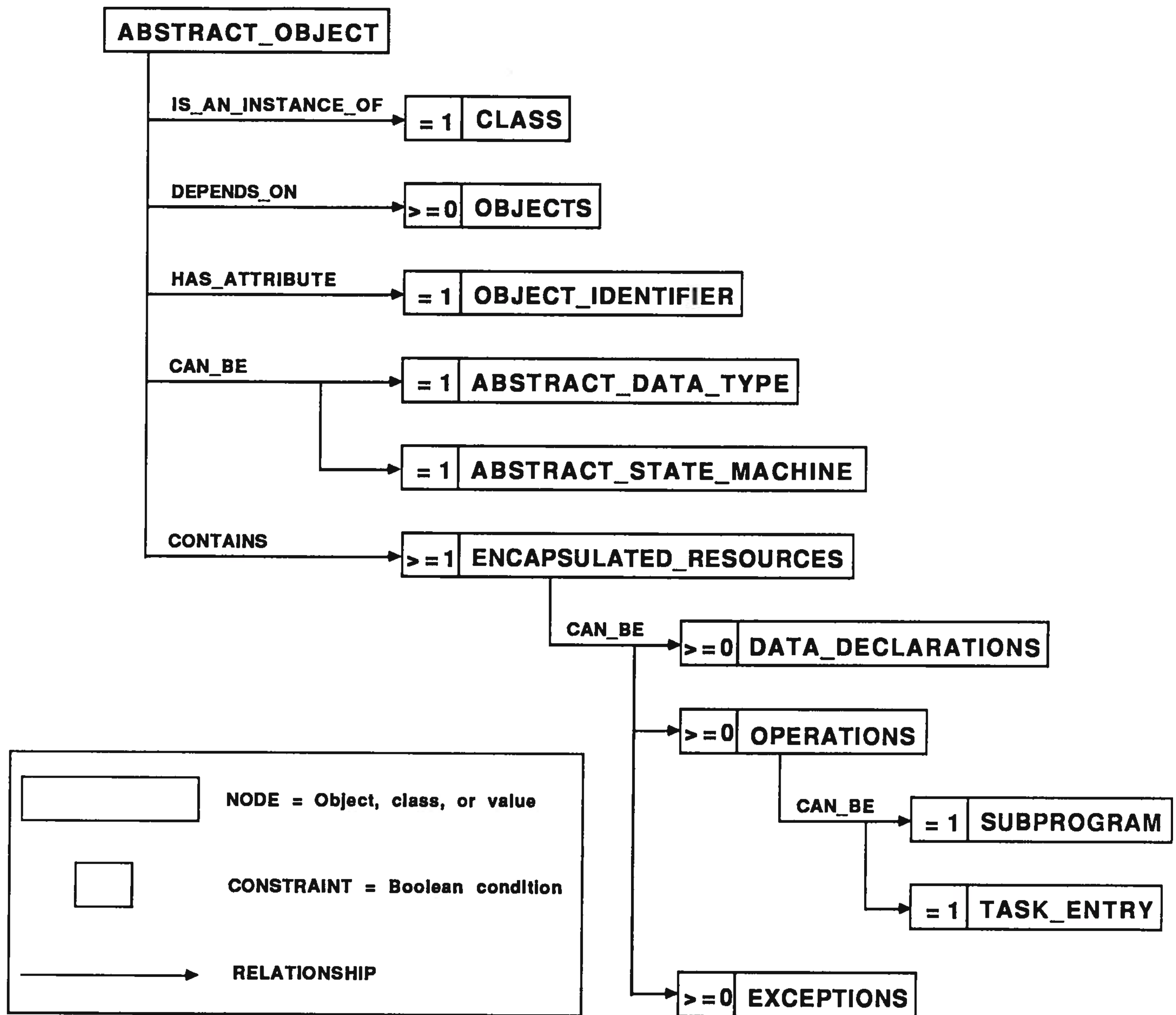


FIGURE 8: SUBASSEMBLY OBJECT SEMANTIC NET

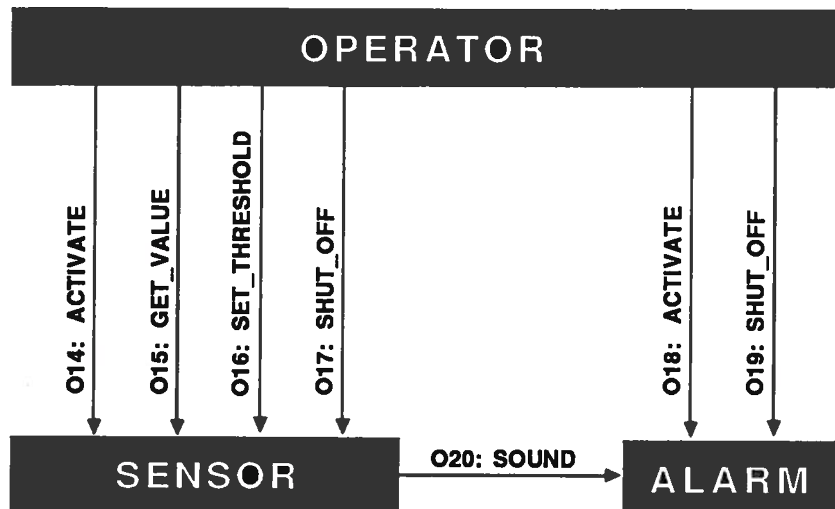


FIGURE 10: SUBASSEMBLY OBJECT_OPERATION DIAGRAM

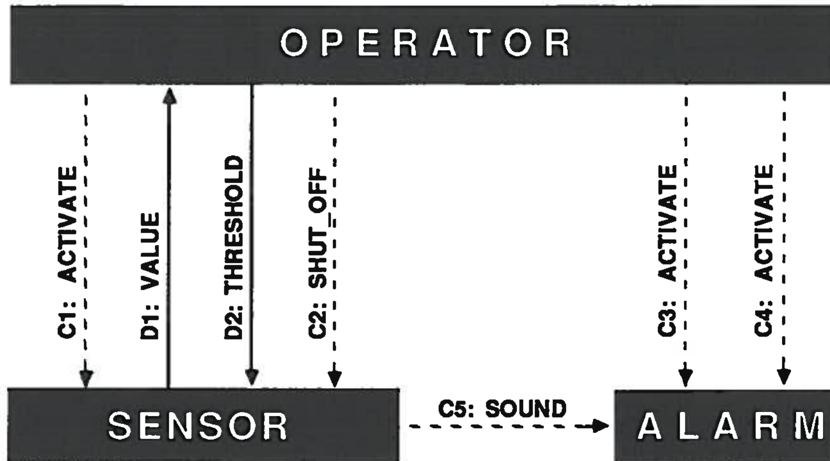


FIGURE 11: SUBASSEMBLY OBJECT DATA/CONTROL FLOW DIAGRAM

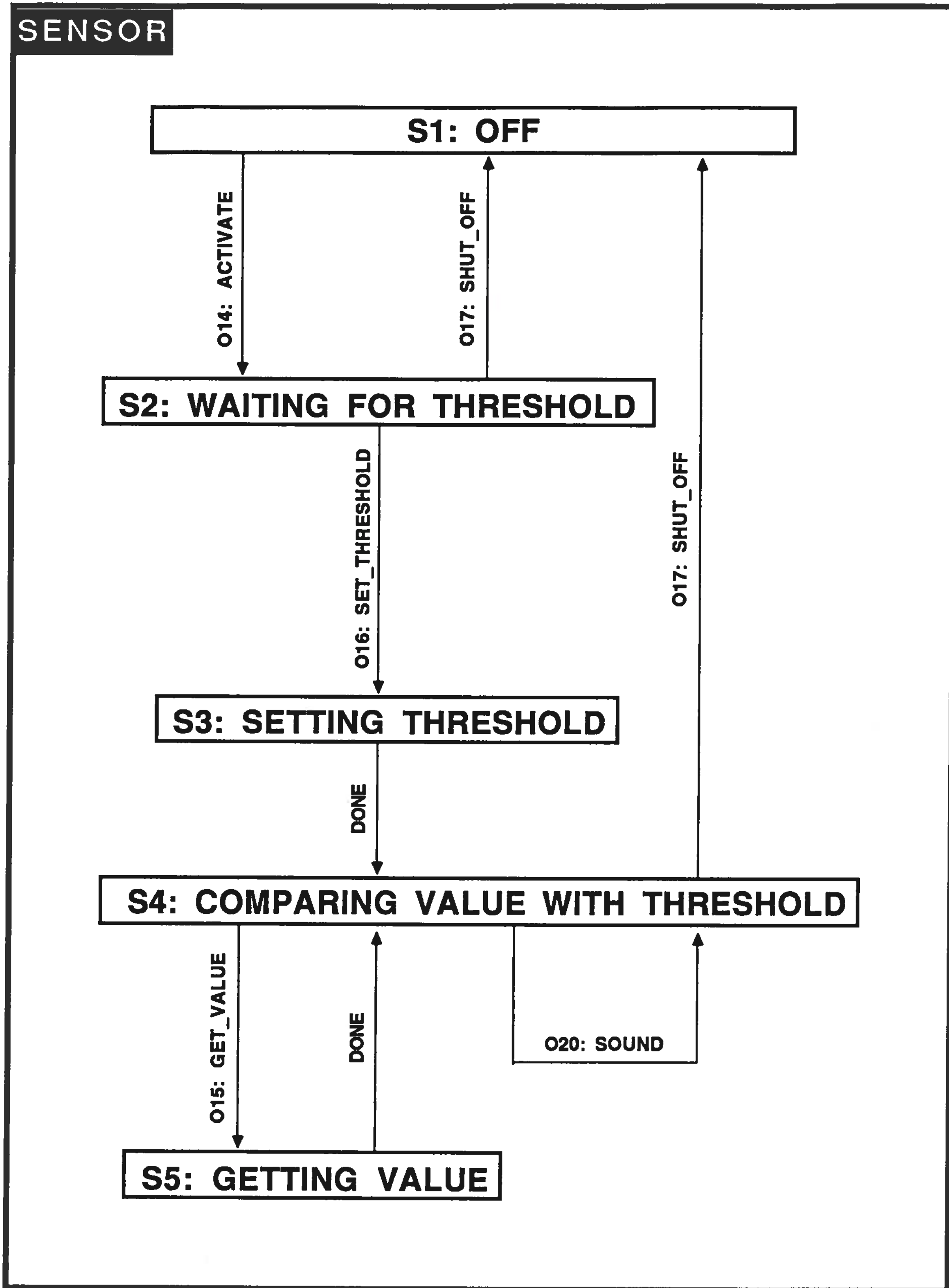


FIGURE 12: OBJECT STATE TRANSITION DIAGRAM

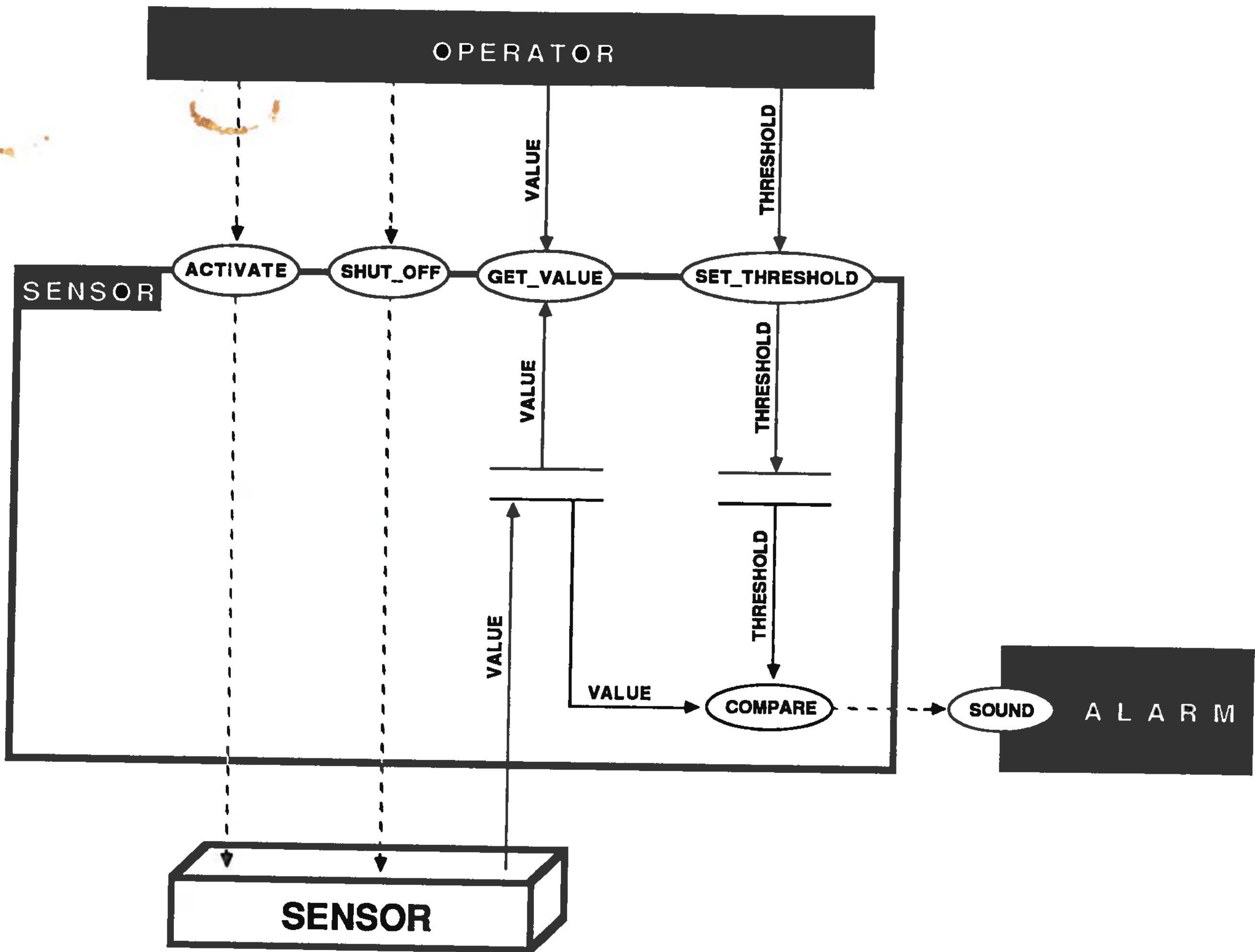


FIGURE 13: LIBRARY UNIT DATA/CONTROL FLOW DIAGRAM (LUD/CFD)