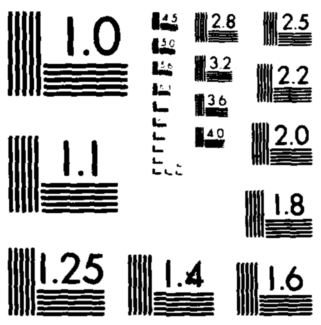
AD-A109 748 TEXAS INSTRUMENTS INC LEWISVILLE EQUIPMENT GROUP F/G 9/2
ADA INTEGRATED ENVIRONMENT III.(U)
DEC 81 F30602-80-C-0293
UNCLASSIFIED RADC-TR-81-361 NL

1 of 1
AD A
109748

END
DATE
FILMED
2-82
DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU

PHOTOGRAPH THIS SHEET

ADA109748

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Texas Instruments Inc.,
Lewisville, TX Equipment Group - ACSL
ADA Integrated Environment III
Dec. 81

DOCUMENT IDENTIFICATION

Contract F30602-80-C-0293 RADC-TR-81-361

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR		
NTIS	GRA&I	☒
DTIC	TAB	☐
UNANNOUNCED		☐
JUSTIFICATION		
BY		
DISTRIBUTION /		
AVAILABILITY CODES		
DIST	AVAIL AND/OR SPECIAL	
A		

DISTRIBUTION STAMP

DTIC
ELECTE
S JAN 19 1982 D
D

DATE ACCESSIONED

82 01 12 006

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

8

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

RADC-TR-81-361
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT III

Texas Instruments Incorporated

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

RADC-TR-81-361 has been reviewed and is approved for publication.

APPROVED: Elizabeth S Kean
ELIZABETH S. KEAN
Project Engineer

APPROVED: [illegible]
JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss
JOHN P. HUSS
Acting Chief, Plans Office

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-361	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ADA INTEGRATED ENVIRONMENT III		5. TYPE OF REPORT & PERIOD COVERED Interim Report 15 Sep 80 - 15 Mar 81
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0293
9. PERFORMING ORGANIZATION NAME AND ADDRESS Texas Instruments Incorporated Equipment Group-ACSL, P O Box 405, M.S. 3407 Lewisville TX 75067		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62204F/33126F/62702F 55811919
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441		12. REPORT DATE December 1981
		13. NUMBER OF PAGES 65
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Elizabeth S. Kean (COES)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada	MAPSE	AIE
Compiler	Kernel	Integrated environment
Database	Debugger	Editor
KAPSE	APSE	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This report describes the rationale of the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an APSE is

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this report include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

TABLE of CONTENTS

APPENDIX B REFERENCES

SECTION 1

INTRODUCTION

1.1 Introduction

This Technical Report supplements the Ada Integrated Environment System Specification and Computer Program Development Specifications. It provides some general design philosophies and criteria, discusses key design issues, and highlights the design decisions made for various components of the system.

The concept of "programming in the large", or system construction programming, was selected as a unifying theme for the system design . This concept emphasizes the fact that software design and development activities take place on at least two levels and that different entities and software tools are important on each level. Some of the differences between levels are summarized in the following table.

Lower level	Higher level
Algorithms	Program units
Data structures	Database objects
File contents	File attributes
Programming language	Command language
Text editing	Program binding

Tools that work with the lower-level entities are relatively well understood. In this design, additional emphasis has been placed on understanding the higher level entities and providing tools that work with them, especially in the areas of configurations, versions, and libraries. In the same spirit, the functionality of software tools has been carefully examined to separate the low-level functions from higher-level ones, thereby improving control of the software construction process.

Certain other system-wide concerns must also influence the design of the Ada Integrated Environment. The issues of versatility (rehosting and retargeting), durability (low maintenance cost), and performance are critical to all APSE tools and to the software they produce.

1.2 A Model of the Software Development Process

The "programming in the large" view of the software development process considers it to be a sequence of integrating and transforming steps that begins with the simplest lexical units of a programming language and progresses to produce a complete program.

At each level of integration, constituent units are brought together by a software tool to form a larger entity. These integrating tools are primary components of the toolset.

- * A text editor is an example of a tool used to form program units from smaller constituent parts. The editing process may include direct text entry, deletion or changes by an interactive user or may merge text from several source files to create the desired program unit.
- * A program binder (or linkage editor) is an example of a tool used to form a complete program from specified constituent program units. The binding process may merge program units from several libraries to create the desired program.

Various transformations may be performed on the constituent units of a software product at any level of integration. Transformation of a program or program unit changes its representation without changing its meaning.

- * A translator is an example of a tool used to transform the declarations and statements of a program unit from the human-written source text of a programming language to an intermediate representation more suitable for analysis, optimization and other processing.
- * An optimizer is an example of a tool used to analyze and transform a program unit to improve its performance or its use of computing resources.
- * A code generator is an example of a tool used to transform a program unit from an intermediate representation to a form compatible with the instruction set architecture of a target machine. This tool usually performs some additional optimizations.
- * A composite transformation tool consisting of a translator, optimizers, and a code generator is usually called a compiler.

Static and dynamic analysis tools may be applied to the constituent units of a software product at any level of integration.

- * A cross reference analyzer is an example of a static analysis tool that locates the de[illegible]ition of each symbol in a program unit and identifies the [illegible]ram [illegible]tatements that refer to the symbol.

* A source language level debugger is an example of a dynamic analysis tool that maps the memory image of an executing Ada program to the source program text and data definitions, allowing a user to examine or modify data values and control program execution. Configuration management tools attempt to record and control the changes made to constituent units of a software product so that (re)construction of a product is consistently done from known, compatible parts. These tools use a database that provides data structures and facilities for the storage and retrieval of information, generated by MAPSE tools, concerning the constituent units of a software product.

The Ada language provides several features that support *separate compilation* of program units. As a result, construction of a correct context for compilation of a program unit may require access to other program units in the database. The relationships between program units are defined by the static lexical (nested) structure of the program.

The several steps of the compilation process produce intermediate results which must be stored. Listings and data for debuggers or other analysis tools may also be generated, and these must be related to the original source text.

A special file is necessary to specify the structure of a program, name its constituent program units, and provide the mapping from internal program unit names to the external names of database objects. This file is called a library file. It also contains information on the compilation status of each program unit and the names of derived files. The library file is thus a key element in Ada software configuration management.

An integrated software development process must begin with specification of a program structure, which then controls all subsequent construction, transformation and integration of constituent units to form the final program. Program structure information is stored in the library file.

Each of the software tools described in this model obtains its processing instructions from a control file, which may be an interactive user's keyboard in appropriate cases. One or more program units may be processed as input. The processing context for each program unit is determined by reference to the library file. Upon completion of processing, the library file may be updated to show appropriate results. The output -- an integrated or transformed program unit -- is stored in the database with appropriate attribute values and relations set to indicate its position in the set of derived program products.

1.2.1 Database Management Principles

Individual database objects must be managed at several levels. Standard facilities must be provided to identify and manipulate groups of database objects for version control, configuration definition, program libraries, and archiving.

1. At the lowest level are the contents of a database object. This information is created by a running program; it requires storage. The internal strucuture of a database object is defined by the programs that create and use it.

2. At the next level are the attributes of a database object. This information about the object is obtained from various sources. It is usually kept in a separate data structure, not with the object.

3. At the highest level are the relations between database objects. This information linking two or more objects together is user-defined and represents possible static or dynamic interactions between the contents of database objects as they are ceated, used, changed and destroyed.

1.2.2 Control of Software Tools

Each APSE software tool must have a well-defined control file for input. The functionality of each tool should be clearly defined; addition of features outside the tool's designed function should be limited or eliminated.

1. The control file contains the commands that specify the processing to be performed on an input file for a particular invocation of a software tool.

2. Generally, commands to the software tool should not be interspersed with input data. Separation of the two allows additional flexibility and allows changes of commands or options without changing the input data.

3. Use of features such as the INCLUDE pragma in a compiler should be discouraged. In this case, a transformation tool is forced to serve as an integrating tool; a command is embedded in the input text file, and configuration management may be compromised.

1.2.3 Data Structure Interfaces

The software interfaces to major system data structures, such as directories, must follow standardized design rules. All APSE tools and user programs must use the standard interfaces.

1. The lowest level interface maintains and protects the data structure. It controls all access, and it stores and retrieves records on demand. User programs have no direct access to this level.

2. The next level provides a set of primitive functions to handle

data inquiries and storage requests. Routines at this level can locate a requested record in the data structure and can insert or extract detailed information in those records in response to user requests. These are generally the routines visible to an Ada program through standard interface packages.

3. APSE components and user programs at higher levels use the primitive functions furnished by the APSE to construct subprograms that perform more complex operations. Some of the most commonly used subprograms are integrated into standard packages and libraries for all users.

1.2.4 Command Language Principles

Command languages should provide much of the generality of programming languages, but must also provide convenient access to special functions. All command language designs should anticipate mixed interactive and non-interactive use.

1. The decision to design all APSE tools to use a control file implies a collection of "command languages" in which the contents of these files are written. Well known examples are text editor commands, debugger commands, compiler pragmas, and link editor control files.

2. Command languages are used to invoke and control processes, select options, and manipulate whole data objects. They are typically concerned with constants and literals -- the names of objects, processes and parameters.

3. Programming languages are used for algorithmic manipulation of simple data objects and are typically concerned with strongly typed variables, expressions, and the internal detail of structured data objects.

4. The concepts of statements, control structures, and subprograms are significant and similar in both kinds of languages. The use of parallel and pipelined processing is more common in command languages.

5. Immediate interaction with a human user is important in a command language. Programming languages, which tend to be used non-interactively, must emphasize readability.

6. The specialized nature of a command language leads to a large vocabulary of specific keywords and keyword-specific command syntax. Programming languages use a smaller vocabulary of very general keywords and a generalized, limited syntax.

1.2.5 Definition of Program Structures

Tools must be provided to record the static (lexical) and dynamic (run-time) structure of an Ada program, name its constituent program units, and provide the mappings from internal program unit names to the internal and external names of database objects.

1. The recorded structure definition is called a library file. It is a key element in Ada software configuration management. It should be used to specify the processing context of each Ada program unit through all construction, transformation and integration steps.

2. The derivation of each program unit, its compilation status, the names of derived files, and other attributes and relations may be kept in the general database or in the library file.

3. Each software tool involved in the processing of an Ada program unit is responsible for initializing or updating the values of the attributes and relations that are associated with the database objects it creates or modifies.

1.2.6 *Common Use of Software Products*

The by-products of the transformation and integration tools that operate on Ada program units must be designed for general use by other software tools.

1. Certain compilation by-products, such as listings and symbol tables, are used by analysis tools such as source language level debuggers or symbolic cross reference analyzers.

2. Intermediate representations of program units may be subjected to special analysis or used to produce filtered source texts.

3. The same design principles that apply to interfaces with major system data structures must be applied to the database objects produced by Ada language processing tools.

SECTION 2

ADA SOFTWARE ENVIRONMENT

2.1 Program Segmentation

"KAPSE/MAPSE software shall be designed to be *modular* and reusable. Software performing a single function required (or potentially required) by more than one system component shall be designed to be reusable to the maximum extent possible." [RADC80].

Support of this requirement was the *fundamental* design goal for the Ada Software Environment in general and its program segmentation technique in particular. The next subsection provides the background for the ASE segmentation; subsequent subsections describe its capabilities.

2.1.1 Background

The program structure and separate compilation facilities of Ada provide the basic tools with which to produce modular software. The degree to which software can be reused depends on implementation decisions within the Ada Software Environment. The modes for sharing software are:

1. Source text -- The Ada program *library* permits sharing of source text since a program can be partitioned into units that are separately compilable. The source text for a unit can be used in several programs by copying that text into the corresponding program libraries or (preferably) by permitting indirect references in a *program library* to units in other libraries. This mode shares the algorithmic content of software but not the expense of compilation.

2. Object modules -- Conventional programming systems share software through libraries of object modules. An object module is produced by the code generation phase of a compiler and contains the machine instructions into which source text is translated. References to external items are symbolic and not resolved until the object modules for a program are bound to form a memory image. Object code can be shared in the Ada environment if program libraries support references to external units.

3. Memory image -- The *object modules* for a program are bound together to remove symbolic inter-module references to form a memory image that can be loaded as an entity. For an image (containing reentrant code) to be shared among users, constraints must be placed on the relocatable quantities it contains. It must

be possible for each address or relative displacement to have the same value in each logical address space in which the image is used. For most architectures, an image can be shared only if it is a complete program or is loaded at a dedicated location in each user's address space. The former case is generally acceptable since a program can be replaced by a new version without affecting the user's interface. The latter case leads to system-wide configuration problems: a change to a piece of shared code causes obsolescence of all bound programs that reference it (unless references are indirect through some type of system service vector that decouples a service from the address of the code that provides it, in which case there must be system-wide agreement on allocation of entries in this vector.)

4. Constrained position-independent image -- Sharing can be made less difficult if inter-module references are made through a program-local table that contains the address of each module. If each module contains position-independent code and all external references are made via an index into this table, a module need not be loaded at the same logical address in all programs that share it. The constaint on sharing is that each shared module have the same index. (This is functionally equivalent to having a system service vector except a module need not be loaded at the same logical address in all programs.)

5. Unconstrained position-independent image -- The technique used in the Ada Software Environment supports unrestricted sharing of modules in memory image form. A collection of object modules can be bound into a memory image called a segment. Each segment has an associated table through which all external references are made. Since each table is segment-local, a module need not be referenced with the same index in all segments. Sharing of a segment requires only that a program-local of its external reference table be built and that any external references to modules in the segment be resolved.

The form of segmentation that is implemented in the Ada Software Environment is a generalization of that used by the Hewlett-Packard HP3000 Series II computer ([HP 77]). The program binder supports the grouping of a collection of object modules into a segment that contains four components: constant section dictionary, constant section, code section dictionary, and code section. The dictionaries provide a uniform method for access to both internal and external objects. An internal entry is represented by its displacement from the corresponding section. An external entry is represented by the number of the segment in which it resides and its index into that segment's dictionary. Each program has a segment table that contains the addresses of the four components of each segment. The constant section contains (read-only) constants that are referenced in each subprogram in the segment; a constant that is declared in subprogram can be accessed from a nested subprogram. The code section contains position-independent (reentrant) code for for each subprogram.

2.1.2 Shared Host System Segments

The inter-segment reference tables described above permit segments to be shared without restriction on a host system. The same segment can be loaded at different logical addresses in different programs.

2.1.3 Dynamic Binding

Each segment contains a symbolic form of its code section dictionary that identifies the internal subprograms that are defined in the segment and the external subprograms that are referenced. With this information, a loader can construct the inter-segment reference tables for a program at execution time. This capability can also be used to link a partially bound program to resident segments.

On a host system, dynamic binding can be performed as the disk-resident versions of segments are brought into memory during initial program load. On an embedded system, the analogue of this type of binding is construction of inter-segment reference tables at system power-up. It is feasible for the embedded system to contain an initialization routine that builds the tables based on a list of addresses at which segments may occur, typically in read-only memory (ROM). Usage of this technique means a new version of a software component can be installed by inserting a new ROM and the ROMs that contain references to this component are not made obsolete.

A variation of power-up binding is patching defective software that is in ROM. By controlling the order in which segments are examined during binding, it is possible to have the dynamic binder replace one subprogram of a segment by changing its internal entry in that segments code section dictionary into an external reference to a segment that contains the corrected version of the subprogram. The host system analogue of this capability is testing a new version of a subprogram that is in a shared system segment.

Another variation of power-up binding is to use dynamic binding to down-load partial programs into embedded systems.

2.1.4 Address Space Multiplexing

Segmentation provides the interface that makes possible execution of a program in physical address space that is not large enough to hold all of the code for that program. For this to be practical, it is essential that an object module be partitioned into constant and code sections since the data in this sections have different access characteristics. In particular, Ada permits a structured constant (e.g., a string whose value is known at compile-time) to be passed by reference from the subprogram in which it is declared to a series of unrelated subprograms. Thus, there is not necessarily the locality of references for constants that there is for code.

On a host system, address multiplexing corresponds to overlaying and demand loading of segments. In both cases, the segment reference tables are augmented to include information about the disk and memory addresses at which an overlay or segment resides. When overlaying is employed, the program is bound with the relative positions of overlays specified. With demand segmentation, the ASE manages a region of memory into which segments are loaded as they are referenced. The linkage handler can examine the segment tables to determine when disk transfers must be made. Since all inter-subprogram transfers are made through the handler, there are no restrictions on recursive calls or calls between subprograms in parallel overlays. (Inter-overlay reference data can be used to automatically promote constant sections to ensure constants are resident whenever they might be referenced.)

On an embedded system, address multiplexing corresonds to automatic manipulation of memory mapping registers or ROM enablement lines. For example, the 1750A supports a 16-bit logical address space and a 20-bit physical address space. There are dual logical address space for both instructions and data; for each type of access, there are sixteen mapping register that map 4096-word blocks of logical addresses into physical address space. A special version of the ASE linkage handle can be used to adjust mapping registers at subprogram call and return in such a manner that all instructions for a program are accessed through one or more 4096-word windows in logical address space. On simpler architectures that do not have mapping register, the linkage handler can manipulate ROM enablement lines to select code segments.

2.1.5 Debugging

The ASE linkage handler provides a convenient point to probe programs for dynamic debugging at the subprogram level. Since all calls are made through the handler, the segment tables can be extended to include flags that indicate if a given subprogram should be traced or have performance data gathered.

2.1.6 Optimized Linkages

The overhead to provide the features listed above is generally the expense of one level of indirection of intra-segment calls and two levels for inter-segment calls. Compared to the average number of instructions that are executed per subprogram, this is not excessive. For time-critical application, it is possible to use binder options and/or different versions of the linkage handler to return to more conventional linkages. One optimization is to use self-relative displacements for intra-segment reference to avoid access to the linkage tables. If a target machine has a 16-bit logical address space, logical addresses can be used instead of a table index as the parameter of a call; if the address space is larger, the parameter can be the index into a single table of addresses.

2.2 Parameter Passing

Compilers for Algol-like languages usually pass parameters by "pushing" values on the same stack that is used for storage allocation; an actual parameter becomes an initialized object in the callee's stack frame. This approach is particularly appealing if the instruction set architecture has addressing modes that support auto-incrementing and/or auto-decrementing of addresses held in registers. For machines such as the Perkin-Elmer 8/32, IBM 370, and 1750A that do not have stack instructions, the stack model for parameter passing is not more space-efficient than passing the address of a parameter list that the caller builds in its stack frame. In the context of the Ada optimizing compiler, use of a parameter list is advantageous:

1. If no attempt is made to optimize parameter passing, the two approaches are essentially equivalent. The same amount of stack space is required since a single parameter list can be used that contains maximum storage required for any of the calls in the subprogram. The code size is essentially the same since each parameter must be passed using a base/displacement addressing mode.

2. The parameter list is a structure in the stack frame of the caller that is subject to the full power of the compiler's optimization phase:

   a. If all parameters of a call are IN and are known at compilation time, the parameter list can be allocated in the constant section and passed by loading its address in the parameter list register.

   b. If some of the parameters of a subprogram are invariant inside a compound statement, the initialization of those parameters can be moved out of the statement. In particular, a parameter list can often be partially constructed before a looping construct is entered.

3. Use of parameter list provides the potential for dynamic expansion of the stack region:

   a. Since no reference is made to the callee's stack frame before the call occurs, no data need be relocated if the linkage handler must acquire more stack space from the storage manager.

   b. Since the caller's registers are saved in its stack frame, the linkage handler has free registers with which to acquire memory.

2.3 Input / Output Configurability

The input/output component of the Ada Software Environment uses the subprogram SET_NAME to establish a correspondence between an internal file and the external file or device that it represents. Since the name is a string that can be calculated during program execution, some technique must be found to connect a file dynamically to the device handler that performs its input/output. It is desirable that new devices be added in a modular manner: it should not be necessary to replace existing load modules or to recompile an I/O dispatch routine that uses a large CASE statement to select device service routines.

I/O configurability is provided in ASE through packages called virtual devices. The visible part of the specification of each virtual device package is the same: it contains declarations for subprograms that provide the standard services of the KAPSE virtual I/O interface. The body part of each package contains the implementation of these services. A file name is processed by presenting it to each virtual device package. If the file name is recognized, the virtual device package returns a service vector containing the address of each subprogram that provides a KAPSE virtual I/O service. A user requests device-independent services from a virtual device through interface subprograms that use the device service vector to make a parameterized transfer to the proper subprogram.

2.4 Inter-Program Communication

The Ada Software Environment supports inter-program communication through a virtual device that uses host system message passing primitives to transfer data. This capability has been used in other systems (e.g., Unix ([UNIX78A]) to build complex processors from component programs by using the output of one program as the input of the next. When inter-program communication used to simulate a virtual terminal, a very useful form of software reusability results. For example, the functions of the library utility can be provided to a program through a package that dynamically invokes the library utility program. Service requests are made through calls to subprograms in the package and are converted into simulated terminal inputs that drive the utility.

2.5 Foreground/Background Execution

Many current operating systems (e.g., Unix) distinguish between foreground and background service. A user typically executes non-interactive programs in the background while an interactive task (e.g., editing) is performed concurrently in the foreground.

The Ada Software Environment provides the user the a more general capability to control dynamically which programs are connected to his terminal. Having multiple windows per terminal permits several programs to be monitored concurrently. If a program takes longer to execute than had been estimated initially, it can be disconnected from the terminal and resumed later without loss of any data.

2.6 Structure of Executive Program

The executive program has been designed to make extensive use of the tasking features of Ada. The programs under its control are represented by instances of the command language interpreter task within the executive program. Since these tasks share the same address space, the ensemble of programs associated with a user can be controlled through shared data structures. Asynchronous I/O interactions occur through interrupts.

2.7 VM/370 Implementation

Under VM/370, the Ada Software Environment will be implemented on what appears to be a bare machine. An operating system will be produced that supports multiple concurrent programs, each of which has its own virtual address space. All device interfaces must be supplied. Moreover, this software must be implemented in subsets so testing of other components of the Ada Integrated Environment can begin prior to completion of the ASE.

The following development strategy will be used to minimize the risk in providing the ASE:

1. The KAPSE virtual interface will be designed to be implementable under both OS/32 (on the Perkin-Elmer 8/32) and CMS (on the IBM 370). Under CMS, OS/370 compatible service calls ([IBM79L]) will be used (as much as possible) since they are typical of the services provided by host systems and their usage will make more likey the identification of a transportability interface. (Although the full AIE may not be supportable under OS/370, consideration of OS/370 when *designing* the ASE should make it possible to transport individual programs that do not require the full multi-user, multi-program capability of the AIE.)

2. The first phase of developement will provide execution of a single program containing a single task. This level of support corresponds to the environment of the bootstrap compiler; it is sufficient to test algorithms that do not involve concurrent execution. The primary features to be verified are the Ada execution environment and the high-level I/O interface. Since the Ada optimizing compiler, database, and program binder will not be available when this testing is performed, host system object formats and link editors will be used. At the conclusion of this phase, an adequate environment will be available for the initial development and testing of other components of the Ada Integrated Environment.

3. The second phase of development will support execution of a

single program with multiple tasks. The ASE will be upgraded to include task management, the component that supports simulated concurrent execution within a single program. This component involves complex, time-dependent algorithms that must be tested extensively.

4. The third phase of development will support the execution of multiple programs in a single virtual address space. This phase corresponds to the addition of the program management component, whose implementation depends on the task management component that was verified in the previous phase. The primary features that must be tested are:

   a. dynamic program invocation

   b. executive program:

      * terminal device controllers

      * virtual terminal interface

      * executive command language

      * command language interpreter

      * inter-program communication. At the conclusion of this phase, an adequate environment will be available for the initial integration testing of the components of the Ada Integrated Environment. The programs associated with a user are executing under CMS in a single address space (i.e., no virtual memory management is provided by the ASE); concurrent execution of programs is supported through time-slicing.

5. The last phase of development will support the execution of multiple programs, each having its own virtual address space. The program management component will be extended to become a virtual memory operating system that supports concurrent execution of multiple programs within a user's virtual machine. Device handlers will be provided that are similar to those in CMS. The design of this system will be coordinated with the hardware "assists" that are provided in the IBM 370 firmware.

2.8 Virtual Machines

The VM/370 version of the Ada Software Environment uses a distinct virtual machine for each user. An alternative approach is to use a single virtual machine under which all users execute. A design based on multiple virtual machines has the following advantages:

1. The system is more secure if all interactions among users are restricted to inter-machine communication primitives.

2. The system is more robust since damage can be isolated within the environment of a single user.

3. Resource management management by the ASE is less critical:

   a. If most users at an installation are using the AIE, the utilization of the host processor will depend primarily on how well the ASE performs. The ASE would require more complex algorithms and fine-tuning than if the VM/370 control program schedules the processor and its resources. Scheduling by the control program is desirable since it has been optimized through firmware ([MAC79]).

   b. If a single virtual machine is used, its virtual address space will not be large enough to hold many users unless extensive paging is performed by the ASE. Giving each user a virtual machine has the effect of multiplying the logical address space of the 370 since paging must be performed by the ASE only if the storage of the virtual machine is exceeded by the requirements of a single user, not all users. If each user is allocated a large virtual machine, the ASE can be optimized to permit *concurrent* execution of programs that are memory resident with respect to the user's virtual machine. The memory for these programs is actually paged, but by the control program using firmware assists.

SECTION 3

COMMAND LANGUAGE

3.1 Command Language Design Goals

The Command Language is the principal user interface to the Ada Integrated Environment. It should facilitate all user activity while providing convenient, programmable ways for managers- to control access to all processing and data resources of the system. Some specific objectives of the command language design for the Ada Integrated Environment are:

1. Provide a user friendly environment by including:

   * Common language for interactive and batch users
   * Easy to remember and consistent names
   * Explanatory error messages and prompts
   * On line help facilities with examples

2. Accommodate varying user characteristics by including:

   * Beginner mode: prompting, menus, simplicity, defaults
   * Advanced mode: concise; access to advanced features
   * Customized working environment for each user
   * User-definable commands and command libraries

3. Implement programmable capabilities by including:

   * Capabilities to declare variables and constants
   * Conditional and iterative statements; blocks; procedures
   * Arithmetic and Boolean expressions and operations
   * Program invocation and parameter association
   * Access to attributes of database objects
   * Convenient definition and manipulation of strings

3.2 Command Language Functions

User activities at the command language level generally include the invocation of stored programs, manipulation of database objects, and generation of reports showing the status of the system, the database, or the programs.

3.3 Command Language Programmability

The software development model for the Ada Integrated Environment points out that production of any computer program is a sequence of integrating and transforming steps, potentially involving a large number of database objects and software tools. At each step, the user must identify the inputs, processing controls, and outputs of the program to be invoked. After each program has been run, the user must examine available information to determine success or failure before proceeding to the next step. As the complexity of a processing sequence increases, the potential for error increases as well; errors are especially prevalent if each processing step requires extensive user interaction.

If a programmable command language is available, users may construct command procedures of substantial complexity to automate the routine activity between steps of the software construction sequence. An appropriate command procedure may assist the user by determining the parameters to be passed to each program, checking attributes of database objects for validity, and taking appropriate action when errors occur.

3.4 Command Language Convenience

While programmability is essential, the Command Language user is also concerned about convenience of use and efficiency. The language should be expressive and promote readability without being too wordy. Syntax and semantics should be clear and as simple as possible. Prompting and assistance should be available automatically to the inexperienced user, and command language procedures should be self-documenting, but experienced users should be able to enter commands in very concise, even cryptic, ways if they wish. Every user should be able to construct and maintain a personalized working environment in which commands may be renamed, default database directories may be specified, and operating conditions may be "remembered" between interactions with the system.

3.5 An Ada-like Command Language

A command language conforming as closely as possible to the syntax and semantics of a subset of Ada has been selected for this system. The requirements for programmability are well satisfied, and a systemwide environment is established in which the casual user invokes stored Ada programs and command language procedures identically. Attributes are defined in the language to facilitate the handling of database objects. Strings are provided as a predefined data type to facilitate the handling of literal names. A renaming declaration allows users to specify shorthand names and default parameters for commands.

SECTION 4

ADA DATABASE SUBSYSTEM

4.1 Database Design Goals

Modern software engineering techniques emphasize program modularity, structured design and development, extensive testing, and software maintainability. Software management is concerned with documentation, configuration management, and change control.

The Ada language recognizes these important concerns by providing features to support separate compilation of program units, so that programs may be designed, written and tested in largely independent parts. These features are especially useful for large programs and for the creation of libraries.

Implementation of an integrated Ada software support system therefore requires special emphasis on highly automated, flexible, efficient, programmable capabilities for the storage and retrieval of information concerning the components of a software product, with facilities to use this information to control all processing in a software development and maintenance environment.

4.1.1 General Requirements

Some of the facilities that an Ada database management system must provide are:

* Storage resource management
* Data security and user interfaces
* Data change control and tracking
* Data backup and archiving
* Data configuration management.

This section will discuss the Ada language and environmental requirements for a database subsystem propose a structure to satisfy those requirements, and discuss the operation of Ada support software in an environment based on the proposed structure.

4.1.2 Language Requirements

The Ada language [DoD80B] offers support for separate compilatio n of program units in a way that facilitates program development and maintenance. An Ada program is a sequence of program units; these program units may be subprograms (which define executable algorithms), packages (which define collections of entities), or tasks (which define concurrent computations).

Subprograms and packages may each be further separated into specifications and bodies, which may be compiled separately. Ada provides means to specify or restrict the interdependence between program units, through visbility rules.

Separate compilation of program units is considered a practical necessity to divide large programs into simpler, more manageable parts and to provide a library facility. On the other hand, many of the modern concepts of strong typing, explicit declaration of all identifiers, nesting of program units and scope of declarations have significantly complicated the partitioning of programs into separately managed units. The introduction of generic program units and overloading in Ada will provide additional challenges to the program librarian, as will the compiler features (pragmas) INCLUDE and INLINE.

Other modern language processing practices also require complex library management procedures by introducing a variety of source-language, intermediate,and machine-level representations of program units and associated data structures to support application libraries, machine-independent optimization, object-level program binding, source-level debugging, and retargeting.

4.1.3 Environment Requirements

The STONEMAN requirements in support of Ada [DoD80A] specify a software support environment in which the central feature is a database, the repository for all information associated with a software project for the entire project life cycle.

The Ada database must be a flexible, random-access structure in which the user defines objects that correspond to the systems, subsystems and program units of the software configuration(s) to be managed. Some elements of this data structure will contain information describing the location and physical characteristics of program units, data, documentation and other database objects, as a file management system usually does. Other elements must contain information related to security, configuration management, version identification, development history and status.

The user may wish to specify access controls such as passwords; schedule and status information such as creation dates, test completion dates and versions; programming language and usage codes such as "source", "object", or "data", or other Boolean, numeric, or string-valued attributes as needed.

The user may define different sets of attributes for different collections of objects within the database. It will be necessary to control user access to or modification of certain attributes, to preserve database integrity.

Any object in the database may also possess attributes or contain user-defined information which relates it to one or more other objects. Examples of such relations are

* The set of all versions of a particular program unit (the "version group" of STONEMAN);

* The set of Ada program units that forms a program (the "program library" of the Ada reference manual);

* The set of database objects that are the responsibility of a particular programmer or project;

* The set defined by the specifications, source code, intermediate code, symbol tables, listings, trouble reports, change records, test data and documentation of one component of a software product (to satisfy DoD configuration management requirements, such as those of MIL-STD-1679 [DoD78A]).

An integrated Ada database management system must provide both batch and interactive interfaces through the system command language. It must also be a component of the low-level interface through which running user programs and software tools create, access and modify database objects. It must provide controlled mechanisms by which users at all levels may define, specify and modify the attributes of and relations between database objects.

4.1.4 Previous Work

Most large scale operating systems provide sophisticated mass storage management and file management routines, but they are seldom integrated with user programmable management systems. Several recent efforts [KP76, IV77] provide integrated collections of software tools, facilitating development; some packages are able to track multiple versions of programs [RO75] or facilitate separate compilation of Pascal program units [TI79A]. STONEMAN and other Defense Department specifications [RADC74E] prescribe a "programming support library" as an essential element of a software development system. Implementation of a generalized concept of file attributes, library management, and high-order job control language has been achieved by at least one major computer manufacturer [BU77A, BU77B].

4.2 Organization of an Ada Database Subsystem

The following sections propose an organization and structure for an Ada database. This organization is based on a highly structured directory that stores user- and system-defined information concerning all objects in the database. Later sections describe access to the database through the directory and give some examples of its use by compilers, command language, and general user programs.

4.2.1 Logical Organization

There are three general classes of Ada database objects: files, directories, and dictionaries, distinguished as follows:

* Every database object is a file. A file has a unique name, it has attributes (defined below), and it contains information.

* A directory is a special kind of file in the Ada database. It is a form of relational database whose elements contain information pertaining to files. There is one and only one directory element for each file in the Ada database. A directory element contains the attributes and relations which describe a file.

* A dictionary is a special kind of file in the Ada database. It contains the definitions of attributes and relations specified for a particular set of files. Each Ada directory is linked to a specific dictionary.

* Any file that is neither a directory or a dictionary is called a "data file" in this discussion, regardless of its contents.

Each file is assigned a permanent, unique database name at the time of its creation. This name -- essentially a registration number in the context of the host system -- may be used by database management and other routines for convenient identification.

Attributes consist of data that concerns only one file. Each attribute has a name and a value; allowable values are Boolean, numeric, enumeration or string literals.

Conceptually, relations are labeled arcs that connect any two files. Each relation has a unique name. Any file may be connected to one or more others by the same relation. The value of a relation may be visualized as a list of file names. All relations are bidirectional; inverse relations are automatically maintained, and relations may be followed in either forward or inverse directions.

The logical structure of an Ada database is that of a tree of files. In turn, the logical structure of a directory is that of a tree of elements corresponding to some subtree of the database. The pathname of a file or directory element corresponds to the special attribute NAME and is derived rom its position in this hierarchical structure.

A special relation called CHILD may be used to traverse the tree of descendants of any directory element. Each element has a single parent, which may be accessed by following the relation PARENT (or INVERSE_CHILD). The number of descendants of an element is limited only by the amount of physical storage available for the database.

4.2.2 Physical Organization

An Ada database may reside on a combination of media. Immediate access mass storage, such as disk, is appropriate for active files and directories. Off-line bulk storage media, such as tape, may be more appropriate for long-term archiving, backup, and storage of early versions of active files. A large database may occupy several removable disk packs. In distributed processing systems, several independent processors may require concurrent access to shared mass storage.

Efficiency is a major consideration in the design of the database, especially in the design of directories. Access to directories should be simple and reliable; updates must be easily and quickly accomplished; the overhead associated with directory searches must be minimized; and directories must be protected against inadvertent damage.

The directories suggested here have many of the well-known properties of hierarchical trees, but also contain linkages best implemented in a general graph structure. In addition, efficiency of storage and updates must be balanced against needs for open-endedness; any element in the tree structure may have an arbitrary number of descendants. The directory structure must also conform to physical constraints imposed by the storage medium, here assumed to be disk.

One structure that meets all of the above requirements reasonably well is a random access file, logically organized as a tree of B-trees [Wi 76]. (A B-tree is a multiway tree structure subdivided so that subtrees are stored as units, called pages, to reduce the number of disk accesses. This organization provides efficient storage utilization, simple algorithms for search, insertion and deletion, and some other useful properties.) Each physical record of a directory file is one page of a B-tree and contains one or more nodes.

Two types of pages are found in the directory. Directory elements, constructed mostly of pointers, make up directory pages; attributes and relations, constructed mostly of literal text, make up attribute pages.

Each directory element has a name, a pointer to a B-tree of children, and a pointer to a B-tree of attributes and relationships.

A B-tree may contain many pages, depending on the number of nodes (elements or attribute data) it contains. One distinguished page, called the root page, is the first allocated for a new B-tree and is released only when the entire B-tree is deleted. It is thus possible to use a root page number as a permanent pointer to a B-tree, even though B-trees may be re-balanced and all other pages allocated or deallocated dynamically.

Each directory element has a "given" name, which is a single identifier, and a "full" name which is constructed by concatenating the given names of its ancestors in the tree structure. Each directory element may also be located by its unique "element address," which consists of the root page number of its B-tree and its given name.

4.3 Operation of the Ada Database Subsystem

This section discusses the implementation of access to the Ada database by an operating system, user programs, and command language. Features necessary to implement an Ada compiler are given special attention.

4.3.1 Creation and Deletion of Database Objects

The database subsystem must be capable of creating temporary files as needed by running programs, and of making permanent directory entries for any of these files that Apse tools or users wish to save.

* Each file must be assigned a unique database name upon creation. Its directory record is constructed and entered, and a hierarchical pathname is assigned when the file is saved.

* Provisions must be made to handle the situation that arises when a program attempts to save a file with a pathname that already exists in a directory, thus attempting to replace one or more existing files. This may be permitted for some file categories but denied for others.

Any node in the directory may be given a new pathname. If a node is renamed, all of its descendants are also. Database names, however, are permanent.

* If a node is given a pathname that already exists in the directory, it implies replacement of one or more existing files. This may be permitted for some file categories but denied for others.

* Relations between files should be unaffected by renaming. This may be accomplished by using database names, rather than pathnames, to describe relations.

* Correspondence between database names, host filesystem names and the hierarchical pathnames of database objects must be maintained in a database map. The contents of database objects may be copied to other storage media or removed from online storage.

* Appropriate attributes and relations should be copied with the contants of database objects to archival storage.

* Complete deletion of a database object may not be permissible if it is related to any other objects. On the other hand, deletion may be permissible if the object has no relations.

4.3.2 Access Control

Access to files in the Ada database should be restricted only by the management policies of the user community. Directories and dictionaries, however, define the structure of the database and provide the mapping of database objects into physical storage. User access to directories should be accomplished only through a protected interface.

The following primitive operations must be implemented:

* On dictionaries: Creation and initialization, definition of attributes and relations, and lookup of the definitions of attributes and relations.

* On directories: Creation and initialization, creation of elements, lookup of elements, making and breaking relations between elements, assigning values to attributes, retrival of attribute values and relations, renaming and deletion of elements.

* On files: Creation, location, storage management, access control, data transfer, copying, archiving, restoration, renaming and deletion.

In addition, directories must be protected against damage from concurrent attempts to update by independent processors in distributed processing systems. These synchronization mechanisms must provide locks on individual records in directory files.

File security in the traditional sense is implemented by directories containing the "access privilege," other attributes of users and relations between users, by file attributes or relations which describe access constraints, and by routines that compare user attributes against file attributes whenever access to a file is attempted.

Access control may also be applied by use of "usage" attributes so that, for example, an Ada compiler will reject any input file whose "usage" attribute is not "Ada source text."

These primitive routines will be implemented as components of the kernel Ada environment. System and user programs interact with the database through file attributes and relations. Host operating system facilities will be used where appropriate.

4.3.3 Ada Program Interfaces

Ada programs have facilities to declare files as program variables. Programs should also be able to specify the attributes of files in their declarations, to specify the record structure associated with a file, to determine the value of any file attribute during execution, and to modify the values of appropriate attributes during execution.

An Ada implementation should provide a minimum set of operations for association with an external file (OPEN, CLOSE) and for processing or positioning (READ, WRITE, SPACE). All other functions may be handled as interrogation or modification of the appropriate attributes. For example, given file F:

```
F'NAME         -- returns the external name as a string
F'SIZE         -- returns the number of records in the file
F'OPEN         -- returns TRUE if an external file is associated
F'KIND         -- returns a value representing the external device
F'ACCESS       -- returns IN, OUT, or INOUT as appropriate.
F'CATEGORY     -- returns a string describing the intended use
```

Similarly, attributes may be modified when appropriate:

```
F'NAME := 'ALPHA.BRAVO'  -- assigns the external name
F'KIND := DISK           -- assigns this file to a disk device
F'NEXT := 1              -- rewinds the file.
```

Relations may be handled like attributes in many ways, but the value of a relation is a list in the general case. User-level facilities must be provided to establish (make) a relationship between two files, and to break such a relationship. A convenient data structure and operations must be provided to work with the sets of names of files obtained through relations. (For example, FCHILD refers to all of the immediate descendants of the file F.)

Since attribute identifiers are user-defined in dictionaries in the database, it will be necessary for the Ada compiler to generate a subprogram call, probably to an operating system routine, passing the literal attribute identifier and the file descriptor as parameters. This mechanism provides security and flexibility, but does introduce a modest execution-time overhead.

4.3.4 Command Language Interfaces

The command language for an Ada environment should allow the user to customize the environment to suit his application. It should be possible to develop command language programs to automate routine chores and to enforce the configuration management and quality control policies of a project. A well designed command language should allow programming of arbitrarily complex operations on database objects while maintaining a consistent, clean user interface.

A single "command" (in any command language) typically initiates execution of a stored program, such as a compiler, and passes parameters to it. In most systems, parameters are passed as literal strings of characters which are then interpreted either by the operating system or by the invoked program. Very frequently, these literal parameters are the names of files. Frequently executed or simple commands, such as deletion of a file, may be carried out by routines embedded in the command interpreter; the mechanism is essentially the same as if a stored program had been invoked, but some overhead may be saved.

Many command languages permit specification of some of the attributes of the files passed as parameters to programs, but only a few provide facilities for direct, general interrogation or modification of file attributes or relations between files in a database. (Burroughs B7000/B6000 "Workflow Language"[BU77A] is one example.)

The command language of an Ada environment should provide controlled facilities for access to the database, with simple constructs to interrogate or alter file attributes and relations. It should have facilities for conditional execution of stored programs, for the dynamic construction of parameters and examination of program results, and for a limited repertoire of arithmetic, Boolean and string operations.

4.3.5 Compiler interfaces

Several Ada language features interact strongly with the library management system. Of these, the most important is separate compilation. Some of the requirements that separate compilation places on the library system are:

* Naming conventions that unambiguously identify the files corresponding to the compilation units of a program library, or mechanisms to construct file names from program unit names as they are needed;

* Attributes that identify the various representations of a program unit (source text, intermediate text, object code, and other forms) and files associated with them (listings, symbol tables, flow graphs and others);

* Attributes (or related files) that store development and change history for each program unit, including compilation dates and times, and attributes to identify the various versions and revisions of a unit;

* Relations that associate all of the various representations of a program unit, including specifications and other documentation;

* Relations that show the interdependencies between program units, such as the association between module specifications and module bodies, or links between units that use INCLUDE pragmas and the files INCLUDEd.

In the process of determining a compilation context, checking the validity of a compilation context, enforcing the proper order of compilation, reading symbol tables and other texts, and producing output files, a compiler must interact with the Ada database in a variety of ways. Some of the general functions that must be accomplished by the compiler are:

* Construct file names from the identifiers found in with clauses and INCLUDE pragmas, and dynamically associate internal files of the proper types with external files named by the constructed

names;

* Examine the appropriate attributes of files to determine the validity of a compilation context;

* Construct the derivation record of a each output file created, using the input file name(s), control file name, and compiler identification;

* Assign attribute values (such as "usage," "compiler_name," "compiled_date," and "compiled_time") to output files as they are completed;

* Create or update development-history files, as necessary, for output files as they are completed;

* Establish relations between input files and output files as appropriate;

* Deal with errors as they occur.

4.4 Conclusions and Summary

The directory, dictionary and Ada-like command language concepts described have been implemented and used at Texas Instruments for approximately four years in software development facilities supporting projects in Pascal and Fortran.

The command language appears to be easy (for programmers) to learn and use, and more than adequate for the expression of database manipulation algorithms. The string handling features of this language have been essential for the dynamic construction of file names and are unexpectedly useful for other purposes, such as report generation.

Database performance is a critical issue. There should be as few constraints as possible on the size of data items, or the number of entries in a directory, or the number of operations performed during a single database access. The hierarchical organization of the database leads naturally to the use of recursive procedures in command language programs; the implementation of the command language must provide sufficient memory to accommodate deep recursions.

Database integrity is also an important issue, since most modern software development facilities are multi-user (and frequently multiprocessor) environments. Facilities must be provided to allow multiple access to directories and to allow various users to update different parts of a directory simultaneously, without conflict. Record-level locking appears to be a satisfactory solution to this problem, but in system configurations where multiple processors communicate only through shared disk, the locks must also be written to disk and problems may arise if a processor fails during an update.

The interfaces between user programs, including compilers, and the database should be simple, familiar, and convenient. To the extent that it is reasonable,the physical details of storage media should not concern the user. In particular, storage allocation, backup and restoration of files should be automatic. On the other hand, the user should have the power to configure an environment best suited to his own application and methodology, an argument for flexibility and extensibility.

The general concept of file attributes has been used in several successful large systems. The extension of this notion to describe relations between files appears natural in an environment where many complex, overlapping interdependencies may occur. A few programming language extensions may be required to describe relations in a convenient way.

The database is generally acknowledged to be the central feature of a software development facility; its management is crucial to the success of a ,arge software development project. Thoughtful design of an Ada database subsystem is crucial to the success of the language.

SECTION 5

ADA LANGUAGE PROCESSORS

5.1 Compiler Design Goals

A major compiler design goal was to produce a high-quality production Ada compiler, i.e., a compiler that qenerates object code comparable to hand produced code. Therefore, the design incorporates the latest optimizing compiler technology which has been applied and proven. The philosophy behind this approach is that such an optimizing compiler, meeting the performance requirements in the SOW, can be cost effective for the development and maintenance of software for embedded computer systems. Size of object code is a major consideration for these systems; optimizations must be applied to attain compact efficient code.

5.2 Compiler Structure

The structure of the Ada optimizing compiler is mainly based on the compiler technology and the intermediate language used. Using as a baseline the compiler technology developed by Texas Instruments for its Pascal compilers (TI Pascal and Microprocessor Pascal) and Fortran compiler (on its dataflow processor and Advanced Scientific Computer), the most current, proven compiler technology was injected, viz., the technology used by the Bliss-11 optimizing compiler [WUL75], the technology used by the target compiler, or PQC, produced by the PQCC system [WUL79, WUL80A, WUL80B], and the technology used by the University of Karlsruhe in their compiler front end for the German MoD [GOO80]. There are a number of reasons for this approach.

* Texas Instruments Pascal compiler technology is applicable since Ada is Pascal based. Texas Instruments experience in using a recursive descent parser in its Pascal compilers will be utilized.

* Texas Instruments Microprocessor Pascal language contains a process construct and semaphores, and the experience in handling these constructs will be applied to compiling Ada tasks and the rendezvous constructs. The experience in constructing the Pascal run-time environment will be drawn upon to determine the Ada run-time environment and therefore the proper code to generate and the necessary processing to be performed.

* Texas Instruments extensive experience with machine independent optimization in its Fortran compiler for the ASC and dataflow processors will be applied appropriately to the optimization pass of the Ada compiler.

* The main goal is the generation of high-quality code. To achieve this end requires examining a wider global context of constructs instead of narrow local contexts. The former approach is exemplified by the latest compiler technology, the later by older ad hoc techniques. Therefore, the PQCC technology, which is being used to generate an Ada compiler, is applicable. Bliss-11 is also applicable; it is one of the better optimizing compilers and generates excellent high-quality code.

* The crux to the speed of the compiler are the algorithms employed, not the rewriting of routines in machine language (a rewrite results typically in a 10-20% increase in throughput while use of a better algorithm can result in a factor of 2-5 increase). The algorithms (for code optimization, code generation, tree walks, etc.) used in Bliss-11 and PQCC are the result of vast experience and can be adapted to handle Ada. They are tree-oriented, i.e., designed for IL's represented as trees such as TCOL [BRO80A], AIDA [DAU80E], or DIANA [GOO81]. One can draw on the practical experience obtained from using the algorithms thereby reducing development time (i.e., reinventing the wheel, making non-optimal design decisions, etc.).

* Parts of the PQCC system still in the research area (e.g., automatic generation of tables from a machine description for input to a machine-independent code generator) can be adopted when the technology is developed.

* The goal of the PQCC research effort to construct machine-relative compilers [WUL80B], i.e., compilers parameterized by the characteristics of the target machine, is another desirable goal since it can improve the quality and reduce the cost of retargeting the compiler.

* The Ada-0 approach is based on the formal definition of Ada. This formal approach lends credulity to the belief that the resultant compiler is both correct and complete, moreso than for an ad hoc approach. Correctness and completeness are important characteristics with respect to passing the compiler validation test [SOFT80B].

5.2.1 Description of the compiler

The Ada optimizing compiler consists of two parts: a front end (analyzer pass) and a back end (expander/optimizer and code generator pass). The front end, back end dichotomy is dictated by the definition of DIANA [GOO81], the intermediate language used between passes. DIANA defines an intermediate representation of an Ada program for a particular point in the compilation sequence. As such it assumes a particular model of the compiler. In particular, certain processing is not performed by the front end resulting in a high-level representation of the Ada source test. For example, expansion of generic instantiations, processing of pragmas (e.g.

INLINE), expansion of array subscript calculations into address calculations, etc. must be performed in the back end. For this reason, machine independent optimization is preceeded by a phase which transforms (expands) the high-level IL representation into a lower level representation that embodies decisions based on the target machine.

5.2.2 Organization of the Code Generator

The code generator is table driven [CAT79, GRA80]. The advantages of this organization are

* it results in a code generator that is machine-relative [WUL80], i.e., one that is parameterized by the characteristics of the target computer, thereby simplifying the task of retargeting the code generator for another machine; and

* knowledge about the target computer is encoded in tables as pattern-action pairs and not scattered throughout the compiler code thereby making maintenance, modification and debugging easier.

The code generator is thus simplified to a machine-independent algorithm for pattern matching and is more comprehensive and (possibly) faster than ones that do comparable analysis. The pattern-action pairs will be generated by hand for each target machine. When the technology is developed [LEV79, LEV80, WUL80] to automatically generate them from a formal description of the target machine, it can be used and greatly facilitate retargeting the compiler to other machines.

5.3 Selection of a Recursive Descent Parser

The reasonable alternative parsing strategies are recursive descent and LALR(1). LALR(1) parsers tend to be smaller parsers and are extremely efficient. However, a substantial effort must be expended on the tool which processes the grammar and builds the syntax tables used by the parser. Even if such a tool were available, and there are several, it would have to be recoded in Ada to satisfy the contractual requirements.

Recursive descent has the advantages that it is well understood, straight-forward, and relatively simple to implement. In addition, if an effort is made to factor the grammar as much as possible, it is not much less efficient than table driven methods. Recursive descent parsers also lend themselves well to the development of ad hoc error recovery techniques, and are easy to modify and debug since the syntactic analysis and the semantic analysis are intermixed. The latter point should be emphasized because most table-driven parsers perform only syntactic analysis, therefore, they must be followed by semantic analysis. This means considerable duplication of effort between the two passes. Recursive descent also allows systematic construction of the parser according to a set of rules which map the syntax diagrams into a sequence of statements. This can lead to a more efficient and more easily managable system.

5.4 Selection of an Intermediate Representation

In selecting an intermediate representation for Ada programs, there are two forms to consider: linear and non-linear. Linear representations, such as Polish notation, triples, quadruples, and P-code, are essentially machine instructions for an underlying abstract machine. Linear representations have several drawbacks. Information needed for optimization or efficient code generation is not inherent in the representation, e.g., control constructs may be decomposed into more simpler primitive forms. In particular, for optimization more context is required than can be retained in a linear representaton. Moreover, mapping non-linear information, such as flow graphs, onto the program representation is both awkward and expensive.

A linear representation is attractive if a common intermediate language (IL) is to be used both *for interpretation and code optimization/generation.* This approach has been tried using P-code [SIT79, KOR80] where sufficient information for code optimization/generation is appended to a P-code instruction. This research has not demonstrated that efficient code can be generated for register machines. The design decision that the source level debugger not interpret the intermediate language, but execute the machine language directly alleviates the requirement of a dual purpose intermediate language.

Therefore, it was decided to choose a tree structured intermediate representation. It is more suitable for the back end of a compiler, i.e., for performing global optimizations and generating efficient code.

5.5 Selection of the Intermediate Language

The tree-oriented intermediate language selected by Texas Instruments is DIANA [GOO81]. There are a number of reasons for this design decision.

* First, DIANA is the progeny of two very similar tree-oriented intermediate languages designed for Ada: AIDA and TCOL-Ada. DIANA is better than either of its parents, for it benefits from extensive design experiences with compiling Ada to both previous forms.

* Second, Texas Instruments views the creation of DIANA as the first step in the process of standardizing on an intermediate language. A standard intermediate language will provide a uniform interface between different Ada compilers, tools, and programming environments and can lead to a standard front end.

* Third, DIANA is based on the formal definition of Ada [CII80]. Completeness and correctness of the intermediate representation of an Ada program is guaranteed.

* Fourth, Diana is designed for use by environment tools other than the front and back ends of a compiler.

* Fifth, it is possible to regenerate the source text from its DIANA representation.

* Finally, DIANA is extensible, i.e., different dialects can be created. This is necessary so attributes required by particular tools can be incorporated, in particular, the encoding of information needed by an optimizer or code generator. One tool, which derives a dialect for such purposes, IDL [NES81], already exists and can be utilized. Such a tool should be part of the standard.

5.6 Processing of Generics

Based on the definition of DIANA the front end of the compiler does not expand a generic instantiation or perform generic optimization. When to perform the instantiation expansion is a design issue. There are a number of factors to be considered.

* First, due to separate compilation and the fact a program library may contain a family of programs, the compilation units that constitute the program are not known until link time; input to the program binder is the name of the compilation unit in the program library that is the main program. Expansion of a generic instantiation cannot be delayed until link time for this would require that compilation of units that utilize the instantiation also be delayed.

* Second, a generic definition may not be sufficient to determine the form of an expansion, e.g., in the case of a generic definition whose formal parameters depend on the formal parameters of the generic definition in which it is embedded, or the case where it is necessary to know the size or frequency of a dependency relative to the total code. Deciding on the form requires knowledge of all instantiations and the size of the compiled code, repectively.

* Third, generic expansion is machine dependent and therefore must be performed by the back end.

The approach taken is to expand a generic instantiation at the IL level in the expander/optimizer. The expansion can then be processed by later parts of the compiler resulting in a customized expansion. Prior to linking, a package is called by the program binder that performs generic optimization. The package is given a list of instantiations for a generic definition. If generic optimization determines no code can be shared, the customized expansion is used. If code can be shared, a new expansion is generated. The binder then decides whether it is cost effective to replace the original expansions with the new. If it is, compiled code is retrofitted to use the new expansion in a manner that preserves the existing interface; this is managed by the program binder and performed at link time. For example, in the case of a generic subprogram, a new routine would be formed out of the object module for the new expansion. It would consist of multiple entry points, one for each instantiation. At each entry point additional parameters or case discriminates would be set followed by a branch to the common body.

5.7 Expansion of Inline Subprograms

The expansion of inline subprograms could be handled at the source level as an unnamed block. This would result in a stack frame containing the parameters and local variables. This approach is contrary to the philosophy behind the inline pragma, which was to eliminate as much as possible the overhead associated with a procedure call. Therefore, the inline expansion will take place as an integration of the IL of the callee with the IL of the caller at the point of call. This approach also has the advantage that name conflict, type resolution and parameter matching problems are handled more cleanly than at the source level.

5.7.1 Expansion of Inline Functions

There are a number of issues related to the expansion of inline functions. Functions which are components of an expression may be expanded in one of two places. Either the function is removed from the expression, and expanded and the resulting temporary variable is substituted for the removed function invocation, or the function is expanded in-place in the expression. Placing an inline function in front of a expression will involve the substitution of a 'load resultant temporary' instruction in place of the function invocation, finding the root node of the expression and inserting the inline expansion prior to the expression. This entails some complex rules for the substitution.

It has been postulated [SCH80] that the expansion of an inline function *would be impossible without some complex rewriting rules*. This is true for either expansion method. Inline functions could be expanded within the expression, treating the expanded code and resulting value as an algebraic parenthetical sub-expression. This expansion will make register allocation difficult. A set of registers may have to be stored and restored multiple times within an expression to handle the IL describing the program constructs, i.e., expressions, loops, etc., found within the function.

The controversy surrounds the fact that the removal of the function may alter the results of the expression. For instance, if a function causes the modification of a global variable which is found within the same expression as the function call, then the occurence of the inline function and its relative location to the global variable may change the semantics of the expression. The Ada Reference Manual (Section 4.5) states, 'The language does not define the order of evaluation of the two operands of an operator (excepting short circuit control forms). A program that relies on a specific order (for example because of mutual side effects) is therefore erroneous.' This is an implicit authority to expand an inline function in either manner. To keep the work to a minimum for the code generator, inline functions which are found within expressions will be removed to the front of the expression tree, expanded and the result placed into a compiler generated temporary variable.

5.8 Bootstrapping the Compiler

Texas Instruments plans to use an Ada front end developed by the University of Karlsruhe to construct a bootstrap compiler. A preliminary release is expected in April-May 1981, with the final release scheduled for October 1981. The Karlsruhe front end accepts full Ada [DoD80B], outputs DIANA, and implements the Ada separate compilation facility. It is a production quality program, written for the German Federal Ministry of Defense.

Use of a bootstrap compiler permits implementation of the AIE tools to proceed in parallel, especially those whose interface is DIANA, and allows all tools to be written in Ada from the outset.

The Karlsruhe front end was written in Ada-0, transformed to LIS, compiled by the Siemens 7760 LIS compiler, and linked with the LIS run-time package to produce an executable program which runs under the BS2000 operating system. The Siemens machine is essentially a copy of the IBM 370 and this fact permits the front end object modules to be transported to IBM systems.

Texas Instruments will take the object modules constituting the executable front end and transform them into 370 object modules that may be linked and run on the IBM 370 under VM. This process involves (possibly) rewritting certain routines (e.g., the run-time initialization routine), emulating certain Siemens instructions (e.g., LBF, STBF), fixing the object format (e.g., ESD cards), and emulating certain OS functions (e.g., output).

Texas Instruments will write a simple throw away IBM 370 code generator in Pascal that accepts Diana and outputs IBM 370 object modules. The code generator will generate code based on the Ada execution environment model. The Ada execution environment for the bootstrap compiler will be simplified to use only those features required by the subset of Ada needed to write and maintain the compiler for the Ada Integrated Environment. This subset will be similar to Ada-0 and will not include reals, tasking, and generics. The Ada execution environment will be written in Ada and 370 assembly code. This approach permits Texas Instruments to become familiar with Diana to generate code and to verify some of its Ada execution environment concepts.

With these two executable programs, i.e., the front end and the 370 code generator, Ada programs can be compiled, linked with the Ada execution environment modules, and executed. In particular, the Ada Integrated Environment tools can be compiled and executed in this manner.

SECTION 6

TEXT EDITOR

6.1 Text Editor

The Ada Interactive Text Editor is based upon the capabilities provided in two widely used text editors [DEC80, IBM80], which both access capabilities from two types of terminals.

6.1.1 Terminal Dependencies

Both text editors provide for access from a teletypewriter and a video display terminal (VDT). However, the VDTs from which they may be accessed differ greatly in the capabilities that they possess. The type of VDT used by the DEC EDT editor communicate with the host system at the character level, whereas the VDT used by the IBM XEDIT editor communicates with the host system at the screen level.

The Ada Editor is designed to function with a teletypewriter and both types of VDTs mentioned above. Designing an editor that functions similarly on all three types of terminals requires that a common ground be found among the capabilities of the terminals.

Commands from teletypewriters are limited to textual commands. Commands from VDTs may be in the form of special characters (control keys) or modification of text by overwriting existing text displayed on the screen of the VDT.

6.1.2 Editor Commands

Editor commands have been designed to be similar to natural language in order to make them easier to use by a novice and easier to remember by an experienced user. It has been demonstrated that commands similar to natural language decrease erroneous command entries, increase editing efficiency, and are prefered by users [LED80]. To enable an experienced user to decrease the number of keystrokes required to enter a command, each of the commands may be abbreviated to the shortest possible string which may be uniquely recognized.

Positions within the text being edited are specified by a name which may be chosen by the user. Hence, a user may reference portions of text by a mnemonic of his own choice, rather than a line and column number determined for him by the editor being used. The NAME command is used to give a "name" to a portion of text.

APPENDIX A

GLOSSARY

<u>Abstract Syntax Tree</u> -- An abstract syntax tree (AST) is a data structure built by the analyzer phase of a compiler to define the syntactic relationships between the tokens of the program unit.

<u>Accept Statement</u> -- An accept statement defines the actions to be performed when an entry of a task is called.

<u>Access Type</u> -- An access type is a type whose objects are pointers to dynamically created objects. The object itself is created by an <u>allocator</u>.

<u>Access Value</u> -- An access value designates an object pointed to by an entity of an access type.

<u>Ada Database Subsystem</u> -- The Ada Database Subsystem (ADS) provides data structures and facilities for storage and retrieval of information. Its components provide interfaces between the database and its users, provide utilities for the handling of database objects, and provide facilities for the management of users, access controls and security.

<u>Ada Language Processors</u> -- The Ada Language Processors transform the source text of program units into the machine code of target computers. These tools consist of an <u>Analyzer</u>, an <u>Expander/Optimizer</u>, and a <u>Code Generator</u>.

<u>Ada Software Environment</u> -- The Ada Software Environment (ASE) provides an interface between the user of the Ada Integrated Environment and the host system on which it is installed.

<u>Ada Programming Support Environment</u> -- An Ada Programming Support Environment (APSE) is a collection of software tools which provides facilities for the design, development, maintenance and management of software for one or more target computers.

<u>Ada Programming Toolset</u> -- The Ada Programming Toolset provides miscel tools or the development of Ada software; the minimal toolset includes a <u>Text Editor</u>, a <u>Program Binder</u>, and an <u>Interactive Debugger</u>.

<u>Address Space</u> -- Address space is a set of memory locations available for storage of programs and/or data.

<u>Aggregate</u> -- An aggregate is a written form denoting the value of an object of a <u>composite value</u>. An <u>array aggregate</u> denotes a value of an array type; a <u>record aggregate</u> denotes a value of a record type. The components of an aggregate may be specified using either <u>positional</u> or <u>named</u> association.

Allocator -- An allocator creates a new object of an access type and returns an access value designating the created object.

Analyzer -- An analyzer is a language translator that accepts source text for a compilation unit, performs lexical analysis, checks the syntax and static semantics of the compilation unit, and produces an intermediate representation that is more convenient for processing by compiler components and other tools. See also Front end.

Array Aggregate -- See Aggregate.

Array Type -- An array type is a collection of similar components addressed by one or more indices.

Asynchronous -- An event is said to be asynchronous if its occurrence is independent of other events in a system; e.g., depressing the break key causes an asynchronous program interrupt.

Attribute -- An attribute is a predefined characteristic of a named object.

Back End -- The back end of the Ada Optimizing Compiler is a language translator which accepts the DIANA dialect produced by the front end for a compilation unit and produces an object module for the compilation unit. The back end consists of the Expander/Optimizer and Code Generator passes.

Binder -- See Program Binder.

Block -- A block defines the scope of identifiers and other entities within an Ada program. A block statement contains an optional declarative part, followed by a sequence of statements, with an optional exception handler. Its body must be delimited by the BEGIN and END reserved words.

Body -- A body is a program unit defining the execution of a subprogram, package, or task. A body stub is a replacement for a body that is compiled separately.

Bootstrap Compiler -- A bootstrap compiler is an intermediate Ada compiler used for the development of the Ada Optimizing Compiler (which will compile itself).

Break Key -- A break key is a terminal keyboard key that interrupts execution of the current program.

Breakpoint -- A breakpoint is an event in a target program which causes execution to be suspended and passes control to the Debugger.

Call Handler -- The Call Handler is the Ada Execution Environment routine that implements subprogram calls; i.e., it transfers control from one subprogram to another.

Code Generator -- A code generator is a tool used to transform the declarations and statements of a program unit from an intermediate representation to a form compatible with the instruction set architecture of a target machine.

Code Section -- The Code Section is the portion of a bound program segment which contains the executable object code.

Code Section Dictionary -- The Code Section Dictionary is the portion of a bound program segment which contains entries indicating the locations of internal and external subprograms.

Collection -- A collection is the set of allocated objects of an access type.

Command File -- A command file is a file which contains a sequence of command language statements.

Command Language -- A command language is a collection of instructions to the Ada Integrated Environment specifying the execution of Ada programs; as such, it provides the user interface to the Ada Software Environment.

Command Language Interpreter -- The Command Language Interpreter (CLI) is a task within the Executive Program that is instantiated to translate and interpret the command language with which the Ada Integrated Environment user specifies the execution of Ada programs.

Command Procedure -- A command procedure is a file containing a sequence of command language instructions that is written in a form identical to an Ada procedure. A command procedure has its own name space; it may have parameters.

Compilation Unit -- A compilation unit is a program unit presented for compilation as an independent text. It is preceded by a context specification, naming the other compilation units on which it depends. A compilation unit may be the specification or body of a subprogram or package.

Compiler -- A compiler is a composite transformation tool consisting of a translator, optimizers, and a code generator. See also Ada Language Processors.

Component -- A component denotes one of a group of related objects known as a composite object. An indexed component names a component in an array or an entry in an entry family. A selected component is the identifier of the component, prefixed by the name of the entity of which it is a component, for instance, a discriminant within a record.

Composite Type -- An object of a composite type is a group of related objects known as components. An array type is a composite type, all of whose components are of the same type and subtype; the individual components are selected by their indices. A record type is a composite type whose components may be of different types; the individual components are selected by their identifiers.

Computer Program Component -- A Computer Program Component (CPC) is a functionally or logically distinct part of a Computer Program Configuration Item (CPCI) distinguished for purposes of convenience in designing and specifying a complex CPCI as an assembly of subordinate elements.

Computer Program Configuration Item -- A Computer Program Configuration Item (CPCI) is an aggregation of hardware/software which satisfies an end use function; a system segment.

Configuration -- A configuration is a collection of database objects that are related by some common property or requirement.

Configuration Management Tools -- Configuration Management Tools are used to record and control the changes made to constituent units of a software product so that the product is consistently constructed from known, compatible parts.

Constant Handler -- The Constant Handler is an Ada Execution Environment routine which determines the location of the constant section associated with a program unit in a code section.

Constant Section -- The Constant Section is the portion of a bound program segment which contains blocks of constants (read-only data) associated with program units.

Constant Section Dictionary -- The Constant Section Dictionary is the portion of a bound program segment whose entries indicate the location of internal constant blocks.

Constraint -- A constraint is a restriction on the set of possible values of a type. A range constraint specifies lower and upper bounds of the values of a scalar type. An index constraint specifies lower and upper bounds of an array index. A discriminant constraint specifies particular values of the discriminants of a record or private type.

Context Specification -- A context specification defines the other compilation units upon which a compilation unit depends.

Control File -- A control file provides an interface between a user and a running Ada program. It contains detailed instructions that specify the processing to be performed by the program.

Control Program -- The Control Program (CP) is the component of the VM/370 which acts as the virtual machine monitor. It simulates multiple virtual machines on a single physical machine.

Cross Reference Analyzer -- A cross reference analyzer is a tool that locates the definition of each symbol in a program unit and identifies the program statements that refer to the symbol.

Database Name -- The database name is the unique internal name of an object assigned by the Ada Database Subsystem when the object is created.

Debugger -- A source level debugger is a dynamic analysis tool that maps the memory image of an executing Ada program to the source program text and data definitions, allowing a user to examine or modify data values and to control program execution.

Declarative Part -- A declarative part is a sequence of declarations and related information such as subprogram bodies and representation specifications that apply over a region of a program text.

Demand Segmentation -- Demand segmentation is a method of memory management in which segments are loaded into memory as they are referenced. See also Segmentation.

Derived Type -- A derived type is a type whose operations and values are taken from those of an existing type.

DIANA -- DIANA is a high level intermediate language produced from the source code by the front end phase of the Ada Optimizing Compiler. This intermediate language is later optimized by the Expander/Optimizer and translated to machine language by the Code Generator.

Discrete Type -- A discrete type has an ordered set of distinct values. The discrete types are the enumeration and integer types. Discrete types may be used for indexing and iteration, and for choices in case statements and record variants.

Discriminant -- A discriminant is a syntactically distinguished component of a record. The presence of some record components (other than discriminants) may depend on the value of a discriminant.

Discriminant Constraint -- See Constraint.

Editor -- See Text Editor.

Elaboration -- Elaboration is the process by which a declaration achieves its effect. For example, it can associate a name with a program entity or initialize a newly declared variable.

Embedded Computer -- An embedded computer is designed for a specific function and resides in the system that performs the function.

Entity -- An entity is anything that can be named or denoted in a program. Objects, types, values, and program units are all entities.

Entry -- An entry is used for communication between tasks. Externally, an entry is called just as a subprogram is called; its internal behavior is specified by one or more accept statements specifying the actions to be performed when the entry is called.

Enumeration Type -- An enumeration type is defined by explicitly listing the values which that element may assume. These values may be either identifiers or character literals.

Exception -- An exception is an event that causes suspension of normal program execution. Bringing an exception to attention is called raising the exception.

Exception Handler -- An exception handler is a section of program text specifying a response to the exception.

Expander/Optimizer -- The expander/optimizer is the component of the Ada Optimizing Compiler which performs the expanding and optimizing functions within one pass. See also Back End.

Expression -- An expression is a part of a program that computes a value.

Executive Program -- The Executive Program is the component of the Ada Software Environment that provides the interface between a user and the program invoked from the user's terminal. The Executive includes the user's terminal interface and the Command Language Interpreter.

Front End -- The front end of the Ada Optimizing Compiler is a language translator which accepts the Ada source text for a compilation unit, performs lexical analysis, checks the syntax and static semantics of the compilation unit, and produces the intermediate representation (DIANA) of the compilation unit.

Garbage Collection -- Garbage Collection is a memory management technique that attempts to reclaim allocated memory space as soon as it is no longer designated by any variable.

Generic Clause -- See Generic program unit.

Generic Program Unit -- A generic program unit is a subprogram or package specified with a generic clause. A generic clause contains the declaration of generic parameters. A generic program unit may be thought of as a possibly parameterized model of program units. Instances (that is, filled-in copies) of the model can be obtained by generic instantiation. Such instantiated program units define subprograms and packages that can be used directly in a program.

Generic Expansion -- Generic expansion is the replacement of generic formal parameters in the Intermediate Language template for the generic declaration with the actual parameters.

Generic Instantiation -- Generic instantiation is the substitution of the actual parameters for the generic formal parameters in a copy of the generic dynamic specification.

Generic Optimization -- Generic optimization is accomplished by sharing code between different instantiations of a generic definition.

Global Package Handler -- The Global Package Handler is an Ada Execution Environment routine which determines the locations of visible parts of packages global to the program.

Global Package Table -- The Global Package Table is a table containing the locations of visible parts of packages global to the program.

Heap -- A heap is an area of memory reserved for dynamic variables. In Ada, dynamic variables are of the access type and are created by an Allocator.

Host Computer -- A host computer is a computer which supports a software development effort. It is expected to provide a general-purpose operating system with file management, resource management, scheduling, and other run-time support for all user programs.

Image Binding -- Image binding is a method of program binding by which the bound program is stored on disk in exactly the form it will have when loaded in memory; for example, the program contains all the inter-segment reference tables that will be needed at execution time.

Information Hiding -- Information hiding is the restriction of the visibility of an object or process to protect it from external influence. This function is handled in Ada by the private type.

Indexed Component -- See Component.

Interface -- An interface is a common design that allows communication between programs, tasks or data structures. See also KAPSE Virtual Interface.

Intermediate Language -- An Intermediate Language is a translation of the Abstract Syntax Tree generated by the Analyzer. This translation is usually machine-independent and may be further translated to object code.

Interrupt -- An interrupt is a response to an asynchronous or exceptional event that automatically saves the current CPU status (to allow later resumption) and causes an automatic transfer to a specified routine (called an interrupt handler).

Kernel Ada Programming Support Environment -- The Kernel Ada Programming Support Environment (KAPSE) provides the database, communication, and run-time support functions that enable the execution of an Ada program; these functions are a "kernel" in the sense that they provide a machine and operating system independent interface whose implementation on a host system uffices to install the Ada Integrated Environment. This interface is called the KAPSE Virtual Interface.

KAPSE Interface Task -- The KAPSE Interface Task (KIT) provides for interaction among the various components of the Ada Software Environment.

KAPSE Virtual Interface -- See Kernel Ada Programming Support Environment.

Lexical Descendents -- Lexical descendents are the subprograms that are nested within a parent subprogram.

Lexical Unit -- A lexical unit is one of the basic syntactic elements making up a program. A lexical unit is an identifier, a number, a character literal, a string, a delimiter, or a comment.

Library -- See Program Library.

Library File -- A library file is a separate database for maintaining the compilation state of a program or family of programs.

Library Unit -- A library unit is a compilation unit that is not a subunit of another compilation unit.

Literal -- A literal denotes an explicit value of a given type, for example a number, an enumeration value, a character, or a string.

Load-And-Go -- Load-And-Go is a method of program binding such that the resultant bound program is in a form ready for immediate execution.

Machine-Dependent Optimization -- Machine-dependent optimization includes optimizations performed on a program that are dependent on the target machine.

Machine-Independent Optimization -- Machine-independent optimization includes optimizations performed on a program that are language and target machine independent. Basically, they represent source-to-source transformations.

Main Program -- The main program of an Ada system is a designated subprogram which acts as a driver to the remainder of the package.

Minimal Ada Programming Support Environment -- The Minimal Ada Programming Support Environment (MAPSE) includes the compiler, text editor, debugger, terminal interface routines, project/ configuration control functions, and program binder.

Model Number -- A model number is an exactly representable value of a real numeric type. Operations of a real type are defined is terms of operations on the model numbers of the type. The properties of the model numbers and of the operations are the minimal properties preserved by all implementations of the real type.

Name -- A name denotes a declared entity, a result returned by a function call, or a label, block name, or loop name.

Named Association -- Named association indicates the value of an object by pecifying its identifier. See also Positional association.

Object -- Within the database, an object is a separately identifiable collection of information. Within an Ada program, an object can denote any kind of data element, whether a scalar value, a composite value, or a value in an access type.

Optimizer -- An optimizer is a tool used to analyze and transform a program unit to improve its performance or its utilization of computing resources.

OS/32 -- The OS/32 is the operating system of the Perkin-Elmer (Interdata) 8/32 computer.

Overlay -- An overlay is a portion of a program that resides on disk until it is referenced, at which time it is loaded into memory. Program Binder -- commands are provided to partition a program into overlays and to specify which overlays will share logical memory.

Overloading -- Overloading is the property that literals, identifiers, and operators can have several alternative meanings within the same scope. For example, an overloaded enumeration literal is a literal appearing in two or more enumeration types; an overloaded subprogram is a subprogram whose designator can denote one of several subprograms depending upon its parameter types and returned value.

Package -- A package is a program unit specifying a collection of related entities such as constants, variables, types and subprograms. The visible part of a package contains the entities that may be used from outside the package. The private part of a package contains structural details that are irrelevant to the use of the package but that complete the specification of the visible entities. The body of a package contains implementations of subprograms or tasks (possibly other packages) specified in the visible part.

Parameter -- A parameter is one of the named entities associated with a subprogram, entry, or generic program unit. a formal parameter is an identifier used to denote the named entity in the unit body. An actual parameter is the particular entity associated with the corresponding formal parameter in a subprogram call, entry call, or generic instantiation. The parameter mode specifies whether the parameter is to be passed into and/or returned by the program unit. A positional parameter is an actual parameter passed in positional order. A named parameter is an actual parameter passed by naming the corresponding formal parameter.

Parser -- A parser is a phase of a compiler that considers the context of each token returned by the syntax analyzer and classifies groups of tokens into larger entities such as declarations, statements and control structures; also referred to as lexical analyzer.

Partial Binding -- Partial binding is a technique of segment binding which allows the building of a program segment in multiple phases.

Pathname -- A pathname is a sequence of Ada identifiers that specifies the unique path through the directory hierarchy from the base or root to the specified object.

Positional Association -- Positional association specifies the value of an object based on its positional order. See also Named association.

Pragma -- A pragma is an instruction to the compiler, and may be language defined or implementation defined.

Primitives -- Primitives are functions which are accomplished directly by the Command Language Interpreter. They are indicated by the prefix "SYS." in the procedure name.

Private Type -- A private type is a type whose structure and set of values are clearly defined, but not known to the user of the type. A private type is known only by its discriminants and by the set of operations defined for it. A private type and its applicable operations are defined in the visible part of a package. Assignment and comparison for equality or inequality are also defined for private types, unless the private type is marked as limited.

Program Binder -- The program binder is a tool used to form a complete program from specified constituent program units. The binding process may merge program units from several libraries to create the desired program.

Program Library -- A program library is a collection of the compilation units of a program.

Program Parameter Area -- The Program Parameter Area (PPA) is an associative storage area used for passing parameters between program units.

Program Parameter Descriptor -- The program parameter descriptor is a block containing a list of parameter names and types required by the program. The program manager uses this information to obtain the parameter values from the user and pass them to a program.

Program Segment -- The subprograms that comprise an Ada program may be partitioned (by the program binder) into collections called segments. Intra-segment references are resolved, and inter-segment references are made through tables that facilitate sharing of code. A program segment consists of a Code Section, a Code Section Dictionary, a Constant Section and a Constant Section Dictionary.

Program Unit -- A program unit is the basic units of which programs may be composed. Units may be subprograms, packages, or tasks.

Qualified Expression -- A qualified expression is an expression qualified by the name of a type or subtype. For example, it can be used to state the type or subtype of an expression for an overloaded literal.

Raising An Exception -- See Exception.

Range -- A range is a contiguous set of values of a scalar type. A range is specified by giving the lower and upper bounds for the values.

Range Constraint -- See Constraint.

Record Aggregate -- See Aggregate.

Record Type -- A record type is a collection of similar or dissimilar components.

Rehost -- To rehost is to transport and adapt software from one host system to another.

Relation -- A relation is a labeled, directed arc that connects any two database objects.

Relative -- A relative is a database object associated with another database object through a relation.

Retarget -- To retarget is to adapt software which was designed to execute on a given target computer to run on another target machine.

Rendezvous -- A rendezvous is the interaction that occurs between two parallel tasks when one task has called an entry of the other task, and a corresponding accept statement is being executed by the other task on behalf of the calling task.

Representation Specification -- A representation specification defines the mapping between a data type and its implementation on the underlying machine. In some cases, it completely specifies the mapping, in other cases, it provides criteria for choosing a mapping.

Scalar Type -- A scalar type indicates an ordered set of values by enumerating the identifiers which denote the values. Scalar types comprise discrete types (that is, enumeration and integer types) and real types.

Scope -- The scope of a declaration is the region of text over which the declaration has an effect.

Segmentation -- Segmentation is the technique for *managing segments* in memory. A segment is a logical grouping of information, such as a subprogram. A Segment Table indicates the address of each segment in memory.

Selected Component -- See Component.

Semaphore -- A semaphore is an abstraction operated on by synchronization primitives to coordinate concurrent access to a resource.

Slice -- A slice is a one-dimensional array denoting a sequence of consecutive components of a one-dimensional array.

Stack -- A stack is a sequence of memory locations in which data may be stored or retrieved on a last-in-first-out (LIFO) basis. Storage for a task is allocated in a structure called a stack region, which is subdivided into stack frames. These stack frames are allocated on a LIFO basis as control enters and exits subprograms.

Static Expression -- A static expression is one whose value does not depend on any dynamically computed values of variables.

Subprograms -- A subprogram is an executable program unit, possibly with parameters for communication between the subprogram and its point of call. A subprogram declaration specifies the name of the subprogram and its parameters; a subprogram body specifies its execution. A subprogram may be a procedure, which performs an action, or a function, which returns a result.

Subtype -- A subtype of a type is obtained from the type by constraining the set of possible values of the type. The operations over a subtype are the same as those of the type from which the subtype is obtained.

Subunit -- A subunit is a body of a subprogram, package or task declared in the outermost declaration part of another compilation unit) which may be compiled separately.

Symbol Table -- A symbol table is a table built by a compiler which contains the characteristics of the identifiers used in the program.

Target Computer -- A target computer is the machine on which the specified software is designed to execute.

Task -- A task is a program unit that may operate in parallel with other program units. A task specification establishes the name of the task and the names and parameters of its entries; a task body defines its execution. A task type is a specification that permits the subsequent declaration of any number of similar tasks.

Text Editor -- A text editor is a tool used to form program units from smaller constituent parts. The editing process may include direct text entry, deletion or changes by an interactive user, or may merge text from several source files to create the desired program unit.

Type -- A type defines the structure of a data element (enumeration, integer, real, array, record, or access), the values which the element may assume, and the operations which may be performed on the element. A type definition is a language construct *introducing* a type. A type declaration associates a name with a type introduced by a type definition.

Use Clause -- A use clause opens the visibility to declarations given in the visible part of a package.

Variant -- A variant part of a record specifies alternative record components, depending on a discriminant of the record. Each value of the discriminant establishes a particular alternative of the variant part.

Virtual Machine -- A computer architecture is said to support a virtual machine if it permits multiple instances of the architecture to be simulated on a single processor. Each user is given the full capabilities of the processor.

Virtual Terminal -- A virtual terminal is a logical terminal to which the input/output of an executing program may be directed; a virtual terminal may be connected to an actual terminal through a command to the Executive Program.

Visibility -- The declaration of an entity with a certain identifier is said to be visible at a given point in the text when an occurrence of the *identifier* at this point can refer to the entity, that is, when the entity is an acceptable meaning for this occurrence.

VM/370 -- The VM/370 (Virtual Machine /370) is an operating system of the IBM/370 computer that supports virtual machines.

Window -- A window is a portion of a physical terminal which may be connected to a virtual terminal. In peephole optimization, a window is the sequence of instructions being viewed.

With Clause -- A with clause is used to create an implicit declaration of the named library units.

APPENDIX B

REFERENCES

B.1 Program Definition Documents

[DoD80A] Requirements for Ada Programming Support Environments: STONEMAN, DoD (February 1980).

[RADC80] Revised Statement of Work for Ada Integrated Environments, RADC, Griffiss Air Force Base, NY (March 1980).

[SOFT80A] Ada Compiler Validation Capability: Long Range Plan, SofTech Inc., Waltham, MA (February 1980).

[SOFT80B] Draft Ada Compiler Validation Implementers' Guide, SofTech Inc., Waltham, MA (October 1980).

B.2 Military Specifications and Standards

[DoD80B] Reference Manual for the Ada Programming Language: Proposed Standard Document, DoD (July 1980) (reprinted November 1980).

B.3 Other References

[AHO77] Aho, A.V. and J. D. Ullman, Principles of Compiler Design, Addison-Wesley Publishing Co., Reading Ma., (1977).

[ALL76] Allen, F.E., A Program Data Flow Analysis Procedure, CACM 19, 3 (March 1976), 137-147.

[APP79] Applewhite, C.M., Distributed Computer Architecture for the Discrete Address Beacon System, Proceedings of 1st International Conference on Distributed Computing Systems, Huntsville, Alabama, (October 1979).

[BAK80] Baker, Henry G. Jr., List Processing in Real Time on a Serial Computer, CACM 21, 4 (April 1978), 280-294.

[BAR80] Barnes, J.G.P., An Overview of Ada, Software-Practice and Experience, 10 (November 1980), 851-887.

[BAR79] Barrett, W.A. and J.D. Couch, Compiler Construction: Theory and Practice, SRA (1979).

[BAT76] Bates, D. (editor), Program Optimization, Infotech State of the Art Report, Infotech International Limited (1976).

[BAT79] Bate, R.R. and D.S. Johnson, Putting Pascal to Work, Electronics, 7 (June 1979), 111-121.

[BEL80] Belmont, P.A., Type Resolution in Ada: An Implementation Report, SIGPLAN Notices, 15, 11 (November 1980), 57-61.

[BOE76] Boehm, B.W., Software Engineering, IEEE Transactions on Computers, C-25, 12 (December 1976), 1226-41.

[BOH66] Bohm, C. and G. Jacopini, Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules, CACM, 9, 5 (May 1966), 366-71.

[BOO80] Booch, Lt. E.G., ADA Tutorial, U.S. Air Force Academy, (Fall 1980).

[BOU80] Boute, R.T., Simplifying Ada by Removing Limitations, SIGPLAN Notices, 15, 2 (February 1980), 17-29.

[BRA80] Bradshaw, F.T., et.al., Procedure Semantics and Language Definition, SIGPLAN Notices, 15, 6 (June 1980), 28-33.

[BRE80] Brender, R., The Case Against Ada as an APSE Command Language, SIGPLAN Notices, (Oct 1980), 27-32.

[BRO80A] Brosgol, B.M., et.al., TCOL Ada: Revised Report on an Intermediate Representation for the Preliminary Ada Language, Department of Computer Science, Carnegie-Mellon University (February 1980).

[BRO80B] Brosgol, B.M., TCOL-Ada and the "Middle End" of the PQCC Ada Compiler, SIGPLAN Notices, 15, 11 (November 1980), 101-112.

[BUR77] Burroughs Corporation, B7000/B6000 Series I/O Subsystem Reference Manual, Number 5001779, (September 1977).

[BUX80] Buxton, J.N., L.E. Druffel and V. Stenning, Rationale for STONEMAN, Proc. COMSAC, Chicaco, October 1980.

[CAR80] Carlson, W.E., Ada: A Standard Programming Language for Defense Systems, SIGNAL, (March 1980), 25-28.

[CAT77] Cattell, R.G., A Survey and Critique of Some Models of Code Generation, Department of Computer Science, Carnegie-Mellon University (November 1977).

[CAT78] Cattell, R.G., Formalization and Automatic Derivation of Code Generation, PhD Thesis, Carnegie-Mellon University (April 1978).

[CAT79] Cattell, R.G., et. al., Code Generation in a Machine-Independent Compiler, SIGPLAN Notices, 14, 8 (August 1979), 65-75.

[CAT80] Cattell, R.R., Automatic Derivation of Code Generators from Machine Descriptions, ACM Transactions on Programming Languages and Systems, 2, 2 (April 1980), 173-190.

[CHA79] Champine, G.A., Current Trends in Data Base Systems, Computer, 12, 5 (May 1979), 27-41.

[CII80] CII Honeywell Bull, Formal Definition of the Ada Programming Language, Louveciennes, France (November 1980).

[COD70] Codd, E.F., A Relational Model of Data for Large Shared Data Banks, CACM, 13, 6 (June 1970).

[COD71] Codd, E.F., Normalized Data Base Structure: A Brief Tutorial, Proceedings 1971 ACM SIGFIDET Workshop on Data Description, Access and Control (1971).

[COD71B] Codd, E.F., A Data Base Sublanguage Founded on the Relational Calculus, Proceedings 1971 ACM SIGFIDET Workshop on Data Description, Access and Control (1971).

[COD72A] Codd, E.F., *Relational Completeness of Data Sublanguages, in* Data Base Systems, Courant Computer Science Symposia Series, Vol. 6, Prentice-Hall (1972).

[COD72B] Codd, E.F., Further Normalization of the Data Base Relational Model, in Data Base Systems, Courant Computer Science Symposia Series, Vol. 6, Prentice-Hall, (1972).

[COD74] Codd, E.F., Recent Investigations in Relational Data Base Systems, Proceedings IFIP Congress, (1974).

[COM79] Comer, Doug, The Ubiquitous B-Tree, Computing Surveys, (1979).

[COO79] Cooprider, L.S., The Representation of Families of Software Systems, Carnegie-Mellon University PhD thesis (1979).

[COR80] Cornhill and Gordon, ADA - The Latest Word in Process Control, Electronic Design, (1 September 1980), 111-116.

[DAT71] Date, C.J. and P. Hopewell, Storage Structure and Physical Data Independence, Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, (1971).

[DAT75] Date, C.J., Relational Database Systems: A Tutorial, Proc. 4th International Symposium on Computers and Information Science, Plenum Publishing Corp., (1975).

[DAT75-77] Date, C.J., An Introduction to Database Systems, Addison-Wesley

Publishing Co., Reading, Ma., (1975-1977).

[DAU79A] Dausmann, M., et. al., Notes on TCOL, Universitat Karlsruhe, West Germany (October 1979).

[DAU79B] Dausmann, M., et. al., AIDA: An Intermediate Representation of Ada Programs - Global Design, Universitat Karlsruhe, West Germany (November 1979).

[DAU79C] Dausmann, M., et. al., AIDA: An Intermediate Representation of Ada Programs, Universitat Karlsruhe, West Germany (November 1979).

[DAU80A] Dausmann, M., et. al., AIDA: An Informal Introduction, Universitat Karlsruhe, West Germany (February 1980).

[DAU80B] Dausmann, M., et. al., AIDA: Reference Manual (Preliminary Draft), Universitat Karlsruhe, West Germany (February 1980).

[DAU80C] Dausmann, M., et. al., Command Interpreter of the Library-User-System (User Information), Universitat Karlsruhe, West Germany (July 1980).

[DAU80D] Dausmann, M., et. al., AIDA: An Informal Introduction (Draft), Universitat Karlsruhe, West Germany (November 1980).

[DAU80E] Dausmann, M., et. al., AIDA: Reference Manual (Draft), Universitat Karlsruhe, West Germany (November 1980).

[DAU80F] Dausmann, M., et. al., SEPAREE: A Separate Compilation System for Ada (Draft), Universitat Karlsruhe, West Germany (November 1980).

[DAV80] Davidson, J.W. and C. W. Fraser, The Design and Application of a Retargetable Peephole Optimizer, ACM Transactions on Programming Languages and Systems, 2, 2 (April 1980), 191-202.

[DED80] Dedourek, J.M. and U. G. Gujar, Scanner Design, Software-Practice and Experience, 10 (December 1980), 959-972.

[DEM80] Demarco, T., The Ada-Pascal Schism, Yourdon Report, (October 1980).

[DER76] DeRemer, F. and H.H. Kron, Programming-in-the-large versus Programming-in-the-small, IEEE Trans. Soft Eng., June 1976.

[DER80] DeRemer, F., T. Pennello, and R. Meyers, A Syntax Diagram for (Preliminary) Ada, SIGPLAN Notices, 15,7/8, (July-August, 1980), 36-47.

[DEV80] Devlin, M., Preliminary ADA Introduction Plan for the Air Force Satellite Control Facility Data System Modernization Program, (January 1980).

[DIJ65] Dijkstra, E., Programming Considered as a Human Activity, Proceedings of the 1965 IFIP Congress, North-Holland Publishing Co., (1965).

[DIJ68] Dijkstra, E., Co-operating Sequential Process, in Programming Languages (ed. F. Genuys), Academic Press, New York, (1968).

[DON72] Donovan, J.J., Systems Programming, McGraw-Hill Book Co. (1972).

[EVA80] Evans, Arthur Jr., Slides on Tasking in Ada, Ada Implementors Newsletter, (September 1980).

[EVE80] Eventoff, W., D. Harvey, and R. J. Rice, The Rendezvous and Monitor Concepts: Is There an Efficiency Difference? SIGPLAN Notices 15, 11 (November 1980), 156-165.

[FAI80] Faiman, R.N. and A.A. Kortesoja, An Optimizing Pascal Compiler, IEEE Transactions on Software Engineering, Vol. SE-6, No. 6 (November 1980), 512-518.

[FIR80A] Firth, R., Presenting Revised Ada, Presentation at NCC 80.

[FIR80B] Firth, R., Universal Ada Language Issue Report Construction Kit, SIGPLAN Notices, 15, 5 (May 1980), 35-36.

[FIS78] Fisher, C.A. and P.R. Wetherall, STEELMAN, Department of Defense Requirements for High Order Computer Programming Languages, HOLWG, (June 1978).

[FIS79] Fisher, D.A. and T.A. Standish, Initial Thoughts on the Pebbleman Process, IDA Paper P-1392, (June 1979).

[FIS80] Fisher, D.A., Design Issues for Ada Program Support Environments, a Catalogue of Issues, SAI-81-289-WA, (October 1980).

[FRA79] Fraser, C.W., A Compact, Portable CRT-Based Text Editor, Software-Practice and Experience, 9, Department of Computer Science, University of Arizona, (1979), 121-125.

[FRA80] Fraser, C.W., A Generalized Text Editor, CACM, 23, 3 (March 1980), 154-158.

[GAL80] Galkowski, J.T., A Critique of the DOD Common Language Effort, SIGPLAN Notices, 15, 6 (June 1980), 15-18.

[GAN80] Ganzinger, H. and K. Ripken, Operator Identification in Ada: Formal Specification, Complexity, and Concrete Implementation, SIGPLAN Notices, 15, 2 (February 1980), 30-42.

[GES72] Geschke, C.M., Global Program Optimization, PhD Thesis, Department of Computer Science, Carnegie-Mellon University

(October 1972).

[GLA78] Glassei, A.L., The Evolution of a Source Code Control System Design, Proc. Software Quality Assurance Workshop, November 1978.

[GLA79] Glass, Robert, From Pascal to Pebbleman and Beyond, Datamation, (July 1979).

[GOO80A] Goodenough, J.B., Ada (July 1980) Syntax Cross Reference Listing, SofTech, Inc.,48-56.

[GOO80B] Goos, G. and G. Winterstein, Towards a Compiler Front-End for Ada, SIGPLAN Notices, 15, 11 (November 1980), 36-46.

[GRA79A] Graham, S.L., W.N. Joy, and O. Roubine, Hashed Symbol Tables *for Languages with Explicit Scope Control, Proceedings of the* SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices, 14, 8 (August 1979) 50-57.

[GRA80] Graham, S.L., Table-Driven Code Generation, Computer (August 1980), 25-34.

[GRI71] Gries, D., Compiler Construction for Digital Computers, John Wiley and Sons, Inc., New York, (1971).

[HAB80] Habermann, A.N., and I. Nassi, Efficient Implementation of Ada Tasks, Carnegie-Mellon University Report CS-80-103, (February 1980).

[HAL78] Hall, D., D. Scherrer, and J. Sventek, The Software Tools Programmers Manual, Internal Rep. LBID097, LBL, University of California, Berkeley, California, (1978).

[HAL80] Hall, D. et. al., A Virtual Operating System, CACM, 23, No. 9, (September 1980).

[HAR75] Harrison, W.H., A Class of Register Allocation Algorithms, IBM Watson Research Center, Yorktown Heights, NY (March 1975).

[HAR79] Harrison, W.H., A New Strategy for Code Generation-the General-Purpose Optimizing Compiler, IEEE Transactions on Software *Engineering, Vol.* SE-5, No. 4 (July 1979), 367-373.

[HAR77] Hartmann, A.C., A Concurrent Pascal Compiler for Minicomputers, Springer-Verlag, Berlin (1977).

[HEC77] Hecht, M.S., Flow Analysis of Computer Programs, American-Elsevier, New York, NY (1977).

[HIS80] Hisgen, A. et. al., A Runtime Representatin for Ada Variables and Types, SIGPLAN Notices, 15, 11 (November 1980), 82-90.

[HOL79] HOLWG, Ada Environment Workshop, Harbor Island, San Diego, (November 1979).

[HP 77] Hewlitt-Packard Company, HP3000 Series II Computer System: System Reference Manual, Santa Clara, California, (1977).

[IBM71] PL/I (F) Compiler Program Logic Manual, IBM Corporation, Order No. GY28-6800-5 (December 1971).

[IBM72A] FORTRAN IV (H) Compiler Program Logic Manual, IBM Corporation, Order No. GH28-6642-5 (October 1972).

[IBM72B] IBM Corporation, System/360 Operating System System Generation, IBM order no. GC28-6554-1, 1972.

[IBM76] IBM Corporation, IBM System/370 Principles of Operation, IBM order no. GA22-7000-5, (1976).

[IBM77] IBM Corporation, Display Editing System for CMS Users Guide, IBM order no. SH20-1965-0, (1977).

[IBM79A] IBM Corporation, VM/370 Publications, VM/370 Introduction, IBM order no. GC20-1800, (1979).

[IBM79B] IBM Corporation, VM/370 Publications, VM/370 Terminal Users Guide, IBM order no. GC20-1810, (1979).

[IBM79C] IBM Corporation, VM/370 Publications, VM/370 CP Command Reference for General Users, IBM order no. GC20-1820, (1979).

[IBM79D] IBM Corporation, VM/370 Publications, VM/370 System Messages, IBM order no. GC20-1808, (1979).

[IBM79E] IBM Corporation, VM/370 Publications, VM/370 Planning and System Generation Guide, IBM order no. GC20-1801, (1979).

[IBM79F] IBM Corporation, VM/370 Publications, VM/370 Operating Systems in a Virtual Machine, IBM order no. GC20-1821, (1979).

[IBM79G] IBM Corporation, VM/370 Publications, VM/370 System Programmers Guide, IBM order no. GC20-1807, (1979).

[IBM79H] IBM Corporation, VM/370 Publications, VM/370 Release 6 Guide, IBM order no. GC20-1834, (1979).

[IBM79I] IBM Corporation, VM/370 Publications, VM/370 Operators Guide, IBM order no. GC20-1806, (1979).

[IBM79J] IBM Corporation, VM/370 Publications, VM/370 Quick Guide for Users, IBM order no. GC20-1926, (1979).

[IBM79K] IBM Corporation, IBM Virtual Machine Facility/370 Display Management System for CMS: Guide and Reference, IBM order no.

SC24-5198-0, (1979).

[IBM79L] IBM Corporation, IBM Virtual Machine Facility/370: CMS User's Guide, IBM Order no. GC20-1819-2, (1979).

[IBM80A] IBM Corporation, IBM Virtual Machine/System Product: System Product Editor Command and Macro Reference, IBM order no. SC24-5221-0, (1980).

[IBM80B] IBM Corporation, IBM Virtual Machine/System Product: System Programmer's Guide, IBM Order no. SC19-6203-0, (1980).

[ICH79A] Ichbiah, J.D. et. al., Rationale for the Design of the Ada Programming Language, SIGPLAN Notices, 14, 6 (June 1979), (AD A 071 761).

[ICH79B] Ichbiah, J.D. et. al., Reference Manual for the ADA Programming Language, SIGPLAN Notices 14, 6:A (June 1979), (AD A 071 761).

[INT80] Intermetrics, Inc and Carnegie-Mellon University, TCOL-Ada: Revised Report on An Intermediate Representation for the Preliminary Ada Language, Intermetrics Report IR-459, (February 1980).

[IRO72] Iron, E.T. and F.M. Djorup, A CRT Editing System, CACM, 15, 1 (1972), 16-20.

[IVI77] Ivie, E.L., Programmers Work Bench - A Machine for Software Development, CACM 20, 10 (October 1977), 746-753.

[JAN80] Janan, J.M., A Comment on Operator Identification in ADA, SIGPLAN Notices, 15, 9 (September 1980), 39-43.

[JEN79A] Jensen, R.M., A Formal Approach for Communication Between Logically Isolated Virtual Machines, IBM System Journal, 18, no. 1 (1979), 71-92.

[JEN79B] Jensen, R.M. and C.C. Tonies, Software Engineering, Prentice-Hall, (1979).

[JOH78] Johnson, D., C. Kolberg and J. Sinnamon, A Programmable System for Software Configuration Management, Texas Instruments, Inc., Dallas, Texas, (September, 1978).

[JOH80] Johnson, D. and J. Cointment, A Library Management System to Support Ada Programming, Advanced Computer Systems Lab, Texas Instruments, (May 1980).

[JOH75] Johnsson, R.K., An Approach to Global Register Allocation, PhD Thesis, Department of Computer Science, Carnegie-Mellon University (December 1975).

[JON80] Jones D.W., Tasking and Parameters: A Problem Area in Ada, SIGPLAN Notices, 15, 5 (May 1980), 37-40.

[KOR80] Kornerup, P., et. al., Interpretation and Code Generation Based on Intermediate Languages, Software Practice and Experience, 10, 8 (August 1980), 635-658.

[KNU73A] Knuth, D., The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison-Wesley Publishing Co. (1973).

[KNU73B] Knuth, D., The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley Publishing Co. (1973).

[LAM80A] Lamb., David A., Construction of a Peephole Optimizer, Department of Computer Science, Carnegie-Mellon University (August 1980).

[LAM80B] Lamb, David A., et.al., The Charrette Ada Compiler, Department of Computer Science, Carnegie-Mellon University (October 1980).

[LAU79] Lauer, H.C. and E. Satterthwaite, The Impact of MESA on System Design, Proc 4th International Conference on Software Engineering, Munich, 1979.

[LEB79] LeBanc, R.J. and C.N. Fischer, On Implementing Separate Compilation in Block-Structured Languages, SIGPLAN Notices, 14, 8 (August 1979), 139-143.

[LED80] Ledgard, H., et.al., The Natural Language of Interactive Systems, CACM, 23, 10 (October 1980).

[LEF69] Lefkovitz, D., File Structures for On-line Systems, Spartan Books (1969).

[LOV80A] Loveman, Ada Defines Reliability as a Basic Feature, Electronic Design, (27 September 1980), 93-98.

[LOV80B] Loveman, Ada Knack or Multitasking Benefits Process Control, Electronic Design, (6 December 1980), 101-105.

[LOV80C] Loveman, Subprograms and Types Boost Ada Versatility, Electronic Design, (25 October 1980), 153-158.

[MAC79] MacKinnon, R.A., The Changing Virtual Machine Environment: Interfaces to Real Hardware, Virtual Hardware, and Other Virtual Machines, IBM System Journal, 18, 1 (January 1979), 18-46.

[MAC77] MacLeod, I.A., Design and Implementation of a Display Oriented Text Editor, Software -- Practice and Experience, 7 (1977), 771-778.

[MAD74] Madnick, S.E. and J.J. Donovan, Operating Systems, McGraw-Hill

Book Co., (1974).

[MAR77] Martin, J., Computer Database Organization, 2nd edition, Prentice-Hall, (1977).

[MIN79] Mintz, R.J. et. al., The Design of a Global Optimizer, SIGPLAN Notices, 14, 8 (August 1979), 226-234.

[MoD] United Kingdom Ministry of Defence, Ada Support System Study: Phase 2 and 3 Reports.

[MOL79] Molina, F.W., A Practical Data Base Design Method, Data Base, 11, 1 (Summer 1979), 3-11.

[PAL75] Palermo, F.P., A Data Base Search Problem, Proc. 4th International Symposium on Computers and Information Science, Plenum Publishing Corporation (1975).

[PAR76] Parnas, D.L., On the Design and Development of Program Families, IEEE Trans. Soft Eng., March 1976.

[PEA79] Pearson, D.J., The Use and Abuse of a Software Engineering System, Proc. NCC 1979.

[PEM80] Pemberton, S., Comments on an Error-recovery Scheme by Hartmann, Software-Practice and Experience, 10 (1980), 231-240.

[PEN80] Pennello, T., F. DeRemer, and R. Meyers, A Simplified Operator Identification Scheme for Ada, SIGPLAN Notices, 15, 7&8 (July-August 1980), 82-87.

[PER78] Perkin-Elmer Corporation, M83-Series Models 8/32, 8/32C, and 8/32D Processors User Manual, Publication Number 29-428R06, Computer Systems Division, Perkin-Elmer Corporation, Oceanport, N. J., (1978).

[PER79] Perkin-Elmer Corporation, OS/32 Programmer Reference Manual, Publication Number S29-613R04, Computer Systems Division, Perkin-Elmer Corporation, Oceanport, N. J., (1979).

[PER80] Persch, G. et. al., Overloading in Preliminary Ada, SIGPLAN Notices, 15,11 (November 1980), 47-56.

[RADC74] IBM Corporation, Programming Support Library Functional Requirements, Structured Programming Series (RADC TR 74-300), 5, (1974), (AD A 003 339).

[RADC80] Martin Marietta Aerospace Corporation, Recommendations for a Retargetable Compiler, Final Technical Report (RADC TR-79-351), (March 1980).

[ROC75] Rochkind, M.J., The Source Code Control System, IEEE Transactions on Software Engineering, SE-1, 4 (December 1975), 346-349.

[ROS80] Rosenberg, J. et. al., The Charrette Ada Compiler SIGPLAN Notices, 15, 11 (November 1980), 72-81.

[ROS77] Ross, D.T. and K.E. Schoman, Structured Analysis for Requirements Definitions, IEEE Transactions on Software Engineering, SE-3, 1 (January 1977), 6-15.

[RUD79] Rudmik, A. and E.S. Lee, Compiler Design for Efficient Code Generation and Program Optimization, SIGPLAN Notices, 14, 8 (August 1979), 127-138.

[SCH73] Schaefer, M., A Mathematical Theory of Global Program Optimization, Prentice-Hall, Inc., New York, NY (1973).

[SCH77] Scheifler, R.W., An Analysis of Inline Substitution for a Structured Programming Language, CACM 20,9 (September 1977), 647-654.

[SCH79] Scherrer, D., Cookbook, Instructions for Implementing the LBL Software Tools Package, Internal Rep. LBID 098, Lawrence Berkeley Laboratory, University of California, Berekeley, California, (1979).

[SCH80A] Scheer, L.S. and M.G. McCleens, DODs Ada Compared to Present Military Standard Holds A Look at New Capatilities, Engineering Experimentation, Georgia Institute of Technology.

[SCH80B] Schofield, D. et.al., MM/1, A Man-Machine Interface, Software Practice and Experience, 10, (1980), 751-763.

[SEA79] Seawright, L.H., and R.A. MacKinnon, VM/370-A Study of Multiplicity and Usefulness, IBM System Journal, 18, No. 1, (1979), 4-17.

[SHA80] Shankar, K.S., Data Structures, Types, and Abstractions, Computer, 13, 4 (April 1980), 67-77.

[SHE80A] Sherman, M.S. and M.S. Borkan, A Flexible Semantic Analyzer for Ada, SIGPLAN Notices, 15, 11 (November 1980), 62-71.

[SHE80B] Sherman, M. et. al., An Ada Code Generator for VAX 11/780 with Unix, SIGPLAN Notices, 15, 11 (November 1980), 91-100.

[SIT79A] Sites, R.L., Machine-Independent Register Allocation, SIGPLAN Notices, 14, 8 (August 1979), 221-225.

[SIT79B] Sites, R.L. and D.R. Perkins, Universal P-Code Definition, Version 0.3, Department of Electrical Engineering and Computer Sciences, University of California at San Diego (July 1979).

[SMA80] Smart, R., Pointers to Local Variables, SIGPLAN Notices, 15, 7/8

(July-August 1980), 88-89.

[STA78] Standish, T.A., Proceedings of Workshop on Environment, Certification and Control of DoD Common High Order Language, University of California - Irvine, (June 1978).

[STA80] Stallman, R.M., EMACS Manual for TOPS-20 Users, AI Memo 554, MIT Artificial Intelligence Laboratory, (1980).

[STE75] Steele, Guy L. Jr., Multiprocessing Compactifying Garbage Collection, CACM 18, 9 (September 1975), 495-508.

[STE79] Stenning, Vic, et. al., Ada Support System Study -- Requirements and Functions Specification, SDL and SSL, (March 1979).

[TAI80] Tai, Kuo-Chung and K. Garrard, Comments on the Suggested *Implementation of Tasking Facilities in the* "*Rationale for the* Design of the ADA Programming Language", SIGPLAN Notices, 15, 10 (October 1980), 76-84.

[TAU79] Tausworthe, R.C., Standardized Development of Computer Software, Prentice-Hall, Inc., (1979).

[TEO80] Teory, T.J. and J.P. Fry, The Logical Record Access Approach to Database Design, Computing Surveys, 12, 2 (June 1980), 179-211.

[TI 78A] Texas Instruments Incorporated, Model 990 Computer TI Pascal Users Manual, TI Manual No. 946290-9701, (May 1978).

[TI 78B] Texas Instruments Incorporated, Model 990 Computer DX10 Operating System Release 3 Reference Manual, Volume IV, Developmental Operation, TI Manual No. 946250-9704, (1978).

[TI 79A] Texas Instruments Incorporated, *TI Pascal Configuration* Processor Tutorial, TI Manual No. 2250098-9701, (January 1979).

[TI 80A] Texas Instruments Incorporated, TIFORM Reference Manual, TI Manual no. 2250374-9701, (1980).

[TI 80B] Texas Instruments Proposal to Design and Develop from ADA Language Environments, Vol 1 Computer Program Development Plan (June 1980)

[TI 81A] Texas Instruments Incorporated, Design of the Ada Integrated Environment, Equipment Group, Texas Instruments Incorporated, Lewisville, Texas, (1981).

[TI 81B] Texas Instruments Incorporated, Device Independent File I/O User's Manual, MP386, Semiconductor Group, Texas Instruments Incorporated, Houston, Texas, (1981).

[TI 81C] Texas Instruments Incorporated, Microprocessor Pascal Executive User's Manual, MP385, Semiconductor Group, Texas Instruments Incorporated, Houston, Texas, (1981).

[TI 81D] Texas Instruments Incorporated, Microprocessor Pascal System User's Manual, MP351 (Revision B), Semiconductor Group, Texas Instruments Incorporated, Houston, Texas, (1981).

[TIC79] Tichy, W.F., Software Development Based on Module Interconnection, Proc. 4th International Conference Software Engineering, Munich 1979.

[UNIX78A] Ritchie, D.M. and K. Thompson, The UNIX Time-sharing System, Bell System Technical Journal 57, 6, (July 1978).

[UNIX78B] Thompson, K., UNIX Implementation, Bell System Technical Journal 57, 6 (July 1978).

[UNIX78C] Bourne, S.R., The UNIX Shell, Bell System Technical Journal 57, 6 (July 1978).

[UNIX78D] Johnson, S.C., and D.M. Ritchie, Portability of C Programs and the UNIX System, Bell System Technical Journal 57, 6 (July 1978).

[UNIX78E] Ritchie, D.M., A Retrospective, Bell System Technical Journal 57, 6 (July 1978).

[VDB80] van den Bos, Jan, Comments on ADA Process Communication, SIGPLAN Notices, 15, 6 (June 1980), 77-81.

[WAD76] Wadler, Philip L., Analysis of an Algorithm for Real-Time Garbage Collection, CACM 19, 9 (September 1976), 491-500.

[WAR79] Warren, Scott K., and Dennis Abbe, Rosetta Smalltalk: A Conversational, Extensible Microcomputer Language, SIGSMALL Newsletter, 5, 2 (April 1979), 36-45.

[WEL78] Welsh, J., Economic Range Checks in Pascal, Software-Practice and Experience, 8 (1978), 85-97.

[WEG80A] Wegner, Peter, Programming with Ada: an Introduction by means of Graduated Examples, Prentice Hall, Englewood Cliffs, NJ, (1980).

[WEG80B] Wegner, Peter, Conference Report: Seventh Annual ACM Symposium on the Principles of Programming Languages, SIGPLAN Notices, 15, 5 (May 1980), 66-77.

[WIR76] Wirth, N., Algorithms + Data Structures = Programs, Prentice Hall, (1976).

[WUL75] Wulf, W.A., et.al., The Design of an Optimizing Compiler, American-Elsevier, New York, NY (1975).

[WUL79] Wulf, W.A., et.al., An Overview of the Production Quality Compiler-Compiler Project, Department of Computer Science, Carnegie-Mellon University, (February 1979).

[WUL80A] Wulf, W.A., et. al., An Overview of the Production-Quality Compiler-Compiler Project, Computer (August 1980) 38-49.

[WUL80B] Wulf, W.A., PQCC: A Machine-Relative Compiler Technology, Department of Computer Science, Carnegie-Mellon University (September 1980).

DA
FIL