SYSTEM ARCHITECT'S APPRENTICE (SARA) AS THE FOUNDATION FOR A METHODOLOGY-ORIENTED ADA® PROGRAMMING SUPPORT ENVIRONMENT

Eduardo Aaron Krell

January 1987
CSD-870001

The dissertation of Eduardo Aaron Krell is approved.

Eva Baker

Stephen Jacobsen

David F. Martin

Gerald Estrin

Daniel M. Berry, Committee Chair

University of California, Los Angeles

1987

TABLE OF CONTENTS

1 **Introduction** 1
1.1 Motivation for the Research 1
1.2 Related Research Areas 4
1.3 Research Hypothesis 5
1.4 Research Plan 6
1.5 Dissertation Organization 6

2 **Related Research** 8
2.1 Ada Environments 8
2.1.1 ATMAda 8
2.1.2 Arcturus 9
2.1.3 CAEDE 11
2.2 Ada-compatible Design Methods 13
2.2.1 METHODMAN 13
2.2.2 The U.K. Study 14
2.2.3 CORE/MASCOT 17
2.2.4 JSD 18
2.2.5 CCS 19
2.3 Discussion 19

3 **STONEMAN - Requirements for APSEs** 21
3.1 Components of an APSE 21
3.2 The Layered Approach 22
3.2.1 The KAPSE 24
3.2.2 The MAPSE 24
3.2.3 The APSE 25
3.3 APSEs and design methods 26

4 **SARA/IDEAS – A Computer-Based System Design Method** 28
4.1 The Structure Model 30
4.2 The Graph Model of Behavior 32
4.2.1 The Control Domain 32
4.2.2 The Data Domain 35
4.2.3 The Interpretation Domain 38
4.3 The Building Block Library 38
4.4 The Socket Attribute Model 39
4.5 The Module Interconnect Description 40

UNIVERSITY OF CALIFORNIA

Los Angeles

System Architect's Apprentice (SARA) as the Foundation for a Methodology-Oriented Ada® Programming Support Environment

A dissertation submitted in partial satisfaction of the requirement for the degree of Doctor of Philosophy in Computer Science

by

Eduardo Aaron Krell

1987

The dissertation of Eduardo Aaron Krell is approved.

Eva Baker

Stephen Jacobsen

David F. Martin

Gerald Estrin

Daniel M. Berry, Committee Chair

University of California, Los Angeles

1987

TABLE OF CONTENTS

1 Introduction **1**
- 1.1 Motivation for the Research 1
- 1.2 Related Research Areas 4
- 1.3 Research Hypothesis 5
- 1.4 Research Plan 6
- 1.5 Dissertation Organization 6

2 Related Research **8**
- 2.1 Ada Environments 8
  - 2.1.1 ATMAda 8
  - 2.1.2 Arcturus 9
  - 2.1.3 CAEDE 11
- 2.2 Ada-compatible Design Methods 13
  - 2.2.1 METHODMAN 13
  - 2.2.2 The U.K. Study 14
  - 2.2.3 CORE/MASCOT 17
  - 2.2.4 JSD 18
  - 2.2.5 CCS 19
- 2.3 Discussion 19

3 STONEMAN - Requirements for APSEs **21**
- 3.1 Components of an APSE 21
- 3.2 The Layered Approach 22
  - 3.2.1 The KAPSE 24
  - 3.2.2 The MAPSE 24
  - 3.2.3 The APSE 25
- 3.3 APSEs and design methods 26

4 SARA/IDEAS – A Computer-Based System Design Method **28**
- 4.1 The Structure Model 30
- 4.2 The Graph Model of Behavior 32
  - 4.2.1 The Control Domain 32
  - 4.2.2 The Data Domain 35
  - 4.2.3 The Interpretation Domain 38
- 4.3 The Building Block Library 38
- 4.4 The Socket Attribute Model 39
- 4.5 The Module Interconnect Description 40

4.6 The SARA/IDEAS Environment 40

5 Stoneman versus the SARA/IDEAS Requirements 42

5.1 Similarities 42

5.1.1 General Guidelines 42

5.1.2 Requirements for APSEs 44

5.1.3 APSE Control Requirements 44

5.1.4 APSE Toolset Requirements 45

5.2 Differences 46

5.2.1 APSE Toolset Requirements 46

6 GMB models for Ada Tasking Primitives 48

6.1 Ada Tasking Primitives 50

6.1.1 Non-interacting tasks 50

6.1.2 Task synchronization - Rendezvous 51

6.1.3 The select statement 53

6.2 Discussion 61

7 Translating GMB models into Ada program skeletons 62

7.1 Translation Schemes 63

7.1.1 Work by Nelson 63

7.1.2 The P-NUT System 64

7.2 Designing the Translator 65

7.2.1 Control Nodes and Arcs 65

7.2.2 The Task Scheduler 66

7.2.3 Interface to the Interpretation Domain 67

7.2.4 Ada code for control nodes 67

7.3 Terminating the loops 68

7.3.1 Detection by the Task Scheduler 69

7.4 An Example 69

7.5 The Translation Algorithm 75

8 Ada as the Interpretation Domain Language for SARA 77

8.1 The Interpretation Domain in SARA 77

8.2 The Interface Problem 79

8.3 Data Structures Communication 79

8.4 Designing the Interface 80

8.5 An Ada Package Specification 81

8.6 The T side of the Interface 82

8.7 An Example 83

9 Conclusions and Future Research **85**
9.1 Contributions 85
9.2 Usage Scenario 86
9.3 Limitations 87
9.4 Areas of Future Research 89
9.4.1 The GMB-to-Ada translator 89
9.4.2 An Ada-to-GMB translator 91

A User's Guide to the Translator **92**
A.1 User's Guide 92

B List of Acronyms **94**

References **96**

LIST OF FIGURES

3.1 Architecture of an APSE 23

4.1 Structure Model of the Buffer System 31
4.2 GMB Control Graph for the Buffer System 34
4.3 GMB Data Graph for the Buffer System 36

6.1 GMB model for Non-interacting Tasks 51
6.2 GMB model for simple rendezvous 52
6.3 GMB model for compounded rendezvous 53
6.4 GMB model for the plain select statement 54
6.5 GMB model for guarding conditions 56
6.6 GMB model for delay alternatives 57
6.7 GMB model for timed-out entry calls 59

7.1 A simple GMB 69

9.1 A simple GMB 90

ABSTRACT OF THE DISSERTATION

System Architect's Apprentice (SARA) as the
Foundation for a Methodology-Oriented
Ada® Programming Support Environment

by

Eduardo Aaron Krell
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1987
Professor Daniel M. Berry, Chair

The design of inherently complex concurrent systems is inhibited by supportive languages and design environments. This dissertation probes the possibilities of integrating the strengths of a design method supported by a programming system called SARA (System ARchitect's Apprentice) and the well known language, Ada®.

It has been recognized that one of the weaknesses of Ada Programming Support Environments (APSEs) is the lack of integration of design methods into them.

It is shown that SARA and Ada can be integrated and, together, provide a significant advance over the state of the art.

[0]® Ada is a registered trademark of the U.S. Department of Defense (AJPO).

CHAPTER 1

Introduction

1.1 Motivation for the Research

Ada® has been chosen by the U.S. Department of Defense as their new Common High Level programming language for embedded systems. The definition of Ada Programming Support Environments (APSEs) came in a document known as Stoneman [Buxt80]. This document gives the requirements for APSEs and develops a model for them. It also provides a framework to accommodate a wide variety of design methods and software tools.

Although Stoneman does not enforce any particular design method to be used by APSEs, it states in 2.B.16

> ... A comprehensive APSE may encourage, or even enforce, one specific system development methodology [sic] [1].

The fact that APSEs are not built around any particular design method is being seen today as a major drawback. John Barnes (a member of the original Ada design team) recently said in a conference:

® Ada is a registered trademark of the U.S. Department of Defense (AJPO).

[1] People often use "methodology" when they should use "method" instead. Methodology is the study of methods or a system of methods. I will use "method" but when quoting other people, the word "methodology" will appear when it does in the original.

> "The language itself is ready (referring to the ANSI standard Ada). The compilers are almost ready (there are already several in the market). The training is established (Ada courses). The APSEs are in need of urgent action".

John Buxton himself, the creator of Stoneman, when criticizing his own work, said that "not enough was done on using a methodology as a driver for an APSE design". He also said that the environment development should be separated from Ada, introducing the idea of multi-language environments, which could be tailored then to support a specific programming language. He strongly suggests that APSEs should be method-specific instead of language-specific.

John Buxton and the rest of the team who wrote Stoneman are not to be blamed for doing a poor job since the state-of-the-art in requirements for design environments was at that time (and still is) a long way from being as precise as, say, requirements for programming languages.

Nevertheless, the need for design methods to be part of Ada Environments is real, as discussed in [Fisc84] and [Druf82]. Quoting from the latter:

> "... we will not realize the full potential of Ada until we are able to define a software development methodology complete with management practices which can in turn be supported by automated tools."

SARA (System Architect's Apprentice) is a requirements-driven design method for concurrent digital systems [Camp78,Estr78]. IDEAS (Intelligent Design Environment for Analyzable Systems) is a set of tools supporting the SARA method in an interactive, workstation environment. The SARA method provides modeling in three domains: Control, Data, and Interpretation. SARA does not enforce the use of any particular programming language for its Interpretation Domain.

A team of British experts conducted a study of several design methods (including SARA) to see how well would they support software design in Ada. Their results were published in [Jack81]. The report shows an Ada design modeled using SARA, where Ada specifies the behavior of the model. The authors state:

> "It is assumed that the module packages will be processed by a special SARA processor, which may include the Ada compiler or which may be a pre-processor to it. This processor will construct appropriate simulators and analysers for the system, or the data structures necessary to drive general purpose tools. Such a processor does not currently exist, the existing SARA support system being oriented towards PL/I. However, such a tool would form an essential part of a SARA-based APSE."

As of today, none of the existing APSEs is driven by a specific design method. They are just a collection of general purpose tools, not tied to any particular method.

It is true that a design method could be later on adapted to an APSE since there are many tools in an APSE that can be used to support many different methods. However, I believe the tools of an APSE should be there to support the method, not the other way around. Therefore, it is essential to have an APSE *driven* by a design method.

One problem with method-oriented APSEs is that there is no commonly accepted underlying model to support full life cycle development, as discussed in [McDe84]. The Ada community has been trying to define the requirements for an Ada-compatible method for some time. The first draft of this document, known as METHODMAN I [Wass82], surveys several design methods and analyzes their characteristics such as life-cycle coverage, suitable application areas, technical concepts supported and Ada compatibility.

SARA is found, in this study, to cover the entire life cycle from requirements analysis to implementation and validation, and well suited for embedded system applications. In the Ada compatibility study, SARA is found to be Ada-compatible to a great extent.

1.2 Related Research Areas

Very little has been done in the Ada environments area (which is one of the motivations for this research). There are some experimental Ada environments in development such as ARCTURUS [Will84,Will83] (from the University of California at Irvine). The U.S. Army is also building an Ada Environment together with their Ada compiler in what is known as the ALS (Ada Language System), but it is not expected to be completed until 1987.

These Ada environment are nothing more than a collection of software tools to design, write and debug Ada programs. Among these tools are:

- Ada compilers and/or interpreters
- Syntax-directed Ada editors
- Ada cross-reference generators
- Debuggers and debugging tools

Once again, the problem is that these environments do not address the issue of an Ada Design Method, which I believe should be the heart of an Ada environment.

1.3 Research Hypothesis

The primary hypothesis of this dissertation is that SARA can and should be used as the design method for an Ada Programming Support Environment. In order to test this hypothesis, it is necessary to test this list of secondary hypotheses:

- That SARA/IDEAS and APSEs have similar requirements
- That SARA tools be able to model concurrency in Ada
- That Ada can be used as the Interpretation Domain language in SARA
- That SARA models can be translated into skeleton Ada programs

1.4 Research Plan

I will carry the following research activities in order to prove the research hypotheses listed above:

- Review Literature on APSEs and try to use at least one of them.
- Show similarities between the requirements for APSEs as indicated in Stoneman and the SARA/IDEAS requirements.
- Show that the Ada tasking primitives can be modeled using SARA.
- Design and implement a GMB-to-Ada translator to build Ada program skeletons from GMB models.
- Show the feasibility of using Ada as an Interpretation Language for SARA by building an appropriate interface.

1.5 Dissertation Organization

Chapters one through four provide the reader with an introduction to the problem area, a review of related research and products, an analysis of Stoneman and a description of the SARA method and its support environment.

Chapter five compares the SARA requirements with those in Stoneman. Chapters six through eight are the heart of the dissertation. Chapter six gives GMB models for all of Ada Tasking Primitives. Chapter seven shows how GMB models can be translated into Ada code skeletons. Chapter eight show how Ada can be used as the Interpretation Domain Language for SARA.

Chapter nine lays out the contributions, future research suggestions and conclusions.

CHAPTER 2

Related Research

This chapter surveys existing Ada Environments, or prototypes thereof, as well as attempts made to define Ada-compatible Design Methods.

2.1 Ada Environments

2.1.1 ATMAda

The ATMAda Environment is an enhanced version of the Rolm/Data General Ada Development Environment (ADE) from Texas A & M University [Matt86]. The ATMAda Environment comprises seven major components:

- *The Ada Editor.* There are two full-screen editors, one of which has been modified for use with Ada by supporting the use of Ada source code templates and access to the on-line reference materials.

- *A Help Facility.* A help facility provides consistent help on commands, utilities, and provides commentaries on various aspects of the environment and its features.

- *An Environment Customization Facility.* The user can customize simple things such as terminal default colors and prompt, pretty printing defaults, custom templates, etc.

- *An On-line Ada Syntax Reference Guide.* A synopsis for the syntax of most Ada constructs is provided. Each synopsis has a BNF description, examples of use, and references to the Ada Language Reference Manual (LRM).
- *An On-line Reference Manual.* The Ada LRM is included in the environment.
- *An Automatic Recompilation Facility.* This facility keeps track of module dependencies for each program unit so that when each module is recompiled, all dependent modules which need to be recompiled will be according to the compilation sequence specified by the Ada LRM.
- *A Library Management Facility.* This is only a proposed facility for the future as no library management functions have been implemented at the time of this writing.

The ATMAda developers recognize that "one problem with developing large and complex software is the lack of coherent development methodologies [sic] and software management tools". They view this problem as one to be addressed in the ATMAda Environment "in the future".

2.1.2 Arcturus

Arcturus [Will84,Will83] is a prototype of an APSE integrating together tools for system design, coding, testing, and maintenance for *programming-in-the-large.*

Users interact with Adash (Ada Shell), a text editor with syntactic and semantic knowledge about Ada. Adash sends characters to the lexical analyzer for Arcturus and receives characters from the tools in Arcturus. The lexical analyzer was written using LEX [Lesk75] and interfaces directly to the parser, which was written using YACC [John75]. The parser generates abstract syntax trees, members of the DIANA [Goos83] family of trees.

The heart of Arcturus is an interactive Ada interpreter. The user can write expressions using Ada syntax and the interpreter will print back the result of evaluating that expression. The user can also enter Ada statements that will get executed immediately. Entire functions, procedures, packages and programs can be typed or read from files and then later on executed. There is a pretty-printer to properly indent pieces of programs when printed and there is also a user-callable procedure called *break* which will produce a breakpoint at run time. Users can then get a backtrace of the last statements executed and continue execution if they wish to do so.

Arcturus supports the implementation and partially the maintenance phases of the life-cycle model but there is no support for the earlier phases of requirements specification and design.

Arcturus does not include a design method as a driver for the environment. In addition, the tools in Arcturus are language-specific (that is, Ada-specific) rather that method-specific, exactly the opposite of what I am looking for. I therefore believe that Arcturus is not suitable for my purposes.

2.1.3 CAEDE

CAEDE (CArleton Embedded system Design Environment) is an experimental design environment for embedded systems being built at Carleton University [Buhr85d,Buhr85b,Buhr85c,Buhr85a]. It is based on an extended version of Buhr's graphical design method [Buhr84]. CAEDE's intended application area is embedded system design. It claims to be a design environment by integrating a design method, a design entry system (graphical paradigms), a design data base and design toolsets. The design data base and most of the tools are written in Prolog.

Graphics in CAEDE are used to capture the logical structure of the system under design in a Prolog data base and the generation from this data base of partial Ada source code for the system under designed.

Underlying Principles

CAEDE is founded on the following principles:

1. METHODOLOGY: The design of embedded systems is a creative process and the method's and the environment's tasks are to assist this process.

2. GRAPHICS: Graphical paradigms provide a framework for reasoning about a system.

3. CLOSENESS TO PROGRAMMING: The design level should be close enough to Ada so that mappings between design and Ada code are straightforward in either direction.

4. POSTPONEMENT OF COMMITMENT: The tools should be usable without full software implementation of a design.

5. DESIGN SUPPORT: Design tools should support the system structure, temporal behavior and functional behavior aspects of the system.

Since the system is written in Prolog, it is relatively simple to translate designs (which are stored in Prolog's data base) into partial Ada source code using Prolog's translation rules. The system is implemented on a SUN workstation under 4.2 Unix using SUNCORE graphics, C and C-Prolog.

The modeling primitives in CAEDE are very close to those of Ada. In fact, they map one-to-one with Ada's features. The translation of CAEDE's models into Ada code is, thus, an almost trivial one since they are two different representation of the same concepts.

Also, the complexity and size of Ada manifest themselves in CAEDE: there are more than 15 different modeling primitives with more in sight. This makes modeling a difficult task. The designer is presented with too many choices and choosing the right one is not easy.

Furthermore, the closeness to Ada implies that the designer has to think in terms of Ada during the design process. This is particularly annoying in the concurrency area, where the model of concurrency in Ada is a controversial one. One would rather design in terms of higher level concepts, which makes a model simpler and smaller. These higher level models could then be translated into a given target language like Ada.

2.2 Ada-compatible Design Methods

This section will present the purpose and main results of two major efforts to define Ada-compatible design methods: METHODMAN and the *Report on the Study of An Ada Based System Development Methodology* (U.K. Study, for short). In addition, the main characteristics of the Ada-compatible methods found by the latter study are shown here.

2.2.1 METHODMAN

METHODMAN I [Wass82] was the result of an effort by the Ada community to define the requirements for an Ada-compatible design method. An extensive survey of some 48 methods in use was done, to see to what extent they were (or were not) Ada-compatible.

The methods were evaluated in four main categories:

technical: support for hierarchical models, modularization and interface definition among modules, support for control flow, data flow, data abstraction, procedural abstraction, parallelism, safety, reliability and correctness.

usage: how easy is to understand the results, to teach the method to someone else, reusability of previous designs, are there automated tools that support the entire method (or phases thereof)?, What range of the life-cycle is covered by the method?, application area, extent of current usage, ease of transition from one phase in the life-cycle model to the next one, validation of models, repeatability.

management: use of standard management techniques, support for teamwork, configuration management, project scheduling, cost estimation.

economic: benefits of using the method versus costs (of acquisition, of use, of management).

Twenty seven organizations (representing 27 different methods) answered the long questionnaires and the results were analyzed and tabulated. One of the tables, shown in the next page, represents the degree of Ada compatibility found in each method.

Of special interest to us is the line for SARA. It says that SARA has no serious incompatibility with Ada. Furthermore, it can be used to model Ada Packages, Tasks, Generics and Exception Handling. The only "negative" entry is in the machine representation section, where SARA does not currently support explicit machine representation schemes, but they can be realised by whatever language is used in the Interpretation Domain.

2.2.2 The U.K. Study

The Report on the Study of An Ada Based System Methodology [Jack81] was the main result of a joint effort between the U.K. Department of Industry and a

ADA COMPATIBILITY						
Methodology	Ada Construction				Machine Represen-tation	Serious Incompat-ibility?
	Packages	Tasks	Generics	Exception Handling		
ACM/PCM	yes	no	yes	yes	no	Database%
DADES	?	?	?	?	?	?
DSSAD	...	...	...	...	...	...
DSSD	yes	yes	?	yes	?	?
EDM	?	?	?	?	?	?
GEIS	no	no	no	no	no	...
HOS	yes	yes	yes	yes	yes	no
IBMFSD-SEP	yes	?	yes	yes	yes	...
IESM	yes	yes	yes	yes	no	?
ISAC	yes	yes	...	no	yes	no
JSD	yes	?	no	?	?	?
MERISE	yes	yes	yes	yes	no	...
NIAM	yes	yes	?	?	?	...
PRADOS	...	...	...	...	...	...
REMORA	yes	yes	yes	possible	no	no
SADT	yes	yes	yes	yes	yes	no
SARA	yes	yes	yes	yes	no	...
SA-SD	yes	yes	...	yes	...	Database&
SD	yes	yes	yes	...	...	...
SDM	yes	yes	...	yes	yes	...
SEPN	no	no	no	no	no	yes
SREM	yes	yes	yes	yes	no	yes
STRADIS	...	...	...	...	...	?
USE	yes	yes	no	yes	no	Database&

Key:

?	=	"Not known"
...	=	no answer
yes	=	methodology supports mapping into Ada feature
no	=	methodology does not support mapping into Ada feature
%	=	Query Language
&	=	Modeling/Design; I/O

number of british companies.

Their goal was to first conduct an extensive review of as much published material as possible in the area of design methods. They realized that published material rarely contains the full story, and members of this team visited 24 organizations where they discussed further details with both implementors and users of those methods.

A set of thirteen different characteristics was defined to assess each method against. These characteristics can be divided into five major categories:

- life cycle coverage
- technical characteristics
- management and control characteristics
- usability
- Ada compatibility

This study finds four potential Ada-compatible methods among the over 40 methods originally studied. These are CORE/MASCOT, JSD, SARA and CCS. Since SARA is covered in a separate chapter, I shall give an overview of only the other three methods.

2.2.3 CORE/MASCOT

Controlled Requirements Expression (CORE) and MASCOT are two different methods often used in conjunction to cover the entire life cycle, CORE dealing with the requirements and specification phases and MASCOT with the design and implementation phases.

2.2.3.1 CORE

CORE is concerned with the collection and collation of information which defines the requirements of a system. The specification and design procedure within the CORE system is concerned with the analysis of data flow through a system. Initially, data flow is viewed from a number of different viewpoints in an attempt to identify logically independent, isolated processing paths. The various viewpoints are then combined to form a complete system representation.

A standard questionnaire is used to gather information on requirements from several viewpoints. A standard procedure then provides for the identification of data flow through the system and a top-down decomposition of the requirement is imposed such that sub-requirements can be identified and handled separately if needed. Consistency of data flow within and across levels is a prime concern of the method.

The method is mainly of use for large, complex systems and the Ada Tasking model may require changes to the CORE method.

2.2.3.2 MASCOT

MASCOT is based on the analysis of data flow to perform a structural decomposition of the problem domain. The aim is to identify co-operating parallel processes which have a minimum of communication with each other.

The essence of MASCOT is a graphical notation and a set of building blocks for expressing real-time system designs. MASCOT applies to the design, implementation and maintenance phases of the life cycle. Interfaces to a specific high-level language (CORAL) have been defined.

The principal building blocks in MASCOT are Activities and Intercommunication Data Areas (IDAs). IDAs in turn consist of Channels and Pools. There is a graphical representation of these building blocks and they are connected by lines in a diagram, showing communications among different activities in a system.

There are rules for connecting Channels, Pools and Activities. Problem decomposition is achieved by hierarchically dividing a system into communicating subsystems.

Since there is an intimate interface between MASCOT features and the underlying operating system and high-level implementation language tools (like compilers and linkers), porting MASCOT to a new host or target is a non-trivial task.

2.2.4 JSD

The Jackson System Design (JSD) method is aimed at the system development level, from somewhere in the requirements phase through to implementation

[Jack83,Hugh79,Came83].

JSD should not be confused with JSP (Jackson Structured Programming). The latter is a program design method developed between 1972 and 1974.

JSD is a superset of JSP covering almost the whole life cycle, beginning with a model of the entities in the problem domain and ending with program implementation and subsequent maintenance. The JSD phases are:

1. modeling the real-world entities.

2. extending the model to include the functions of the system.

3. implementing the model as a program.

2.2.5 CCS

The Calculus of Communicating Systems (CCS) is an algebraic way of describing the meaning of a concurrent system. A system is viewed as being constructed from a number of process objects that have externally visible ports which are used to connect processes together.

2.3 Discussion

The first two APSEs presented in this chapter, ATMAda and Arcturus, are not suitable for my purposes since they do not allow for any design method to be part of the APSE. A particular design method could be later on added to these APSEs but it should be done the other way around: you have the method first and only then the environment is built to support the method.

The third APSE, CAEDE, seems at a first glance as a promising candidate but when looked at closely reveals that it has a serious drawback by lacking support for bottom-up composition of designs. In addition, the method relies too heavily on Ada being the underlying implementation language and does not provide higher level abstractions. This is good because the mapping between designs and Ada code is straightforward but it locks the designer into thinking in Ada terms early on.

The Ada-compatible methods suffer from a series of different problems that make them unsuitable for my purposes. It is not clear how CORE would support Ada tasking, and one of my goals is to deal with Ada tasking.

CCS provides a formal way of describing concurrency but does not have a support environment. It is a formal algebraic notation rather than a design method.

CHAPTER 3

STONEMAN - Requirements for APSEs

The U.S. Department of Defense Common High Order Language program started in 1975. Its goal was to establish a single high level language for new embedded computer systems. Ada was the culmination of that effort.

It was recognized from the beginning that the major benefits from the use of Ada would be, among other things, from its use as a mechanism for distributing effective software development and support environments.

Stoneman was the nickname given to the document named *Requirements for Ada Programming Support Environments*, published by the Department of Defense in early 1980. It paints a broad picture of the needs and identifies the relationships of the parts of an integrated Ada Programming Support Environment (APSE).

Stoneman specifies the requirements for an APSE, provides criteria for assessment and evaluation of APSE designs, and offers guidance for APSE designers and implementors.

3.1 Components of an APSE

An APSE, as defined by Stoneman, has three principal components:

The Data Base: acts as the central repository for all information associated with a project throughout its life cycle.

The Interface: includes both the interface to the user and to the data base and toolset.

The Toolset: tools for program development, maintenance and configuration control.

3.2 The Layered Approach

To address the problem of portability, Stoneman proposes a layered architecture for APSEs:

level 0: Hardware and host software.

level 1: Kernel APSE (KAPSE); provides database, communication and run-time support functions. Presents a machine-independent portable interface.

level 2: Minimal APSE (MAPSE); minimal set of tools, written in Ada, necessary and sufficient for the development and continuing support of Ada programs.

level 3: A full APSE, extending a MAPSE to provide support for a particular application or method.

This layered architecture is represented in Figure 3.2.

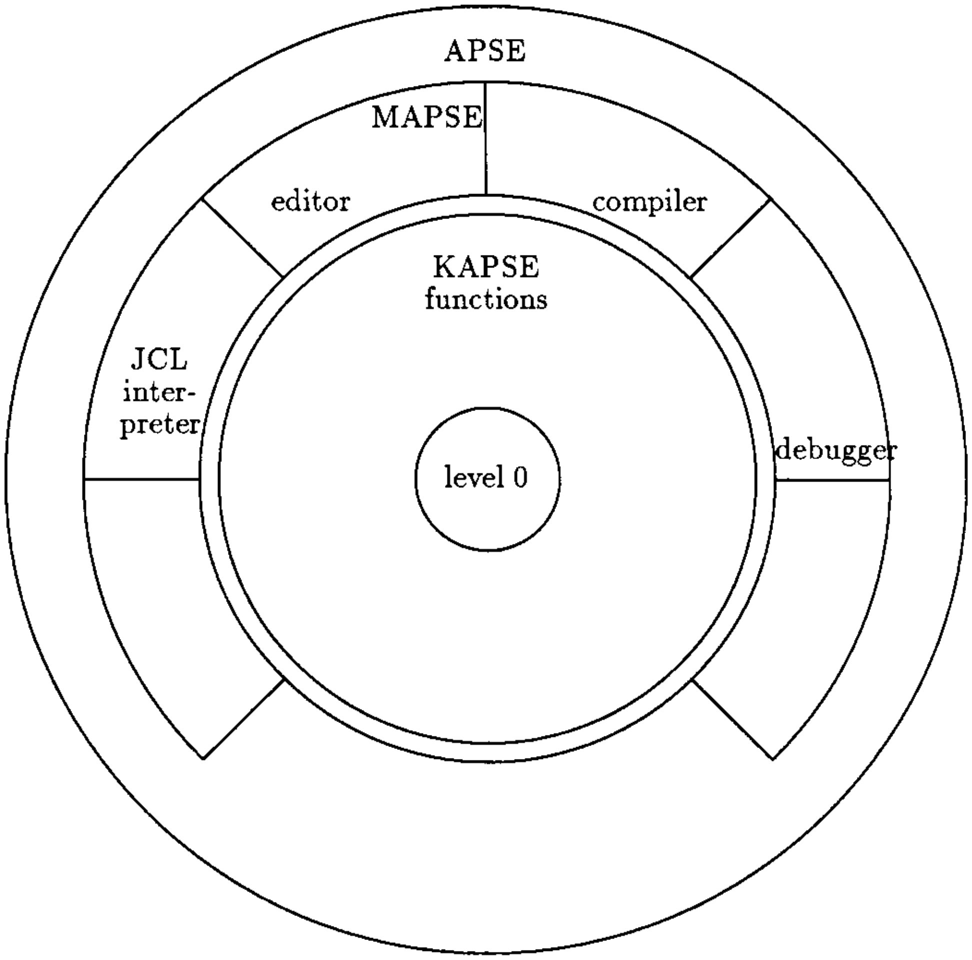

Figure 3.1: Architecture of an APSE

3.2.1 The KAPSE

The KAPSE provides the basic run-time support facilities that are required by Ada programs within an APSE. A typical KAPSE will provide all Input/Output services as well as the tool invocation mechanism, including suspension, resumption, abortion, and termination of programs. A KAPSE will also provide the interface with the underlying database manager.

The features of a KAPSE are given in one or more Ada package specifications. These declarations include operations that are made available to any tool in an APSE as well as abstract data types common to all APSEs (including abstract data types used in the interface specification for the various phases of program compilation and execution).

The KAPSE defines, therefore, an abstract data type for a virtual support environment for Ada programs, i.e., a virtual machine, by hiding all the peculiarities of the underlying operating system and providing a uniform interface to an Ada program, making it easy to port Ada programs across different APSEs.

3.2.2 The MAPSE

A MAPSE provides a minimal toolset written in Ada and supported by the KAPSE which both necessary and sufficient for the development and support of Ada programs.

Tools within a MAPSE or an APSE are to be written in Ada and can, therefore, use (in the Ada sense) the interface packages provided by the KAPSE. That is, these tools can make use of the abstract data types and object declarations provided by the Ada packages used by the KAPSE to specify the interface with the low level facilities.

Requirements for MAPSE tools include

1. a standard text editor,

2. a source code prettyprinter,

3. an Ada compiler (plus linker and loader),

4. a set-use static analyzer (for cross reference),

5. a control flow static analyzer,

6. a dynamic analysis tool,

7. a file administrator,

8. a command interpreter, and

9. a configuration manager

3.2.3 The APSE

An APSE includes tools that go beyond the MAPSE requirements. Among them are:

1. syntax-directed editors,
2. documentation tools,
3. project control systems,
4. configuration control systems,
5. measurement and performance tools,
6. fault report systems,
7. requirement specifications,
8. system and program design tools, and
9. program verification tools.

3.3 APSEs and design methods

Because of the layered architecture, KAPSEs deal only with issues pertaining to low-level, machine-independent user interface to an APSE. The only support provided by a MAPSE for different activities throughout the life cycle of a project consists of general text manipulation facilities (text editors, prettyprinters, etc).

As stated by Stoneman in section 2.B.16,

> Clearly, the MAPSE does not emphasize any particular development methodology at the expense of any other. However a comprehensive APSE may encourage, or even enforce, one specific development methodology.

From this it is clear that the place where a design method comes into the picture is in a full blown APSE. Stoneman does not go one step further, defining a method that could be used in an APSE. This is a gap that needs to be closed and this dissertation tries to do just that.

CHAPTER 4

SARA/IDEAS – A Computer-Based System Design Method

SARA (System Architect's Apprentice) is a requirement-driven top-down and bottom-up design method for concurrent digital systems [Camp78,Estr78]. The method provides for a separate modeling of the structure and the behavior of the system being designed. The history of SARA can be found in [Estr86].

I will carry out an example throughout this chapter to illustrate the different modeling aspects attacked by SARA. The example chosen is a simple input/output buffer system. The specification is given by the following Ada Package:

```
package buffer_package is
   type message_slot is private;
   type buffer is array(1..MAX) of message_slot;
   procedure write(m : in message_slot);
   function read returns message_slot;
   procedure init;
end buffer_package;
```

A buffer is, then, a sequence of `MAX` `message_slots`. The `init` procedure initializes the buffer to be empty. Calls to procedures `read` and `write` can occur concurrently. If `write` is called when the buffer is full, the caller will be suspended until there is an empty slot in the buffer. If `read` is called when the buffer is empty, the caller will be suspended until some message is written onto the buffer. The system needs to manage the buffer in such a way to ensure that:

1. messages are delivered in the same order as they are received,
2. no message is destroyed (written over) before being read, and
3. no message is read twice.

Environment Assumptions

The environment will contain two processes, `sender` and `receiver`. The `sender` process behaves as follows:

1. it sends messages to the buffer system through the `write` procedure,
2. it sends only one message at a time and, after sending one message, can proceed only after `write` finishes, and
3. after the last message is sent, it terminates.

The `receiver` process behaves as follows:

1. it requests messages from the buffer system through the `read` procedure,
2. it reads one message at a time and, after requesting a message, can proceed only after `read` finishes, and
3. after the last message is read, it terminates.

4.1 The Structure Model

The structure of a system is expressed in terms of the Structure Model (SM). The SM has three primitives:

- modules,
- sockets, and
- interconnections

Modules can be connected with other modules by an interconnection connecting two sockets, one socket in one module and one socket in the other module. Thus, sockets are communication ports for modules.

The interconnection is not directed, so it just models a communication line but does not reveal which way the information flows. Nothing is said about the type of data passing through that interconnection; it could be either data or control or both. An interconnection always connects two and only two sockets. Furthermore, a socket can have only two interconnections attached to it: one going out and one coming in.

There is a top level module called `universe` which has no sockets. Hierarchical decomposition is achieved by refining a module into submodules. This process can be repeated until the system has been decomposed into small enough modules, whose behavior can be directly mapped to an existing behavioral model stored in the Building Block Library or whose behavior is simple enough to be understood and expressed using the behavioral primitives.

In our buffer system, we would decompose our universe module into the buffer system and its environment. The environment and the buffer system would communicate through the `write` and `read` operations. Figure 4.1 shows the SM for the Buffer System.

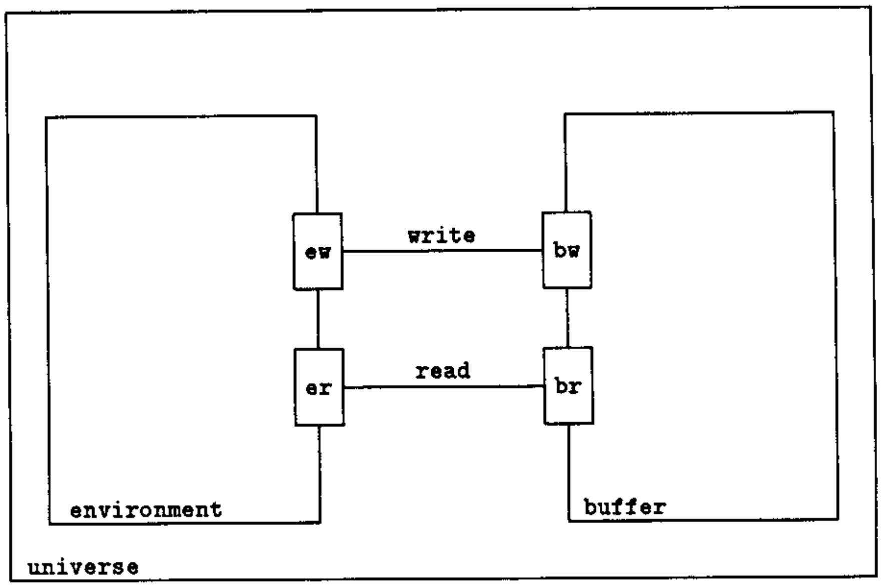

Figure 4.1: Structure Model of the Buffer System

The `environment` module has two sockets, `ew` (for environment write) and `er` (for environment read). These sockets are connected through interconnections `write` and `read` respectively to sockets `bw` and `br` in the `environment` module.

The `environment` module or the `buffer` module could be partitioned further into submodules if needed.

4.2 The Graph Model of Behavior

In SARA, the behavior of the system is modeled using the Graph Model of Behavior (GMB) [Razo80]. The GMB offers the designer three different but related modeling domains, control, data, and interpretation. The designer can, then, think independently of these three aspects of his system and describe the system using these three different views that will need to be made consistent with each other.

4.2.1 The Control Domain

The control flow model describes concurrency, synchronization and precedence relations in a graph using an underlying theoretical model similar to Petri-Nets [Pete81].

The control domain of the GMB is a directed hypergraph, i.e., a graph in which the edges may have multiple sources and/or multiple destinations. *Control nodes* (the vertices) represent events and *control arcs* represent precedence constraints, or a partial ordering, among the events.

Each node has an input logic expression, which is a boolean expression on the input arcs, that expresses the condition under which that node can be initiated. An OR (+) in the input logic means any of the operand arcs can initiate the node. An AND (*) in the input logic means that all operand arcs must pass control before that node can be initiated.

Each node has an output logic expression, a boolean expression on the output arcs, which shows where is control passed upon termination of that node. An OR here implies control is passed to one of the designated arcs. An AND implies control is passed to all the designated arcs.

Both input and output logic expressions can be arbitrary functions using ANDs and ORs. Control Flow in the control graph is represented by the passing of *tokens* through control arcs. When a node is initiated, it consumes the tokens which enabled it. Upon node termination, tokens are created and placed on output arcs according to the node's output logic expression. The semantics of a control graph are dictated by an underlying machinery known as the *token machine* which performs state-to-state transformations on the graph, starting from an *initial token distribution* and terminating if and when no further transformations are possible.

Continuing with our buffer system, we would define a control graph for each of the modules defined in the SM. The mapping between the control graph and the SM is shown by drawing the control graph on top of its corresponding SM. Figure 4.2 shows the control graph for the buffer system.

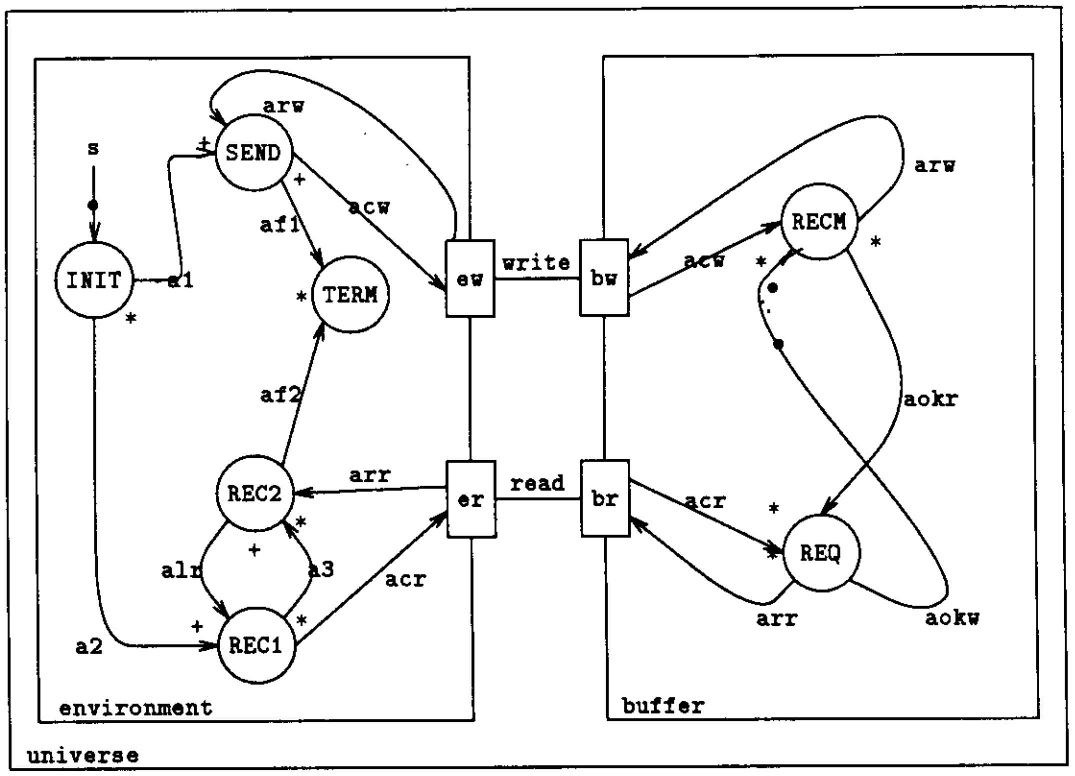

Figure 4.2: GMB Control Graph for the Buffer System

The following tables describe what the function of the various components in the control graph are:

Control Nodes

INIT	initiation process, initiates sender
TERM	termination process
SEND	sender process; sends message to the buffer and receives acknowledgment
REC1	receiver process I, requests for message

`REC2`	receiver process II, actually receives messages from the buffer and informs `REC1`
`RECM`	receives message from the environment, acknowledges, and performs the **`write`** operation
`REQ`	receives request, performs the **`read`** operation, and sends it to the **`environment`**

Control Arcs

`aokw`	semaphore, indicates the number of empty slots in the buffer
`aokr`	semaphore, indicates the number of messages in the buffer

4.2.2 The Data Domain

The data domain of the GMB is a bipartite directed graph, i.e., a graph in which there are two types of nodes (*datasets*, represented as rectangles and *data processors*, represented as hexagons) and in which arcs (called *data arcs*) are used to connect datasets with data processors. Thus, every data arc goes from a data processor to a dataset or viceversa. This graph represents the data flow of the system by defining its data paths.

Datasets model static collections of data. Data processors are data transformers which can read from and/or write to datasets. Data arcs define the read and write accesses of a data processor to a dataset.

Continuing with our buffer example, we would draw the data graph over the SM and show the mapping existing between the data and control graph.

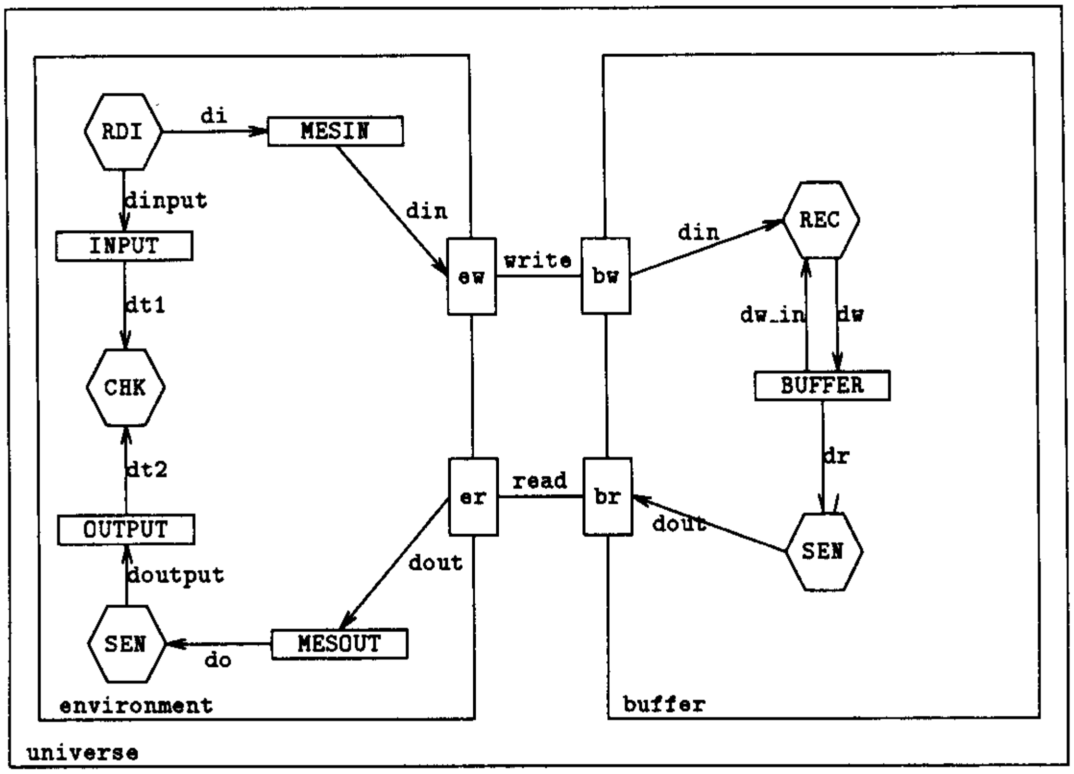

Figure 4.3: GMB Data Graph for the Buffer System

The following table describes the function of the various data processors and datasets in the data graph:

Data Processors

CHK	mapped to control node TERM; it checks the message received against the initial messages to determine that they are the same and in the same order before termination
RDI	mapped to SEND in the control graph; it reads messages from INPUT into MESIN, one at a time
RDO	mapped to REC2 in the control graph; it reads messages from MESOUT, and deposits them into OUTPUT, one at a time
REC	mapped to RECM in the control graph; it receives messages from the **environment** and deposits them into dataset BUFFER.
SEN	mapped to REQ in the control graph; it reads messages from BUFFER and passes them back to the environment

Datasets

INPUT	initial sequence of messages
OUTPUT	messages read from BUFFER, to be checked against INPUT
MESIN,	
MESOUT	message slots, interfaced with submodule **buffer**
BUFFER	the actual buffer in the system

4.2.3 The Interpretation Domain

The Interpretation Domain defines the format of the data stored in datasets and defines the transformations of data performed by the data processors. Many interpretation languages can be used for this domain. The original SARA system used PLIP (an extension of PL/1) as its interpretation language. The current system, being implemented in a Lisp dialect supports a Lisp-like interpretation language.

One of the proposals of this dissertation is that Ada be an interpretation language for SARA.

4.3 The Building Block Library

In order to support bottom-up design, it is necessary to have a collection of previously designed and tested models, appropriate for the design domain, stored in a design database. The SARA design database is called the Building Block Library by Drobman in [Drob80]. His work concentrates on hardware building blocks but the procedure is also applicable to software building blocks.

The primary hypothesis of Drobman's work is "a set of models of hardware and software building blocks can be created and utilized as primitive elements in a computer-aided design system and methodology such that the composition of requirement-satisfying, partially correct, microprocessor-based digital systems is dramatically enhanced". He demonstrated satisfaction of the hypothesis by defining building block descriptions of a number of hardware devices, and then using those building blocks to design a 16-bit microprogrammable microprocessor.

Other SARA researchers have studied the requirements and organization of a design library [Land83,Land86].

4.4 The Socket Attribute Model

During research on the Building Block Library and the SARA simulation tools, it was felt that many of the errors found during simulation could have been detected much earlier by analysis of some as-of-yet undefined static description of the building blocks. This observation spawned Sampaio's research into the Socket Attribute Model, SAM [Samp81], and Penedo's research into the Module Interconnect Description, MID [Pene81]. While both deal with a description of a building block at its interfaces, sockets or interconnects, SAM concentrates on hardware building blocks and MID concentrates on software building blocks.

Sampaio provides a language to describe the behavioral attributes of a hardware module's sockets, for example, electrical characteristics (fan-in, fan-out), timing (set-up and hold times), bandwidth, and perhaps physical characteristics. With these descriptions attached to a module's sockets, it is possible to detect inconsistencies occurring during composition of two or mode modules. The detection of socket mismatch errors occurs at the time the socket connection is attempted, not later during an expensive and time consuming simulation that may not detect the error at all.

4.5 The Module Interconnect Description

Penedo attacks the same problem as Sampaio, but on the software front. She describes software modules as they appear at their interfaces. Most type checking compilers detect some of the errors that Penedo is after, for example, procedures called with the wrong number or type of arguments. The product of her research is the Module Interconnect Description, MID [Pene81,Pene79]. Berry shows later in [Berr84] that Ada package specifications meet the needs of Penedo's MID.

4.6 The SARA/IDEAS Environment

The SARA method is supported by automated tools in an integrated interactive environment. These tools allow the user to create and edit SM and GMB models. There is also a design data base where models can be stored and retrieved from.

The GMB simulator [Razo79] is one of the main tools in the SARA environment. It conducts a simulation of the GMB model, allowing the user to set up breakpoints in the control graph. Extensions have been made to the GMB and to the simulator to model queueing systems for performance analysis. Another tool is the Control Flow Analyzer, used to analyze control graphs for control flow anomalies such as deadlocks and to assure some control flow properties such as proper termination.

A second generation of these tools is being implemented at UCLA on the Apollo workstations. Extensive efforts are being made into providing a state-of-the-art graphics interface and user interface. More details on this can be found in [Krel85a].

CHAPTER 5

Stoneman versus the SARA/IDEAS Requirements

It should be of interest to see similarities (and differences) between the requirements for APSEs as stated in Stoneman and the SARA/IDEAS requirements. When building an APSE by adapting an existing design environment, one needs to keep always in mind these similarities and differences.

The similarities help to reassure one that the underlying principles of both Stoneman and SARA/IDEAS are very much alike.

5.1 Similarities

This section consists of short quotes from Stoneman followed by some discussion and comparison to the SARA/IDEAS requirements and design goals. The quotes are presented according to the section they appear in.

5.1.1 General Guidelines

> 3.F **USER HELPFULNESS**: High priority will be given to human engineering requirements in the design. The system shall provide a helpful user interface that is easy to learn and use...

The IDEAS system provides a rich flexibility for the user interface aspects of a tool, with accommodations for state-of-the-art interaction techniques. A specification language is used to describe the user interface of a tool in the IDEAS environment. Facilities are included for defining the syntax of the interface as well as logical and physical devices available. More details can be found in [Krel85b].

> 3.G **UNIFORMITY OF PROTOCOL**: Communications between users and tools shall be according to uniform protocol conventions.

IDEAS provides a consistent, common user interface across tools. A specification language is used to describe the user interface. This specification language can be used to implement those protocol conventions.

> 3.M **INTEGRATED**: An APSE shall provide a well-coordinated set of useful tools, with uniform inter-tool interfaces and with communication through a common database which acts as the information source and product repository for all tools.

IDEAS implements a paradigm to build an integrated environment. The IDEAS method follows an object-oriented strategy resulting in uniform inter-tools interfaces. For instance, as part of an experiment, three different developers were given the task to implement a certain SARA tool using the IDEAS method. As a result, any of the three implementations could be used without any changes in the system (and without the user noticing it).

A common database facility is used to store objects and their relations. This database service is available to all tools requiring it.

> 3.P **OPEN-ENDED**: An APSE shall facilitate the development and integration of new tools.

Not only is this true of the IDEAS system as well, but IDEAS goes one step further by providing a constructive method to integrate new tools into an existing environment.

5.1.2 Requirements for APSEs

> 4.A.2 The database shall offer flexible storage facilities to all tools.

The IDEAS database manager provides facilities for storing complete and partial designs, as well as relationships among the objects stored in the database.

> 4.A.3 The database shall permit relationships to be maintained between objects.

This is one of the requirements of the IDEAS database manager as well.

5.1.3 APSE Control Requirements

> 4.C.4 The user may access the virtual interface from a variety of physical terminal devices.

One of the design goals of the IDEAS user interface was to achieve device independence. It is very easy to write a device driver for a new I/O device. IDEAS has actually been ported to a variety of terminals, ranging from "dumb" ASCII terminal to sophisticated workstations.

5.1.4 APSE Toolset Requirements

> 4.E.4 The set of tools in an APSE shall remain open-ended: it shall always be possible to add new tools.

As I mentioned before, IDEAS provides a constructive method to integrate new tools with the potential to reuse significant amounts of software written for the existing tools. The entire user-interface aspects of the tools are taken care of by the IDEAS system and the tool builder is only concerned with implementing the semantics of the tool.

> 4.E.6 The principles for communication between tools and the user shall be simple and uniform throughout an APSE toolset.

By sharing the same user interface specification scheme, tools built using the IDEAS system will present a uniform interface to the user.

> 4.E.7 An APSE toolset shall offer comprehensive "help" facilities to APSE users.

Any tool built using the IDEAS method will have uniform on-line help facilities and documentation. For example, a help command can be entered by a user at any time to request help from the system.

5.2 Differences

This section tries to keep us aware that there are some important differences between APSE requirements and those of the SARA/IDEAS system.

5.2.1 APSE Toolset Requirements

> 4.E.3 Tools shall be written in Ada and where possible shall conform to standard interface specifications.

This a noble idea, but when an existing set of tools is being adapted to build an APSE (as it is the case here), it should not be required to rewrite them in Ada. Furthermore, Ada might not be the best programming language to write APSE tools in. I can see one significant problem that would arise if the SARA/IDEAS system would have to be rewritten in Ada. The lack of dynamic linking in Ada (being able to load and link code at run-time) would mean that all the code (including the Interpretation Domain one) would have to be linked a priori every time a change is made on any piece of code. This is unacceptable.

There is no requirement for the SARA/IDEAS system to write the toolset in any specific programming language. In fact, the first SARA system at MIT was implemented in PL/1, while the current SARA/IDEAS system is written in a Lisp dialect. The tools could be rewritten in Ada to conform to Stoneman requirement's, but that is not one of the goals of our research group.

The lack of availability of Ada compilers until very recently surely played a role in the APSE development process, as APSE developers had to wait for decent Ada compilers.

CHAPTER 6

GMB models for Ada Tasking Primitives

Concurrency is an important aspect of Ada, which is being designed as a language for embedded systems applications in which multitasking is required. The importance of concurrency in such applications has been recognized early in the design of Ada. It was first stated in the requirements for the language [Defe77,Defe78], and then in the rationale behind the design of Ada [Ichb79]. Concurrency is achieved in Ada by using the various tasking primitives described in the Ada Language Reference Manual [Defe83]. This is a controversial aspect of Ada, due, partly, to the fact that as of today, the area of concurrency and multitasking does not rely upon any universally accepted theoretical model or formalism. There have been various attempts to formally describe parallelism, but there is a lack of consensus on a common notation and these problems worsen when real-time issues arise. Debugging and testing Ada programs involving concurrency is, therefore, a difficult task. Sophisticated methods have been proposed for this purpose [Fair80,Germ84,Helm84].

To the best of my knowledge, all attempts to provide a formal semantic to Ada's tasking system have been unsuccessful so far. Furthermore, the official Ada LRM is hard to read casually. Therefore, I chose to provide my own informal semantics of Ada's tasking system instead.

At a first glance, the concurrency expressed in the Control Graph of the GMB and that of Ada seem far away apart. This chapter will try to reconcile those two views and models of concurrency by trying to come up with GMB models for all possible types of concurrency in Ada.

This will enable Ada programs to be converted to GMB models in order to analyze control flow anomalies (like potential deadlocks, liveness, boundedness, etc.). The early detection of these anomalies is crucial in order to prevent unexpected failures of Ada programs involving concurrency, and thus to increase the reliability of these programs. In addition, such models could serve as a basis for a formal definition of Ada's tasking system.

One difference which will not be dealt with here is that the GMB models are pseudo-static. That is, nodes and arcs can not be created or removed on the fly. That would make any attempt to analyze models all but useless. Ada, on the other hand, does have ways to handle dynamic creation of concurrent processes, in effect, changing the computation paths. I will not try to model that kind of concurrency.

It is not clear whether this restriction is in any way an important one. The class of problems that can not be expressed using a static control flow model includes atrocities such as self-modifiable programs.

6.1 Ada Tasking Primitives

The concurrency model in Ada is based on tasks. Tasks represent concurrent processes which are executed in parallel. Tasks come in different flavors, ranging from simple potentially non-interacting tasks to complicated synchronizing tasks with many scheduling and timing variations. The different flavors of tasking will be analyzed next.

6.1.1 Non-interacting tasks

The simplest form of concurrency in Ada is that where several concurrent tasks execute in parallel without interacting with each other. According to the semantics of Ada, these tasks will run in parallel and their parent unit will become terminated only after all these tasks come to termination themselves.

This example is illustrated by the following Ada template:

```
procedure parent is

  task T1;
  task T2;
  ...
  task TN;

begin

  null;

end parent;
```

Tasks `T1` through `TN` will run in parallel and `parent` will wait for all of them to finish before control is returned to the caller. Figure 6.1 shows the GMB corresponding to the Ada code above.

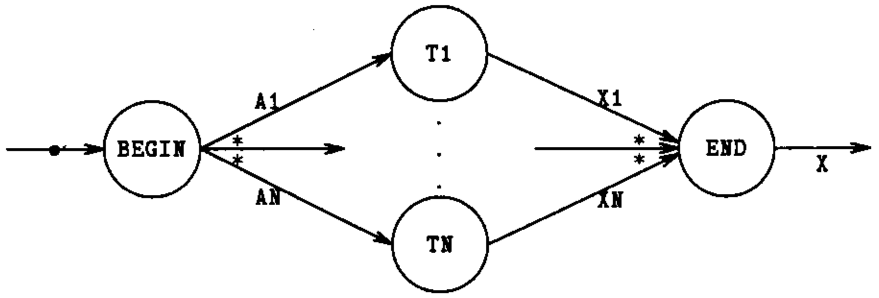

Figure 6.1: GMB model for Non-interacting Tasks

Control node begin will place a token on arcs A1, ..., AN, enabling thus nodes T1, ..., TN. The synchronization takes place at node END, which has to wait until it has a token on every input arc(X1, ..., XN) before it can proceed by placing a token on X.

6.1.2 Task synchronization - Rendezvous

The basic form of synchronization between tasks in Ada is the rendezvous, where one task calls an entry declared in another task. Consider a task defined by:

```
task T1;
  entry E1;
end T1;

task body T1 is
begin
  ...
  accept E1;
  ...
end T1;
```

and a caller task

```
  ...
T1.E1( ... );
  ...
```

when the caller task makes the call to T1.E1, it will wait until T1 reaches the **accept** statement for E1. Similarly, if the **accept** statement is reached before anyone calls that entry, T1 will wait until someone calls E1.

This is called a rendezvous. After the synchronization takes place, the commands in the body of the entry are obeyed (if any) and both the caller and T1 continue with their respective executions. Figure 6.2 shows the corresponding GMB control graph.

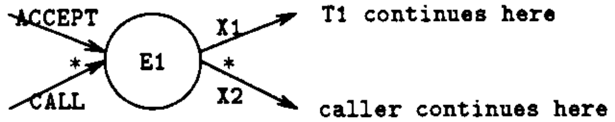

Figure 6.2: GMB model for simple rendezvous

For E1 to proceed, it requires a token on both input arcs (ACCEPT and CALL), achieving thus the required synchronization. Once the rendezvous is over, node E1 will place a token on both arcs X1 and X2 to enable both T1 and the caller to continue their execution. In some cases, however, the same entry could be called from different tasks and any of those calls should trigger E1. Figure 6.3 show the corresponding GMB control graph.

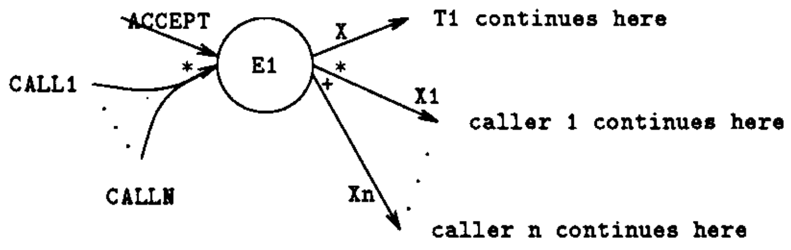

Figure 6.3: GMB model for compounded rendezvous

Here, a token, on both ACCEPT and any of CALL1, ..., CALLN will trigger node E1. If there are tokens on more than one CALL arc, they will be queued and processed in a FIFO (first in - first out) order. Upon termination, E1 will place a token on X to enable task T1 to continue and it will place a token on one of X1, ..., Xn to enable the caller to continue. In this case, it is the interpretation associated with E1 who is responsible for placing the token on the right control arc.

6.1.3 The select statement

The select statement allows a task to select from one of several possible actions. There are several forms of the select statement and all of them will be addressed here.

6.1.3.1 Selection among rendezvous

The simplest case is when the selection is to allow the accepting task to choose among several possible rendezvous. This would correspond to the following select statement:

```
select
```

```
  accept E1 do
    ...
  end;

or

  accept E2 do
    ...
  end;
...

or

  accept En do
    ...
  end;
end select;
```

The rendezvous can occur with anyone of the entries E1, ..., En that have been called upon and are waiting for rendezvous. If none is waiting for rendezvous, then the select statement waits until one of these entries is called. If rendezvous is possible with more than one entry, it has to choose one of them in an arbitrary way. Figure 6.4 show its corresponding control graph.

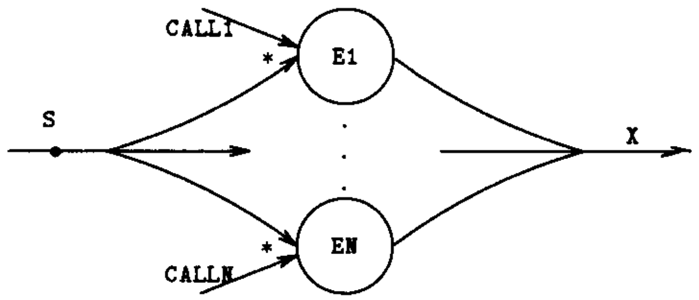

Figure 6.4: GMB model for the plain select statement

The single token on arc S is going to fire one of E1, ..., EN, depending on which ones of them are enabled by having a token on CALL1, ..., CALLN. If the token. Figure 6.5 shows the corresponding control graph.

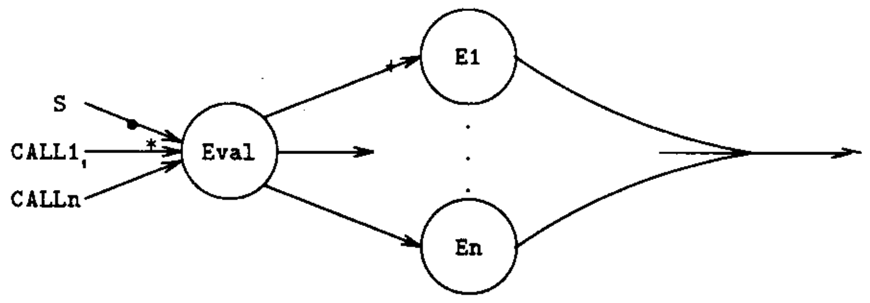

Figure 6.5: GMB model for guarding conditions

6.1.3.3 Delay Alternatives

As explained, the task reaching the select statement has to wait if none of the entries in the select had been called. It is possible to specify a time-out for this waiting period:

```
select
  accept e1 do
    ...
  end;
...

or

  accept en do
    ...
  end;

or

  delay 5.0;
  -- time-out statements;
end select;
```

If the rendezvous can not take place within 5 seconds, the time-out statements are executed and the select statement is considered completed. If one of the entries is called within 5 seconds, then the rendezvous takes place and the time-out is ignored. Figure 6.6 shows the corresponding control graph.

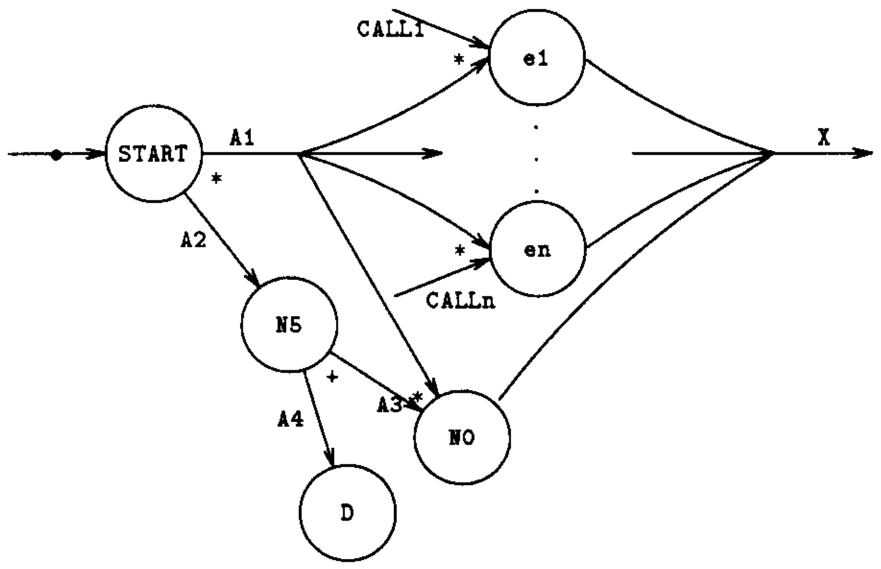

Figure 6.6: GMB model for delay alternatives

The START node will fire and place a token on both A1 and A2. The node on A2 will enable node N5 to fire. The interpretation associated with N5 waits 5 seconds before continuing. Meanwhile, one of the nodes e1, ..., en could be enabled by a token on CALL1, ..., CALLn. When N5 is ready to terminate (after the 5 seconds delay), its interpretation code will ask whether any of e1, ..., en have fired (this can be implemented in the data graph by making the nodes that output a token on CALL1, ..., CALLn to write to a dataset that can be read by the data processor associated with N5). If so, it will place a token on arc A4. Node D is a terminal node with no delay time used to absorb the token produced by N5.

If none of e1, ..., en has fired within the 5 seconds limit then node N5 will place a token on arc A3, enabling thus NO to fire since there's still an unconsumed token on arc A1. The interpretation associated with N0 will implement the time-out statements corresponding to the Ada code and it will place a token on arc X.

6.1.3.4 Immediate time-out

There is a special form of the select statement when the time-out branch has no delay, that is, a **delay** 0.0:

```
select
  accept e1 do
    ...
  end;
...

or

  accept en do
    ...
  end;

else
  -- alternative statements;
end select;
```

In this case, the alternative statements will be obeyed at once if the rendezvous can not take place with the other accept branches. Since this is a special case of the more general delay alternative, its GMB model is the same as the one for delay alternatives with the difference that node N5 is now a zero delay node (instead of waiting 5 seconds as before).

6.1.3.5 Timed-out entry call

Now we deal with selections on the calling side of a rendezvous. The first case is when an entry call is timed-out:

```
select
  T1.E1(...);

or

  delay 10.0;
  -- time-out statements
end select;
```

The time-out statements will be executed if the rendezvous with entry E1 of task T1 can not take place within 10 seconds, that is, if task T1 does not reach the **accept** statement for E1 within 10 seconds. Figure 6.7 shows the corresponding control graph.

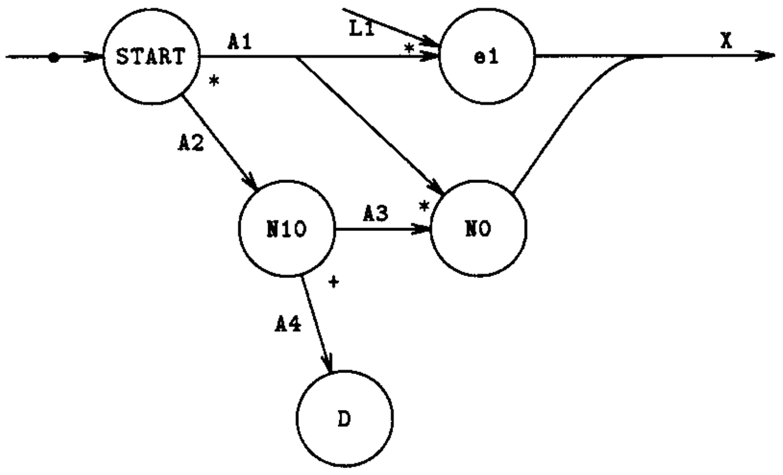

Figure 6.7: GMB model for timed-out entry calls

Here, node START will place a token on both arcs A1 and A2. Node N10 will wait for 10 seconds and, if no token has arrived on L1 (meaning that task T1 has not reached the accept statement for entry E1), it will place a token on arc A3, enabling thus node N0 which aborts the rendezvous and implements the time-out statements in the Ada code. Otherwise (if the rendezvous can occur), N10 will place its output token on A4 which will enable node D (a terminal node with no purpose other than to absorb that token). Node e1 will be ready to fire next, absorbing the tokens in A1 and L1 and executing the rendezvous.

6.1.3.6 Conditional entry call

The second case is when the rendezvous takes place only if there is no waiting involved (this is the counterpart for the immediate time-out case for accept statements):

```
select
  T1.E1(...);

else

  -- alternative statements
end select;
```

If and when the select statement is reached, entry E1 in task T1 is not waiting for rendezvous (i.e., T1 has not reached the "accept E1 ..." statement), the alternative statements are executed and the select statement is completed.

Here again, since this is just a special case of the timed-out entry call (with the waiting time being 0.0), it can be modeled using the same GMB model as before, changing node N10 to be a zero-delay node.

6.2 Discussion

This chapter has conjectured that all Ada tasking primitives can be modeled using the Graph Model of Behavior and presented actual GMB models for all of them. These results can be applied to build an translator from GMB models to partial Ada source code.

Once the concurrency present in Ada programs is expressed on the GMB Control Flow Graph, it can be analyzed for deadlocks and other control flow anomalies, an area where Ada is very weak right now.

In addition, the GMB simulator (another SARA tool) can be used to simulate, test and validate the model against functional and performance requirements.

CHAPTER 7

Translating GMB models into Ada program skeletons

After a GMB model has been built and simulated to the satisfaction of the designer, it must be translated into a conventional programming language for execution. The GMB simulator executes the model but is unable to produce a stand-alone running program that corresponds to the model.

GMB models could be translated manually to a particular programming language, but that task is a tedious one and likely to be the source of many errors, invalidating whatever benefits were achieved by using the GMB tools in the first place. An automatic translation scheme would eliminate these problems and make the SARA-based APSE an attractive one, since Ada code would be produced automatically.

One goal is, then, to write a translator that will convert GMB control graphs into Ada programs involving tasking. This limited goal is not to produce a complete executable Ada program but rather concentrate on the control flow aspects of it by translating the concurrency and control flow embedded in the control graph into Ada terms, i.e. to produce Ada program skeletons.

This Ada program skeleton will need to be filled with actual code for every program unit to make an executable Ada program. That code is assumed to come from the Interpretation Domain and could be expanded in-line by the translator (as opposed to placing a procedure call). This method is meant to support a process of composition of reusable software building blocks.

A further goal of automating the integration of the data graph in the generated Ada code is discussed in section 9.4.1.

7.1 Translation Schemes

Several automated systems to translate Petri Nets into programs have been built. Since the underlying principles of the GMB control graph and Petri Nets are similar and since it has been shown that they are equivalent in power, it is worthwhile to review those efforts.

7.1.1 Work by Nelson

Nelson, Haibt, and Sheridan built a system that translates Petri Nets into PL/I programs [Nels82,Nels83]. The Petri Nets are first translated into a procedural language called XL/1. There are actually two versions of XL/1: a parallel one and a serial one. One can specify whether the code should be generated for a serial XL/1 or a parallel one. The serial version of XL/1 can be translated into PL/I, compiled with a standard PL/I compiler, and executed thereafter.

However, the parallel version of XL/1 could not be translated into PL/I since PL/I does not have facilities for concurrency. That restriction may not be that severe for serial models but when translating into Ada, one wants to take advantage of all the forms of concurrency in the language.

7.1.2 The P-NUT System

The Distributed Systems group at the University of California, Irvine has produced a suite of tools called P-NUT, used to prove partial correctness and evaluate performance of Petri Net models.

There are two major parts to the P-NUT tools:

exhaustive state exploration: a *reachability graph* is built with all the states of the system which can be reached from a given initial state. This graph can be analyzed interactively using the Reachability Graph Analyzer.

path exploration: A Petri Net simulator is used to produce a single path through the reachability graph. Statistics can be gathered from that simulation.

One of the tools in the P-NUT environment, the P-NUT compiler, is used to generate executable Ada code from a standard Petri Net representation [Morg85]. The generated code makes use of the tasking features of Ada. This compiler always translates a complete Petri Net graph and does not have any facilities for Petri Net modules. The real problem is with the flat structure of Petri Nets, which does not allow for modularity in the graph.

The GMB solves that problem by providing modularity in the Structure Model. The translator can translate the control graph of a particular SM module as a package and use the SM structure for structuring the packages.

The basis for the P-NUT compiler implementation is that each transition is represented by an Ada task. A transition in a Petri Net is similar to a node in a control graph. A separate task, called the *Place Manager*, schedules the execution of transitions and their sequencing.

7.2 Designing the Translator

One of the first problems we need to attack is how are the different elements in the control graph to be represented in terms of Ada constructs. It is natural to see the modularity in the Structure Model imposing our modularity in Ada in terms of packages.

One could, therefore, represent every SM module by an Ada package (including environment SM modules). The translator would translate a specific GMB (associated with one SM module) at a time, or it could translate a complete GMB as well.

7.2.1 Control Nodes and Arcs

Control Nodes represent events, or processes, that can potentially execute in parallel. The most appropriate Ada construct would then be a task. Every control node will be translated into a task.

Control Arcs can be represented by the number of tokens they hold at a given moment. Therefore, every control arc can be translated into an integer variable.

7.2.2 The Task Scheduler

All tasks (representing all control nodes) will be running in parallel. There will be one additional task called `Scheduler` which will be responsible for activating and suspending the rest of the tasks according to the underlying semantics of the control graph.

This task scheduler is going to represent exactly what the underlying *token machine* does when it executes the control graph, with the only difference that it will allow concurrency when possible. The token machine is a sequential one.

Every task will communicate with the Scheduler via two entries: one signaling the activation of a control node and one signaling its completion. The Scheduler will have two entries for each control node named `T`: `T_start` and `T_end`.

The Scheduler will have to accept the rendezvous to start a task only when the input logic for that node has been satisfied. This can be done with a `select` statement with guards to evaluate the input logic for all nodes. Only those nodes whose input logic is satisfied will be considered for rendezvous.

Since the presence of a token on an arc is represented in the Ada program by the corresponding integer variable being greater than zero, the test for the guardian conditions will be built according to the following formula:

- if there is only one input arc named `a` to a node, its guardian condition will be `a > 0`.
- if the input logic involves **and** and **or** operators, then its guardian condition

will be the same as the input logic, except that all the arc names have been replaced by `a > 0`.

For example an input logic of the form `a1 and a2` will be translated into a guardian condition of the form `(a1 > 0) and (a2 > 0)`.

7.2.3 Interface to the Interpretation Domain

We need to include whatever code has been written in the Interpretation Domain as part of our executable Ada program. The Interpretation Domain code for a control node named `T` will be a parameterless procedure called `T_code` which we can call whenever the control node becomes activated.

The Interpretation Domain code could also be included in-line within the task itself. However, I think that making it a global procedure instead is a better approach as you could change the Interpretation Domain independently from the control graph without having to recompile your entire program (making use of Ada's separate compilation facilities).

7.2.4 Ada code for control nodes

As noted earlier, each control node will be translated into a task with no entries. The task will need to get the go-ahead signal from the Scheduler before it can execute and it will notify the Scheduler upon completion. It then needs to go back and see if it can fire again. The entire body will thus be surrounded by a loop. The body of the task will then be:

```
task body T1 is
begin
  loop
    scheduler.T1_start;     -- get the go-ahead from the Scheduler
    T1_code;                -- call Interpretation Domain procedure
    scheduler.T1_end;       -- notify Scheduler we are done
  end loop;
end T1;
```

7.3 Terminating the loops

The problem with the Ada code in the previous section is that every task executes an infinite loop. We need to find a way of terminating the Ada program somehow.

We first need to find out the state in which the various tasks will be when that situation occurs. The token machine terminates when there are no more nodes ready to be fired. That would correspond in our Ada program to all tasks waiting to rendezvous with the task scheduler's associated `start` entry.

The task scheduler will need to be aware of this situation and act accordingly. One way of implementing this is by keeping a count of the number of tasks that are waiting to rendezvous with their corresponding `start` entry. We will increment this variable at the beginning of the `start` rendezvous and will decrement it at the end of the end rendezvous.

We do not have to worry about concurrent updates to this global variable since all the updates are done by one process (the task scheduler). If we wanted every individual task to update this variable, we would need to protect it in a separate task with entries to read and write that variable.

7.3.1 Detection by the Task Scheduler

The Task Scheduler will detect this termination situation by an `else` branch in the `select` statement. When the value of this shared variable becomes the same as the number of tasks, then we know that the execution can not go any further. The Scheduler can then `abort` all calling tasks and terminate itself.

7.4 An Example

To get a feeling of what the translator should do, we can start with a simple example of a GMB:

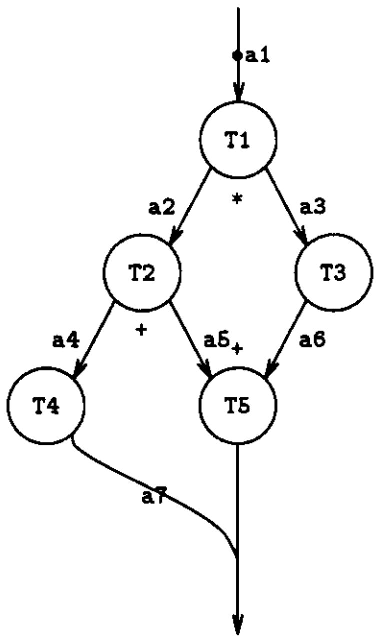

Figure 7.1: A simple GMB

Figure 7.1 shows a GMB with a non-trivial control flow complexity. A first step would be to write the body of the tasks associated with each control node. There will be one task for every control node and they will be named after the corresponding control node:

```
task body T1 is
begin
  loop
    scheduler.T1_start;
    T1_code;
    scheduler.T1_end;
  end loop;
end T1;

task body T2 is
begin
  loop
    scheduler.T2_start;
    T2_code;
    scheduler.T2_end;
  end loop;
end T2;

task body T3 is
begin
  loop
    scheduler.T3_start;
    T3_code;
    scheduler.T3_end;
  end loop;
end T3;

task body T4 is
begin
  loop
    scheduler.T4_start;
    T4_code;
    scheduler.T4_end;
  end loop;
end T4;

task body T5 is
```

```
begin
  loop
    scheduler.T5_start;
    T5_code;
    scheduler.T5_end;
  end loop;
end T5;
```

Now we have to write the task Scheduler for this GMB. Control arcs will be translated to integer variables:

```
procedure main is

  waiting_tasks : integer := 5;

task scheduler is          -- task specification. Just a list
  entry T1_start;          -- of all its entries
  entry T1_end;
  entry T2_start;
  entry T2_end;
  entry T3_start;
  entry T3_end;
  entry T4_start;
  entry T4_end;
  entry T5_start;
  entry T5_end;
end scheduler;

task body scheduler is

  a1: integer := 1;                        -- initial token on a1
  a2,a3,a4,a5,a6,a7 : integer := 0;       -- no other initial tokens

begin
  loop
    select
      when a1 > 0 =>
      accept T1_start do
        waiting_tasks := waiting_tasks - 1;
        a1 := a1 - 1;                      -- remove token from a1
      end T1_start;
    or

      accept T1_end do
        a2 := a2 + 1;                     -- place a token on a2
        a3 := a3 + 1;                     -- and on a3 as well
waiting_tasks := waiting_tasks + 1;
      end T1_end;

    or

      when a2 > 0 =>
      accept T2_start do
        waiting_tasks := waiting_tasks - 1;
        a2 := a2 - 1;                     -- remove token from a2
      end T2_start;

    or

      accept T2_end do
        chose(a4,a5);           -- place token on either a4 or a5
waiting_tasks := waiting_tasks + 1;
      end T2_end;

    or

      when a3 > 0 =>
      accept T3_start do
        waiting_tasks := waiting_tasks - 1;
        a3 := a3 - 1;                     -- remove token from a3
      end T3_start;

    or

      accept T3_end do
        a6 := a6 + 1;                     -- place token on a6
waiting_tasks := waiting_tasks + 1;
      end T3_end;

    or

      when a4 > 0 =>
      accept T4_start do
```

```
        waiting_tasks := waiting_tasks - 1;
        a4 := a4 - 1;                  -- remove token from a4
      end T4_start;

    or

      accept T4_end do
        a7 := a7 + 1;                  -- place token on a7
waiting_tasks := waiting_tasks + 1;
      end T4_end;

    or

      when (a5 > 0) and (a6 > 0) =>
      accept T5_start do
        waiting_tasks := waiting_tasks - 1;
        a5 := a5 - 1;                  -- remove token from a5
        a6 := a6 - 1;                  -- and from a6
      end T4_start;

    or

      accept T5_end do
        a7 := a7 + 1;                  -- place token on a7
waiting_tasks := waiting_tasks + 1;
      end T5_end;

    else

      -- if all tasks are waiting, abort everyone and exit
      if waiting_tasks = 5 then
        abort T1, T2, T3, T4, T5;
        exit;
      end if;

    end select;
  end loop;
end scheduler;

end main;
```

To understand this program, we can list the sequence of actions that take place in time:

1. Tasks T1 through T5 as well as the scheduler begin executing in parallel. All tasks stop as soon as they start executing, waiting to rendezvous with the scheduler. The scheduler enters the loop and, because of the guards, the select statement only selects entry T1_start for rendezvous since there is an initial token on arc a1 (a1 > 0 in the Ada program).

2. The rendezvous occurs and T1 continues executing in parallel its Interpretation Domain code (by calling procedure T1_code). The scheduler decrements a1 (representing the removal of a token from a1) and is ready for another round of the select statement.

3. The scheduler at this point can not rendezvous with anyone, so it will wait until T1 reaches its rendezvous call to T1_end. The scheduler will accept the rendezvous call from T1 and it will increment both a2 and a3. T1 completes and it goes back into the loop statement waiting to rendezvous with T1_start again.

4. The scheduler can now accept both entries T2_start and T3_start. One of them is chosen, be it T2_start. The rendezvous takes place and a2 is decremented. T2 starts executing its Interpretation Domain code.

5. Now the scheduler accepts the rendezvous with T3, decrementing a3. T3 starts executing its Interpretation Domain code (in parallel with T2. The

`scheduler` now waits until either `T2` or `T3` finish. Let us say that `T2` finishes first.

6. The `scheduler` would then accept the `T2_end` rendezvous and executes the "`chose(a4,a5)`" statement. This is a procedure that arbitrarily choses one of its arguments and increments it. This is how the nondeterminism in the control graph is translated.

7. The execution continues until there are no further possible rendezvous. Then the `else` branch of the `select` statement will be obeyed and the calling tasks will be aborted (and thus they will become terminated). The scheduler will then exit the loop and terminate.

7.5 The Translation Algorithm

We can now write an algorithm for the translation process:

1. Let `n` = number of control nodes in the GMB.

2. emit the `main` procedure heading and global variables (including translating control arcs into integer variables initialized according to their initial number of tokens).

3. For each control node, emit its associated task specification and body.

4. Construct the Task Scheduler by looking at the input and output logic expressions for every node.

5. Add an `else` clause to the `select` statement to terminate the program. Use the value of `n` to check against `waiting_tasks`.

CHAPTER 8

Ada as the Interpretation Domain Language for SARA

In order to have a SARA-based APSE, we must be able to use Ada as the Interpretation Language of the GMB. This chapter deals with defining exactly what would be needed to do that.

The chapter is structured as follows: The Interpretation Domain in SARA is defined as well as its place in the SARA method. The problems that arise when building inter-language interfaces are discussed and an interface is designed for an Ada Interpretation Domain. An Ada package is then built that defines that interface. the **T** part of the interface is also shown and an example is presented.

8.1 The Interpretation Domain in SARA

Interpretation is one of the three modeling domains in the GMB (the other two are Control and Data). The purpose of the Interpretation Domain is

- to define the data types of all datasets
- to define the initial values of all datasets
- to define the function performed by data processors

- to determine where the control token goes when there is an OR in a control node's output logic
- to assign delays

At the heart of the Interpretation Domain is an underlying Interpretation Language. The old SARA system at MIT used PLIP as its Interpretation Language. PLIP is an extension of PL/I, augmented with statements and declarations necessary in order to accomplish the functions mentioned above. The definition of the GMB does not preclude the use of a different Interpretation Language, and the current SARA/IDEAS implementation uses **T** (a Lisp dialect), augmented in the same way as PLIP. Other languages have been suggested to be used as the Interpretation Language as well (Concurrent Pascal, ISP, DCDL, MPDL and FORTRAN).

The interface between the Interpretation Domain and the rest of the GMB is realised by associating a piece of program written in the Interpretation Language with a particular Data Processor and, through the mapping between Data Processors and Control Nodes, with its associated Control Node(s).

This piece of program takes the form of a procedure, or a similar construct available in the Interpretation Language, that is called by the GMB Simulator whenever its associated Data Processor becomes active (triggered by the firing of any of its associated Control Nodes(s)).

8.2 The Interface Problem

In the old SARA system and in the SARA/IDEAS system, the problem of interfacing the Interpretation Domain to the rest of the system is non-existent, since the Interpretation Language is just an extension of the language the system is written in.

In our system, we would need to be able to call Ada programs from within our system (a running **T** program), which poses some inter-language interface problems.

Fortunately, we already have the necessary code to build an interface from **T** to other languages. **T** allows one to set aside a certain amount of memory to be used to load foreign code (compiled code from other compilers). In particular, there is a function called `define-apollo` that is used to define the heading of foreign procedures (name plus number and type of arguments). I will be making use of this function. Although this is an Apollo-specific function, similar facilities exist in other implementations of **T**.

8.3 Data Structures Communication

One of the practical problems in trying to interface an Ada program with the SARA/IDEAS system (or any other two programming languages for that matter) is that of communicating data structures written in one language to a program written in a different language.

This is certainly the case here. The SARA/IDEAS system has a complex web of **T** data structures, in a variety of flavors. Of particular interest to us are control nodes, control arcs and data arcs. Control arcs are needed because the Interpretation Domain will have to choose among control arcs when a control node's output logic contains an OR. Data arcs are needed because reading and writing to from and to datasets is done through them.

Interfacing **T** to any other language poses even more problems as **T** 's data structures are inaccessible from outside, among other reasons, because of their internal representation. The interface routines, thus, need to be written in **T** and calls to these routines will be made from the Ada part. It is clear that the interface routines form an abstract data type and will thus be implemented as an Ada package (see section 8.5).

8.4 Designing the Interface

A key issue is how to make references to **T**ś data structures from within Ada. We have to find some way of uniquely identifying a particular control node or control arc or data arc. Their name is a good choice but it is not unique. For instance, different control nodes in different modules can have the same name.

However, the name of a control node together with the name of the module in which the control node lies **is** indeed unique. We will represent that by a fully qualified name. For example, control node `INIT` in Figure 4.2 can be represented by the unique string "`universe/environment/INIT`".

The **T** routines will have to translate this string into a reference to the appropriate internal data structure representing that particular control node.

8.5 An Ada Package Specification

An Ada Package Specification is provided that defines this interface. This package should be included in every compilation of an Ada program unit in the Interpretation Domain.

```
generic
  type DATASET_TYPE is private;
   -- this is the type of the dataset
   -- this package needs to be instantiated for
   -- every type of dataset used

package IDEAS_INTERFACE is

  type T_OBJECT is private;

  procedure READ( in DATA_ARC : T_OBJECT;
                  out RESULT : DATASET_TYPE );
  -- reads a value from DATA_ARC and assigns it to RESULT

  procedure WRITE( in DATA_ARC : T_OBJECT;
                   in VALUE : DATASET_TYPE );
  -- writes VALUE to DATA_ARC

  procedure OUTPUT_ARC( in ARC : T_OBJECT );
  -- used to select among OR output arcs in the control graph

  procedure DELAY( n : INTEGER );
  -- assigns a delay of n time units

  procedure SET_ENVIRONMENT( in ENV_NAME : T_OBJECT );
  -- needs to be called with the string representation of the
  -- environment (SM module) this Data Processor lies in

private
```

```
  type T_OBJECT is string(1 .. 50);
  -- a T_OBJECT is really a string of up to 50 characters

end IDEAS_INTERFACE;
```

This package is a generic one, the generic parameter being the type of the dataset. Therefore, there will be one instance of this package for every type which is that of a dataset.

8.6 The T side of the Interface

All the functions provided by the `IDEAS_INTERFACE` package are actually part of the GMB simulator and are written in **T**. The Ada package serves the purpose of allowing Interpretation Domain Ada procedures to use the objects defined in it. There is no package body since the procedures in it are actually written in **T**. The facilities provided by the Ada compiler to link foreign code with Ada programs will have to be used. The following table lists the name correspondence between the functions in the `IDEAS_INTERFACE` package and the actual functions in the GMB simulator.

IDEAS_INTERFACE	**T**
READ	$INPUT
WRITE	$OUTPUT
OUTPUT_ARC	$OUTPUT_ARC
DELAY	$DELAY

Some **T** code will be written to convert the Ada parameters into what the **T** functions expect (this is part of the `define-apollo` package). Since we do not have Ada compilers for the Apollo workstations [1] (where the SARA/IDEAS system is being developed), we can only speculate about the exact details of the interface, but it will be something very close to that described above.

The scenario is one in which an Ada compiler is used to compile all the Interpretation Domain procedures. These procedures are loaded into the **T** environment using the `define-apollo` facilities described previously in this chapter. The GMB simulator will make calls to these Ada procedures as the simulation requires.

8.7 An Example

Continuing with our small example of a buffer system, we will define an Ada Interpretation Domain for some of the data processors in figure 4.3. Data processors `RDI` and `CHK` in the module `environment` will be chosen.

```
with IDEAS_INTERFACE; use IDEAS_INTERFACE;
-- include the generic package previously defined

procedure RDI is

  package MSG_INTERFACE is IDEAS_INTERFACE( message_slot );
  -- instantiate IDEAS_INTERFACE to the message_slot type

  message : message_slot;     -- a local variable

begin
  SET_ENVIRONMENT( "universe/environment" );
  -- initializes the MSG_INTERFACE package by providing it
```

[1] Actually, there are validated Ada compilers for the Apollos, but we do not have them yet.

```
  -- with the name of the SM module

  READ("dinput", message);
  -- reads a message from dinput

  if empty_message( message ) then        -- no more messages
    OUTPUT_ARC( "af1" );
    -- choose af1 in the control graph (signal termination)
  else
    -- valid message
    WRITE("di", message);
    -- write into dataset MESIN via data arc di
    OUTPUT_ARC( "acw" );
    -- signal to the buffer module (ready to send messages)
    DELAY( 2000000 );
    -- delay 2 mega units

end RDI;
```

Processor RDI reads one message from data arc `dinput` and, if there are no more messages to be read, then control arc `af1` is selected to place the control token on in the control graph. This signals termination.

If, on the other hand, the message is a valid one, then `RDI` writes it into dataset `MESIN` through data arc `di` and selects control arc `acw` as the one to get the control token from control node `SEND`. Finally, the delay associated with this processor is set to 2 million time units.

CHAPTER 9

Conclusions and Future Research

This dissertation has attempted to show that SARA is an Ada-compatible design method as it was suggested by two independent studies, METHODMAN I and the U.K. study. Below we list some of the most important contributions made by this dissertation.

9.1 Contributions

Provide an Ada-compatible method: The SARA method has been used to build a method-based APSE with a support environment.

Better understanding of concurrency: The control graph provides a formal method to understand and reason about concurrency, one of the aspects in which Ada is highly controversial and least understood.

Graphical representation: Ada packages are represented by the SARA Structure Model which has a graphical pictorial representation. Ada tasks are represented by SARA control nodes, also having graphical counterparts.

Static control flow analysis: Using the GMB Control Flow Analyzer, SARA designs can be analyzed for control-flow anomalies such as deadlocks, lack of liveness, etc.

Performance analysis: Using the performance analysis tools added to the SARA environment, queuing systems can be built and analyzed.

Animated Simulation: One of the main tools in the SARA environment is the simulator which provides an animated interactive simulation. Models can be simulated **before** any Ada code has been written.

Automatic Ada skeletons generation: A tool built as a result of this dissertation generates skeletons of Ada programs which can be executed to produce the same results as the original SARA model.

9.2 Usage Scenario

There are several possible scenarios in which these tools could be used. The best scenario is the one described by the goals of this dissertation: to aid in the design of concurrent Ada systems.

Users will use the SARA method and tools to build the models of the system being built. The Interpretation Domain will be the one described in chapter 8. Ada code will be written for each processor and the separate compilation facilities of Ada compilers will be used together with the foreign code interface available in **T** to compile these Ada procedures and load them into the **T** environment.

The GMB simulator will make calls to these Ada procedures to carry on the simulation. The model will be tuned according to the simulation results and analysis given by the Control Flow Analyser. The GMB-to-Ada translator will be used afterwards to get a stand-alone Ada program that can be compiled and executed outside the SARA/IDEAS environment. This Ada program might be the final version of the system being designed or it can serve as a prototype thereof.

Another possible scenario would be to analyze an existing concurrent Ada program using the SARA tools. Before this can be done, an Ada-to-GMB translator needs to be built.

9.3 Limitations

It is important to understand what the limitations of what was designed and built as a result of this dissertation are.

- There are no claims of being able to model all the features of Ada. In particular, there is no hope of being able to model the dynamic aspects of Ada tasking, as the GMB graphs are static. There have been suggestions to have dynamic GMBs, but that is questionable as most of the control flow analysis capabilities would be lost.

  For example, consider the following Ada code:

  ```
  task type T is
    entry E(...);
  end T;

  task body T is
    ...
  end T;
  ```

This task type declaration creates a template that can be used to create similar tasks. Objects can be declared of type `T`, and `T` can be used as a base type for arrays (of tasks) or as the type for record fields.

Moreover, tasks can also be created through access types (Ada jargon for pointers):

```
  type REF_T is access T;
  X : REF_T;
```

When `X` is assigned a value at run-time (such as in `X := new T;`), the task it is bound to becomes activated. This corresponds to control nodes being created on the fly in the control graph, which is forbidden according to the rules (the graphs are static).

No claims are made to cover other aspects of Ada, such as types, generics, etc. The main problem with that is that sockets in the SM are typeless (see section 4.4 for some possible extensions). The SM by itself is not powerful enough to express the richness of Ada packages (see section 4.5 for an extension to the SM).

- The Ada code skeletons generated from the GMBs are far from what one would get if the system was written in Ada from scratch (but then, that was never a goal and, as a prototyping tool, it is quite acceptable).

- The software modeling capabilities of the SM are quite inadequate. They can be supplemented by an appropriate Module Interconnection Language (such as Ada packages). No attempt has been made to make the SM play a more active role in the process of translation to Ada.

9.4 Areas of Future Research

This dissertation has uncovered many areas for future research that could make this SARA-based APSE a more effective environment. Below, we list some of these areas as well as some weaknesses of the current system that could be improved.

9.4.1 The GMB-to-Ada translator

The GMB-to-Ada translator is only a prototype. It could be improved by automating the integration of the data graph as well as the control graph. It could use the Ada type definitions given to the datasets and include them as part of the Ada program automatically. Also, debugging statements could be added to the Ada program if requested by the users.

The straightforward translation of the Token Machine does not produce optimal Ada code, in terms of concurrency control. The Task Scheduler is not really needed as all of its role can be distributed in the tasks themselves, as it would be the case in an Ada program written from scratch.

For instance, consider the following fraction of a GMB:

A corresponding Ada program translated by hand would be

```
procedure main is
...
begin
```

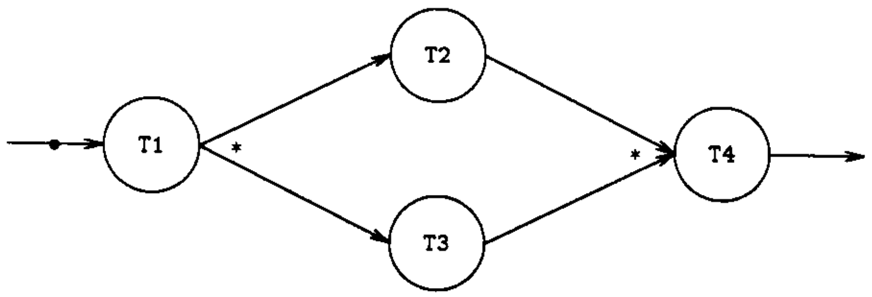

Figure 9.1: A simple GMB

```
    T1_code;

    declare
      task T2;
      task T3;

      task body T2 is
      begin
        T2_code;
      end T2;

      task body T3 is
      begin
        T3_code;
      end T3;

    begin
      null;
    end;

    T4_code;
    ...
  end main;
```

The difference between this code and the one produced by the GMB-to-Ada translator is that the former does not need a task scheduler. All the concurrency control is done directly by the Ada code, whereas, in the code produced by the translator, a task scheduler is needed (with the associated run-time overhead cost).

9.4.2 An Ada-to-GMB translator

Another interesting project would be an Ada-to-GMB translator. It could be used to convert Ada programs written elsewhere into SARA models and analyze them using the SARA tools. A starting point could be the results of chapter 6.

APPENDIX A

User's Guide to the Translator

This appendix includes a short user's guide to the GMB-to-Ada translator.

A.1 User's Guide

The GMB-to-Ada translator is embedded within the SARA/IDEAS system. It is one of the commands available at the GMB editor level. A typical menu at this point will look like this:

```
1. addControlNode

2. addControlArc

3. moveControlNode

4. ...

5. gmbtoAda

6. ...
```

The user will type `gmbtoada` in order to invoke the Ada translator. After typing the command name, the system will prompt for the two arguments required:

the GMB to be translated and a file name where the resulting Ada code will be stored.

- The GMB to be translated is given by pointing with the mouse (or any other pointing device available) to the SM module containing the GMB to be translated. This SM module has to be a terminal one (cannot have children modules) and has to contain a GMB.

- The system will then prompt for a file name to be used to write the Ada code onto. The file will be placed in the user's home directory. If the specified file exists, it will be overwritten.

The Ada code is enclosed within a `main` procedure, as described in chapter 7. To compile the resulting program, it needs to be linked with the Interpretation Domain Ada procedures to produce an executable Ada program.

APPENDIX B

List of Acronyms

The reader can easily loose track of the numerous acronyms used in this dissertation. Therefore below is a list the main ones, in alphabetical order.

ADE	Ada Development Environment
APSE	Ada Programming Support Environment
CAEDE	Carleton's Embedded system Design Environment
CCS	Calculus of Communicating Systems
CORE	COntrolled Requirements Expression
GMB	Graph Model of Behavior
IDEAS	Intelligent Design Environment for Analyzable Systems
JSD	Jackson System Design
KAPSE	Kernel APSE
MAPSE	Minimal APSE
LRM	(Ada) Language Reference Manual
MID	Module Interconnection Description
MIL	Module Interconnection Language

SARA	System Architect's Apprentice
SM	Structure Model

Bibliography

[Berr84] Daniel M. Berry, On the Use of ADA as a Module Interface Description, in *Proc. of the Hawaii International Conference on System Sciences* (January 1984).

[Buhr84] R.J.A. Buhr, *System Design With Ada*, Prentice Hall (1984).

[Buhr85a] R.J.A. Buhr et al., *CAEDE User's Manual*, Technical Report, Carleton University, Ottawa, Canada (May 1985).

[Buhr85b] R.J.A. Buhr et al., Experiments with Prolog Design Descriptions and Tools in CAEDE: An Iconic Design Environment for Multitasking, Embedded Systems, pp. 62–67, in *Proc. of the 8th International Conference on Software Engineering*, IEEE Computer Society, London, England (August 1985).

[Buhr85c] R.J.A. Buhr et al., An Overview and Example of Application of CAEDE: A New, Experimental Design Environment for Ada, pp. 173–184, in *Proc. of the Ada International Conference*, Cambridge Univ. Press, Paris, France (May 1985).

[Buhr85d] R.J.A. Buhr and G.M. Karam, An Informal Overview of CADA: A Design Environment for Ada, *Ada Letters*, 4(5):49–58 (April 1985).

[Buxt80] J. Buxton, *STONEMAN - Requirements for Ada Programming Support Environments*, U.S. Department of Defense (February 1980).

[Came83] John Cameron, *JSP & JSD: The Jackson Approach to Software Development*, IEEE Computer Society Press (1983).

[Camp78] Ivan Campos and Gerald Estrin, Concurrent Software System Design Supported by SARA at the Age of One, in *Proceedings 3rd International Conference on Software Engineering*, Atlanta, GA (May 1978).

[Defe77] U.S. Department of Defense, Requirements for High Order Programming Languages (revised IRONMAN), *SIGPLAN Notices*, 12(12):39–54 (December 1977).

[Defe78] U.S. Department of Defense, *Requirements for High Order Programming Languages (STEELMAN)*, Technical Report (June 1978).

[Defe83] U.S. Department of Defense, *Ada Programming Language Reference Manual (ANSI/MIL-STD-1815A)*, U.S. Government Printing Office (February 1983).

[Drob80] J. H. Drobman, *A Model-Based Design System and Methodology for Composition of Microprocessor-Based Digital Systems*, PhD dissertation, UCLA Computer Science Department (1980).

[Druf82] Larry Druffel, The Need for a Programming Discipline to Support the APSE: Where does the APSE path lead?, *Ada Letters*, 1(4):21–23 (May 1982).

[Estr78] Gerald Estrin, A Methodology for Design of Digital Systems – Supported by SARA at the Age of One, pp. 313–336, in *Proceedings of the National Computer Conference*, AFIPS (1978).

[Estr86] G. Estrin, R. Fenchel, R. Razouk, and M. Vernon, SARA: Modeling, Analysis, and Simulation Support for Design of Concurrent Systems, *IEEE Transactions on Software Engineering*, 12(2) (February 1986).

[Fair80] Richard Fairley, Ada Debugging and Testing Support Environments, *SIGPLAN Notices*, 15(11):16–25 (November 1980).

[Fisc84] Herman Fischer, ADATEC Dallas Panel on Design Methodologies and How They Relate to Ada, *Ada Letters*, 3(4):13–21 (February 1984).

[Germ84] Steven M. German, Monitoring for Deadlock and Blocking in Ada Tasking, *IEEE Transactions on Software Engineering*, 10(6):764–777 (November 1984).

[Goos83] G. Goos, *DIANA – An intermediate Language for Ada*, Springer-Verlag (1983).

[Helm84] David Helmbold and David Luckham, Debugging Ada Tasking Programs, pp. 96–105, in *Proceedings Conference on Ada Applications and Environments*, IEEE Computer Society, St. Paul, Minnesota (October 1984).

[Hugh79] J.W. Hughes, A Formalization and Explication of the Michael Jackson Method of Program Design, *Software Practice and Experience*, 9(3):191–202 (March 1979).

[Ichb79] J.D. Ichbiach et al., Rationale for the Design of the ADA Programming Language, *SIGPLAN Notices*, 14(6) (June 1979).

[Jack81] Mel Jackson, *Report on the Study of An ADA Based System Development Methodology*, Technical Report, British Department of Industry, London, England (September 1981), UK Task Force.

[Jack83] Michael A. Jackson, *System Development*, Prentice Hall International, Englewood Cliffs, New Jersey (1983).

[John75] S. C. Johnson, *Yacc - Yet Another Compiler-Compiler*, AT&T Bell Laboratories, Murray Hill, New Jersey (July 1975).

[Krel85a] Eduardo Krell and Edward Lor, Current State of the SARA/IDEAS Design Environment, in *SOFTFAIR II Proceedings (A Second Conference on Software Development Tools, Techniques, and Alternatives)*, IEEE Computer Society (December 1985).

[Krel85b] Eduardo Krell and Duane Worley, User Interface in the SARA Design System, in *Proc. of the IFIP Working Conference on the Future of Command Languages*, Rome, Italy (September 1985).

[Land83] Dorothy Landis, *Design Considerations for the Satisfaction of CAD Library Requirements*, Technical Report CSD-830614, UCLA Computer Science Department (June 1983).

[Land86] Dorothy Landis, CADIS: A Kernel Approach Toward the Development of Intelligent Data Management Support for Computer Aided Design Systems (June 1986), UCLA Computer Science Department.

[Lesk75] M. E. Lesk, *Lex - A Lexical Analyzer Generator*, AT&T Bell Laboratories, Murray Hill, New Jersey (October 1975).

[Matt86] Edmund R. Matthews and William Lively, The ATMAda Environment: An Enhanced Ada Programming Support Environment, *Ada Letters*, 6(3):61–64 (May 1986).

[McDe84] John McDermid and Knut Ripken, *Life cycle support in the Ada environment*, Cambridge University Press (1984).

[Morg85] E. Timothy Morgan, Instantiating Petri Nets as Concurrent Programs, submitted to the 1986 Petri Net Workshop (1985), Department of Information and Computer Science, University of California at Irvine, Irvine CA 92717.

[Nels82] R. A. Nelson, L. M. Haibt, and P. B. Sheridan, *Specification, Design, and Implementation Via Annotated Petri Nets*, Technical Report, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1982).

[Nels83] R. A. Nelson, L. M. Haibt, and P. B. Sheridan, Casting Petri Nets into Programs, *IEEE Transactions on Software Engineering*, SE-9(5) (September 1983).

[Pene79] Maria H. Penedo and Daniel M. Berry, The Use of a Module Interconnection Language in the SARA System Design Methodology, in *Proc. of the 4th International Conference on Software Engineering* (September 1979).

[Pene81] Maria H. Penedo, *The Use of a Module Interface Description in the Synthesis of Reliable Software Systems*, PhD dissertation, UCLA Computer Science Department (1981).

[Pete81] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, New Jersey (1981).

[Razo79] Rami R. Razouk, Mary Vernon, and Gerald Estrin, Evaluation Methods in SARA – The Graph Model Simulator, pp. 189–206, in *1979 Conference on Simulation, Measures and Modeling of Computer Systems* (1979).

[Razo80] Rami R. Razouk and Gerald Estrin, The Graph Model of Behavior, pp. 67–76, in *Proceedings of the Symposium on Design Automation and Microprocessors*, IEEE Computer Society, Piscataway, New Jersey (December 1980).

[Samp81] A.B.C. Sampaio, *Scheme of Attributes in Multilevel Design of Computer Systems*, PhD dissertation, UCLA Computer Science Department (1981).

[Wass82] Anthony Wasserman and Peter Freeman, *Software Development Methodologies and Ada (METHODMAN I)*, U.S. Department of Defense, Ada Joint Program Office (November 1982).

[Will83] Stephen Willson, *Introduction to Arcturus*, Department of Information and Computer Science, University of California, Irvine, CA (July 1983).

[Will84] Stephen Willson, *Arcturus User's Guide*, Department of Information and Computer Science, University of California, Irvine, CA (January 1984).