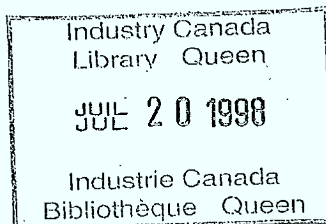
intellitech

Queen
P
91
C655
C6661
1982

The Definition and Specification of an Integrated Set of CAE Tools for Spacecraft Multiprocessor System Design

Queen
P
91
C655
C6661
1982

The Definition and Specification of an Integrated Set of CAE Tools for Spacecraft Multiprocessor System Design



Report No. INT-82-16

March 1982

Authors: Dr. C. Laferriere
Mr. W.T. Brown
Mr. J.G. Ouimet
Dr. S.A. Mahmoud

Approved by: Dr.S.A. Mahmoud

Government of Canada / Gouvernement du Canada

Department of Communications

DOC CONTRACTOR REPORT DOC-CR-SP-82-046

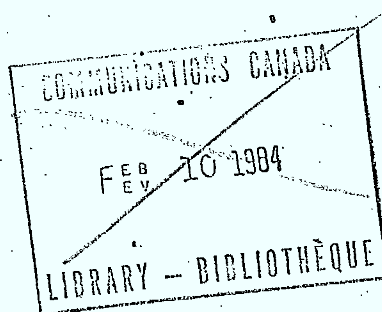
DEPARTMENT OF COMMUNICATIONS - OTTAWA - CANADA

SPACE PROGRAM

TITLE: The Definition And Specification Of An Integrated Set Of CAE Tools For Spacecraft Multiprocessor System Design

AUTHOR(S): C. Laferriere
W. Brown
J. Ouimet
S.A. Mahmoud

ISSUED BY CONTRACTOR AS REPORT NO: INT-82-16



PREPARED BY: Intellitech Canada Ltd.
352 MacLaren St.
Ottawa, Ontario
K2P 0M6

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: 3ER.36100-1-0273
SN: 0ER81-03151

DOC SCIENTIFIC AUTHORITY: R.A. Millar

CLASSIFICATION: Unclassified

This report presents the views of the author(s). Publication of this report does not constitute DOC approval of the reports findings or conclusions. This report is available outside the department by special arrangement.

DATE: March 1982

Preface

This work was performed for the Department of Communications, Communications Research Centre under DSS Contract No. OER81-03151, entitled "Computer-Aided Engineering Tools for Spacecraft Multi-Microprocessor Design", from September 15, 1981 to March 31, 1982. This report is one of the following four contract deliverables:

1. Executive Summary
2. Report #1 - Review of Multiprocessor Systems and their Spacecraft Applications.
3. Report #2 - A Survey of Computer-Aided Engineering (CAE) Tools for the Design and Simulation of Multiprocessor Systems.
4. Report #3 - The Definition and Specification of an Integrated Set of CAE Tools for Spacecraft Multiprocessor System Design.

Acknowledgement

The study team gratefully acknowledges the technical guidance of Mr. R.A. Millar of the Communications Research Centre. His knowledge and experience in the field of computer simulation of spacecraft systems have contributed to the quality of the work and provided a constant source of encouragement to the study team.

As well, the study team wishes to thank Mr. J.M. Savoie, also from CRC, for his fruitful discussions and critical reviews.

Table of Contents

Page

Preface i
Acknowledgement ii
Table of Contents iii
List of Figures v
List of Tables vii

1. Introduction 1

1.1 Scope and Definition of Multiprocessors 5
1.2 Structure of the Report 7

2. Proposed Methodology 9

2.1 Scope of Existing Tools 10
2.2 Toward an Integrated Set of CAE Design Tools 13
2.3 Approaches for Modelling Functional Component Description 14

3. Specification/Validation 21

3.1 Introduction 21
3.1.1 Problem Definition 21
3.1.2 Spacecraft Environment 23
3.1.3 Design Philosophy 26
3.1.4 Overview of the Section 28

3.2 Specification/Decomposition 29
3.2.1 High Level Specification 29
3.2.2 Methods of Decomposition 30
3.2.3 Formalizing the Specification/ Decomposition Process 34
3.2.4 Ada as a Specification Tool 36
3.2.5 A Decomposition Example 44
3.2.5.1 Various decomposition levels 45
3.2.5.2 Observations on the model 62
3.2.5.3 System simulation and testing 69
3.2.5.4 Computer aided tools and the specification process 74

3.3 Validation of Specifications 76
3.3.1 Validation 76
3.3 2 Testing 80
3.3.3 Verification 83
3.3.4 Automated Verification Systems 96
3.3.5 Proposed Validation Capabilities 107

3.4 Summary and Conclusions 112

4. Performance and Reliability.......................... 115

4.1 Introduction.. 115
4.2 Scope of CAE Tools in the Performance Area......... 116
4.2.1 Architecture Selection........................ 117
4.2.2 System Model (Hardware and Software Selection)............................ 118

4.3 Reliability Models.................................... 122
4.3.1 Component Reliability Model.................. 122
4.3.2 Exhaustion of Spares Model.................. 123
4.3.3 Imperfect Coverage Model..................... 126

4.4 Resource Usage Models.............................. 127
4.4.1 Simple Totals Model........................... 128
4.4.2 Effects of Allocation Model................ 129
4.4.3 Effects of Dynamic Interaction Model........ 131

4.5 Areas for New or Improved CAE Tools............... 134
4.5.1 Architecture Independent CAE Tools.......... 135
4.5.1.1 Ada Based General Purpose Simulation Language................ 135
4.5.1.2 Exhaustion of Spares Analysis Tool.......................... 136
4.5.1.3 Imperfect Coverage Analysis Tool................................. 136
4.5.1.4 General Reliability Analysis Tool................................. 136
4.5.1.5 Resource Allocation Analysis Tool................................. 137

4.5.2 Architecture Dependent CAE Tools............ 138
4.5.2.1 Hardware Reliability Analysis Tool................................. 138
4.5.2.2 Static Resource Usage Analysis Tool................................. 140
4.5.2.3 Dynbamic Resource Usage Analysis Tool................................. 141

4.6 Summary.. 142

5. Integration with Existing Tools....................... 156
5.1 Introduction.. 156
5.2 Transition Between Different Tools............... 157
5.3 Selection of Existing Tools......................... 159

6. Summary and Further Work.............................. 166
6.1 Summary.. 166
6.2 Further Work.. 171

References.. 173

List of Figures

		Page
Figure 2.1	Multiprocessor Specifications/Design Levels and Corresponding CAE Tools............	17
Figure 2.2	Design Phases Using an Integrated Set of CAE Tools................................	18
Figure 2.3	Functional Decomposition in a Top-down Approach..............................	19
Figure 2.4	Example of a Data Flow Model..................	20
Figure 3.1	A Spacecraft Control System..................	24
Figure 3.2	Hierarchy of Machines and Programs............	26
Figure 3.3	Machine Equivalence...........................	27
Figure 3.4	Tree Structure Resulting from Functional Decomposition......................	30
Figure 3.5	Dataflow Example..............................	32
Figure 3.6	Result of the Mixed Approach..................	33
Figure 3.7	An Ada Package................................	37
Figure 3.8	An Ada RendezVous.............................	38
Figure 3.9	Basic Structure of a Specification Block......	41
Figure 3.10	Device Servicing in Ada.......................	43
Figure 3.11	Description of the Example....................	44
Figure 3.12	A First Attempt at Decomposition..............	46
Figure 3.13	Data Flow Graph...............................	47
Figure 3.14	Definition of Various Commands................	48
Figure 3.15	Functional Decomposition (at various stages)...........................	52
Figure 3.16	System Representation with Ada Building Blocks................................	53
Figure 3.17	Ada RendezVous and Passing of Control........	56
Figure 3.18	CommandStringInterpreter......................	59
Figure 3.19	Expansion of a Separate Procedure............	60
Figure 3.20	Description of the Reset Module..............	61
Figure 3.21	Procedure Call Arrangements in Ada...........	64
Figure 3.22	Representation of an Input Module............	67
Figure 3.23	System Model and TestBed.....................	69
Figure 3.24	Device Simulation Module.....................	70

Figure 3.25	Simulation and Testing Package................	71
Figure 3.26	Example of Transformation......................	76
Figure 3.27	Validation of the Design Process..............	77
Figure 3.28	Skeleton of a Program Control Structure.......	81
Figure 3.29	A Program's Domain and Range..................	81
Figure 3.30	Paths and Assertions...........................	85
Figure 3.31	Simple Loop Example............................	88
Figure 3.32	Subprogram for Simple Division................	91
Figure 3.33	Subprogram with Assertions....................	91
Figure 3.34	Design and Verification of Programs...........	97
Figure 3.35	Procedure InsertSorted.........................	99
Figure 3.36	Verification Condition 15.....................	102
Figure 3.37	Gypsy Verification Environment................	104
Figure 3.38	Description of the HDM System.................	105
Figure 3.39	An Ideal System................................	107
Figure 3.40	Proposed Interim Verification System..........	110
Figure 4.1	Performance/Reliability Design Methodology..................................	145
Figure 4.2	Hardware Performance Design Methodology Example..........................	146
Figure 4.3	Resource Usage Design Methodology............	147
Figure 4.4	Example Simple Totals Model...................	148
Figure 4.5	Example Access Graph...........................	149
Figure 4.6a	Example Effects of Loading Model..............	150
Figure 4.6b	Example Effects of Loading Model..............	151
Figure 4.6c	Example Effects of Allocation Model...........	152
Figure 4.7a	Hardware Reliability Analysis Tool Example.................................	153
Figure 4.7b	Hardware Reliability Analysis Tool Example.................................	154
Figure 4.7c	Hardware Reliability Analysis Tool Example.................................	155

List of Tables

Page

Table 5.1 Existing Tool Functional Characteristics...... 163

Table 5.2 Existing Tools Implementation Characteristics................................ 164

Table 5.3 Existing Tools Selection Evaluation........... 165

1.0 Introduction

Interest in multiprocessor and distributed intelligence computer systems have increased dramatically in recent years. This interest has been fostered by the availability of micro-processors with ever increasing performance-price ratios and the expected emergence of monolithic systems with still higher capabilities in the near future.

Advances in LSI and VLSI semi-conductor technology have significantly reduced computer hardware weight, power consumption and cost. It is now feasible and practical to employ multi-processor systems on spacecraft in order to increase the reliability, extend mission duration and satisfy increasingly more computational demand during the mission.

The development of multiprocessor and distributed intelligence computer systems and their utilization in various applications have been impeded by the lack of an appropriate theoretical base. The control of systems containing large number of processors is not well understood. While considerable work has been done recently to develop a theoretical base, it seems unlikely that this work will have significant impact on practical system design in the near future. As a result, multiprocessor system designers have turned to the use of CAE tools for the

the development of such systems. Such CAE tools are used, in general, to support the skill level of the designer, provide insight into the attributes of alternative architectures, allow evaluation of these architectures and support the development, simulation and testing of actual multiprocessor systems.

More specifically, computer-aided engineering tools are required to simulate alternate hardware configurations, evaluate the software implications on selecting a particular hardware configuration, perform required hardware-software tradeoffs, establish that the specified hardware and software are compatible and that overall system performance requirements are met. All of these must be done at an early stage in the design process, before the software is coded and the hardware is constructed.

In the absence of such computer-aided engineering tools, it is difficult for the designer to assess and evaluate system performance adequately before constructing a breadboard prototype, developing its software, and testing the resulting system. At this late stage in the design process, discovered inadequacies and inconsistencies are expensive and time-consuming to correct and often require significant redesign. With the appropriate CAE tools, the chances of this happening at such a late stage in the design process are minimized.

In an accompanying report [MAHM82], a survey which examined existing CAE tools for multi-processor design has been presented. The survey identified six specification and design phases for the purpose of identifying the utility and applications of the various available tools. These phases are:

1. The Requirements Specification phase,
2. The Functional Components Definition phase,
3. The Architectural Design phase,
4. The System Model phase,
5. The Processing Element Partitioning (Register Transfer Level) phase, and
6. The Logic Design (hardware) phase.

The survey indicated the availability of many design simulation tools which satisfy different design needs, depending on the design level (or levels) for which it is developed. Unfortunately, no one simulator was found to be useful throughout all specification and design phases. This multiple simulator approach outlined in the survey has two advantages and several disadvantages. The advantages are:

1. Each simulation can be written in a language tuned for one particular level, and
2. Each simulation tool can optimize its runtime organization for one particular task.

The disadvantages include the following:

1. The design effort is multiplied by the necessity of learning several simulator systems and writing a design in each.
2. The possibility of error is increased as more human manipulation is involved.
3. As the design becomes increasingly fragmented, it becomes impossible to simulate an entire multiprocessor system at a low level of abstraction. Therefore, only small fragments can be simulated at any one time.
4. Each fragment needs to be driven by a supply of realistic data and its output must be interpreted. This may make the software written to serve these needs extremely costly.

Several tools have been developed to overcome the above difficulties and provide the designer with a uniform simulation approach starting at the architecture design level and going down to the register transfer simulation level. The utility of these tools can be improved substantially by augmenting them with a high level specification package which allows the designer to describe the functional components of the system being designed and to interface this high level

description to existing tools at the architectural level. In addition, two design aspects must be addressed in augmenting existing tools:

1. Redundancy and fault-tolerance characteristics analysis must be provided at the architectural levels if the tools are to be useful in the design of spacecraft (or avionics) multi-processor systems.
2. The high level specification language must contain mechanisms for system verification. As the design process continues, these mechanisms will evolve naturally towards validation and verification of the software.

The study reported here is concerned with the issue of augmenting existing tools to generate an integrated set of multiprocessor design and simulation tools that can be useful throughout the various phases of the design.

1.1 Scope and Definition of Multiprocessors

The proliferation of various publications dealing with interconnecting microprocessors to form unified systems has given rise to some ambiguity with respect to the definition of "multiprocessor systems" and "distributed microprocessor systems". To avoid such ambiguity, we introduce a definition

for the term "multiprocessors" which will be used throughout this report. We also define the "scope of configurations" of such systems considered to be relevant for spacecraft applications.

For the purpose of this report, we define a multiprocessor system to be [JENS78]:

> "a multiplicity of microprocessors that are physically and logically interconnected to form a single system in which overall executive control is exercised through the cooperation of decentralized system elements".

Moreover, we define the scope of multiprocessor systems considered in this study through the following general characteristics:

1. The microprocessors forming the system, as well as all other system elements co-exist in the same locality (i.e., no telecommunication lines are used since the elements are not geographically separated).

2. The microprocessors and other system elements are interconnected according to one of alternative structures (uni or multi-bus, a loop or ring connection, a matrix switch, etc.).

3. Conceptually, a single executive manages all of the system's physical and logical resources in an integrated fashion. The kernel (control) logic and data structures are replicated among a number of processors or memories.

4. The number of processors to be interconnected is relatively small (e.g., under 30 processors).

5. Redundancy in the hardware is assumed through the use of identical spares, which along with other fault recovery mechanisms constitute what is known as "fault-tolerant" architectures.

1.2 Structure of the Report

The basic methodology adopted for generating an integrated set of CAE tools for multiprocessor systems is explained in Section 2. It is shown that the underlying concept is based on a top-down design approach starting from a high level specification phase.

Section 3 of this report introduces the basic definitions and specifications of a high level design tool constructed using ADA as the basic programming language. An example is provided to illustrate the basic functional

decomposition process. The example is based on a hypothetical application of a multiprocessor system as a controller for a set of sensors and actuators in a spacecraft.

Section 4 examines the performance evaluation aspects associated with designing multiprocessor systems for spacecraft applications. Two performance criteria are considered: resource utilization and reliability (redundancy and recovery from failures). The use of CAE tools to assist in evaluating both criteria is examined.

Section 5 investigates the interfaces needed to integrate existing CAE design tools at the architectural and system model levels with the high level functional specification tool described in Section 3. Finally, Section 6 presents a summary of the contents of this report and a set of recommendations for future work aimed at assimilating an integrated set of CAE design tools for multi/microprocessors.

2.0 Proposed Methodology

As explained previously, current design practice of multiprocessors consists of a series of steps which starts by stating the general requirements and terminates by detailed hardware and software design, development and testing. A broad spectrum of tools exists to assist the designer at each step.

Our survey of existing tools [MAHM81] indicated that while several design and simulation tools exist to satisfy different needs, no one simulator is useful throughout all the specification and design phases. In addition, a gap exists at the high levels of the design which makes it difficult to use the output of the tools at the functional components specification level to generate the input to the architecture selection stage. This gap will be explained before introducing the proposed methodology.

In this section, we review briefly the design phases and the general features of the tools used in each phase. We use the review to highlight certain deficiencies which exist in the spectrum of available tools. The review is followed by an explanation of the concepts underlying the methodology proposed in this report.

In a general sense, it will be shown that the proposed methodology is aimed at closing the gap which exists between the functional component specification phase and the

architecture design phase, and at the same time augmenting existing tools with mechanisms to evaluate the performance and the reliability of the system at various design stages. This will ultimately result in an integrated set of CAE tools which can be utilized in a consistent fashion throughout the various design levels.

2.1 Scope of Existing Tools

Figure 2.1 illustrates the specifications and design levels of multiprocessor systems (see survey report [MAHM82], also referenced in the preface of this report). Existing tools can be classified according to the design level (or levels) at which the tool is utilized.

At the requirement specification level, tools are used to define the demands placed upon the system in a complete, consistent and unambiguous set of statements. The output of the tools is usually given in a machine readable format. The input can be generated by several authors and the tool is expected to merge the input from these authors while removing all redundancies. The output document is used by all design team members as a reference for the requirements of the system.

Design occurs after the requirements phase has been completed. The first level of the design process consists of an orderly definition of the main functional components of

the system which satisfy the requirements. This is followed by further decomposition of the main functional components into smaller subfunctions, and the process continues until a system architecture emerges in which a hardware-software model can be abstracted and the sub-functions can be mapped into elements in the model.

Several tools are currently available for automating the process of defining the functional components of the system and for decomposing these components into smaller, less complex components. The utility of these tools in the design of multiprocessors is limited by the following factors:

1. the tools lack the ability to describe the dynamic interaction between the decomposed functional components. Thus aspects such as concurrency, synchronization, etc., cannot be formally described.
2. The behaviour of the system cannot be described formally using existing tools at the functional components level. The designer is forced to extract this behaviour manually before deciding on a suitable architecture. This informal extraction is bound to generate errors and inconsistencies.
3. The output of existing tools is not interfaceable directly to architecture

level procedural simulation languages with their formal syntax. This creates a gap in the transition to the next lower design level, i.e., the architecture selection level.

4. Subfunctions and other resources (e.g., data structures) that are shared by the main functions, as well as their access control structures, cannot be described easily by existing tools, particularly in the dynamic interaction environment of spacecraft multi/microprocessors.

The above difficulties motivate the development of a tool at the functional component definition level which can be mutually integrated with the tools used at the architecture selection level.

A large number of simulation tools exist at the architecture selection level and the levels below it. These tools can be substantially enhanced with two additional features:

1. The incorporation of formal rules to verify the modelled behaviour of each module and of the entire system.
2. The incorporation of reliability analysis tools to model and simulate the fault-tolerance characteristics of the system.

The development of a formal specification and verification tool at the functional component description level, together with the enhancement of existing tools to handle the analysis of reliability requirements can be viewed as the catalyst of the proposed methodology.

2.2 Toward an Integrated Set of CAE Design Tools

The discussion of Section 2.1 indicated that a "missing link" exists at present in the set of currently available CAE design tools for multiprocessors. This "missing link" is at the functional component description level (see Figure 2.1). A tool is needed at this level which will be utilized in the design level between the requirements specification level, and the system architecture level. The newly developed tool can be interfaced with existing tools for the requirements specification and with tools currently used in the simulation of system architecture.

Figure 2.2 depicts the role of the proposed Functional Component Specification tool in the multi-phase design approach. The tool is utilized to translate system requirements specification into functional components described using a high level behavioural description language. As well, preliminary system architectures can be selected and evaluated using this tool. The output of this phase will serve as input to the next phase; namely, the detailed architecture phase, for which excellent design tools exist at present.

The main concepts underlying the high level functional description tools are introduced in Section 2.3 with the full details presented in Section 3. The evaluation tools for resource utilization performance models and reliability characteristics models are discussed in Section 4.

2.3 Approaches for Modelling Functional Component Description

The development of a high level specification tool to simulate the functional components is based on a modelling approach which captures the behaviour of its component, its relationship to other components and the interconnection of the components to form the entire system. Two alternative approaches are generally employed for this purpose:

1. A top-down decomposition approach,
2. A data flow model approach.

The general concept underlying the top-down approach can be summarized as follows (see Figure 2.3):

- each of the main functional components of the system, defined in the requirements specification phase, is described in terms of its external interfaces and the basic functions it performs.
- each component can then be treated separately and divided into subcomponents. Collectively, these subcomponents must maintain the same external interfaces of

the parent component. Again, each sub-component is desribed in terms of its external interfaces.

- the process of decomposition can be applied iteratively to the subcomponents until they are refined into the simplest possible (elementary) components.
- a validity check is applied at each decomposition step to ensure the completeness and consistency of the decomposition.

The top-down design approach is suitable for the design of complex systems since it systematically reduces the design process to simpler components which can be tackled separately. Its main drawback in the multiprocessor area lies in its inability to capture the relationship between the various data structures at different decomposition levels. This is because many data structures are difficult to decompose in a hierarchical order.

The data flow model approach represents the system as a set of computational modules, sequentially processing a flow of data. The modules form a network with data merging and branching out. The source of data is a set of input modules and the terminal is a set of output modules. An example of data flow model is depicted in Figure 2.4. This model bears

some relation to many of the features of the spacecraft processing environment in that the latter consists of input sources (sensors), output terminals (actuators) and a set of data processing modules (algorithms). In general, data flow models are not convenient to use in describing and designing complex systems.

The development of a high level functional component description tool (Section 3) is based on a hybrid approach in which the data flow model is used initially to describe the behaviour of the system. Each computational module in the data flow model is regarded as a system component which is then decomposed in a top-down approach. This hybrid approach will be shown to combine the advantages of the data flow model and the top-down decomposition approach.

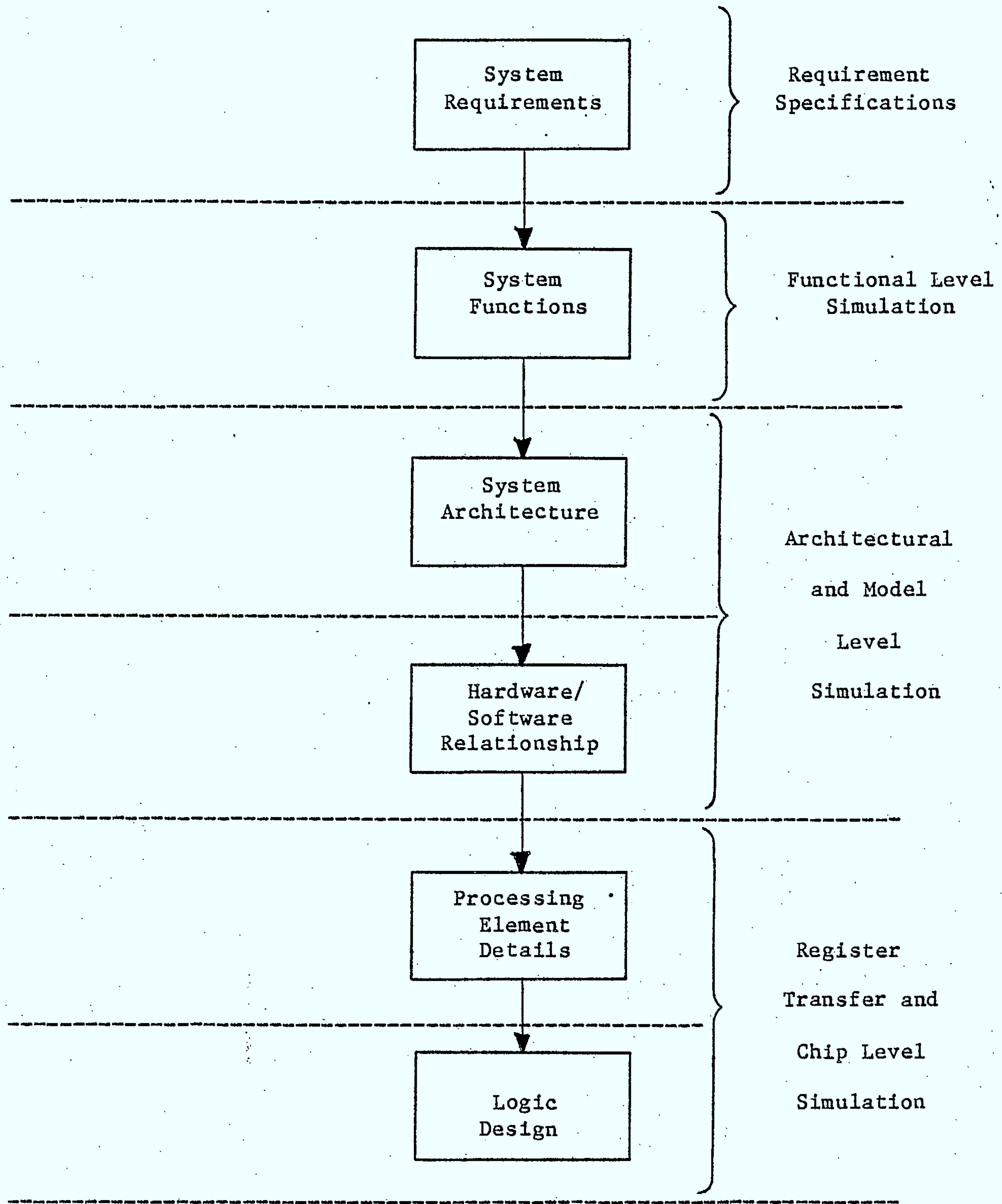


Figure 2.1 Multiprocessor Specifications/Design Levels and Corresponding CAE Tools

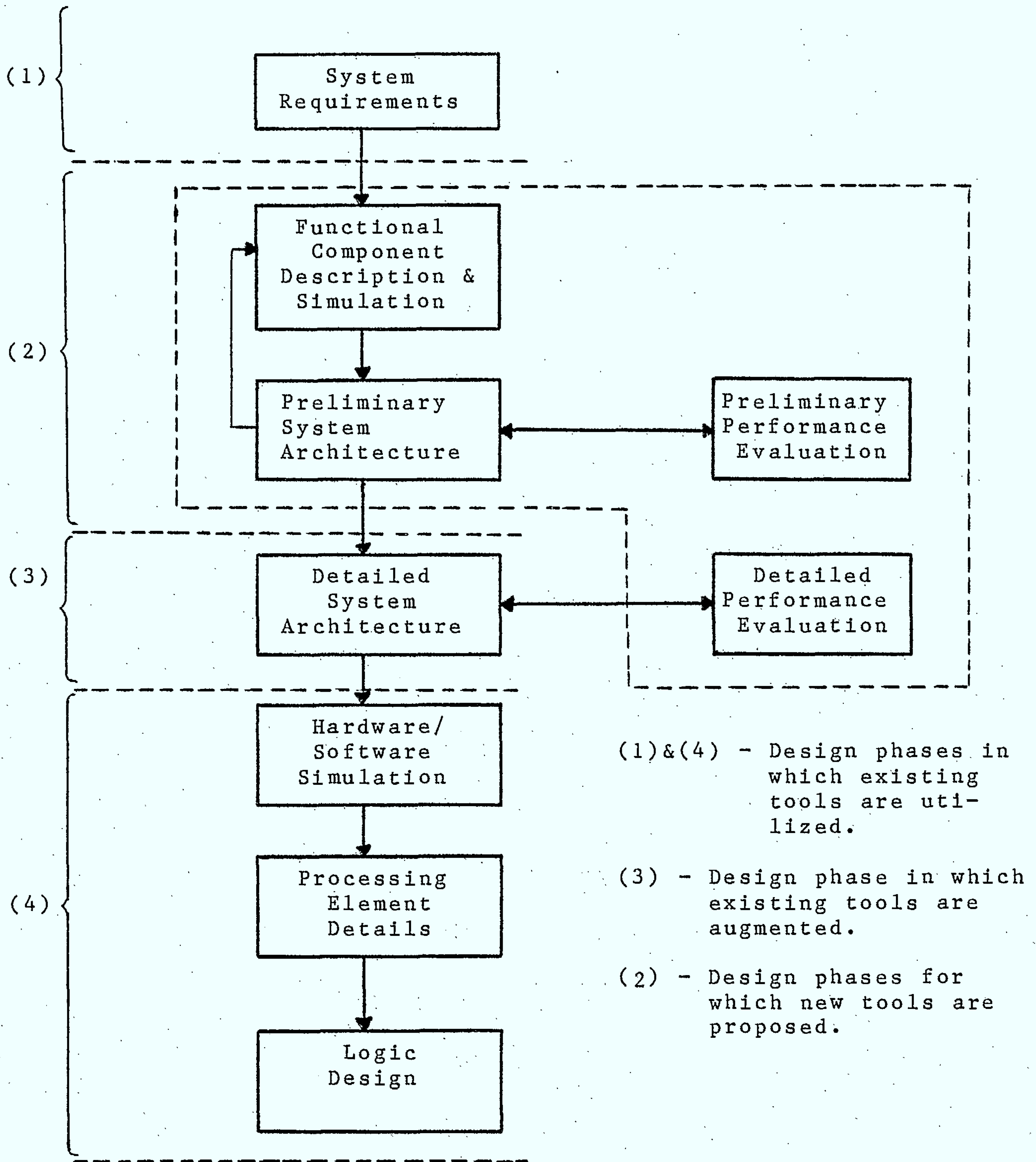


Figure 2.2
Design Phases Using an Integrated Set of CAE Tools

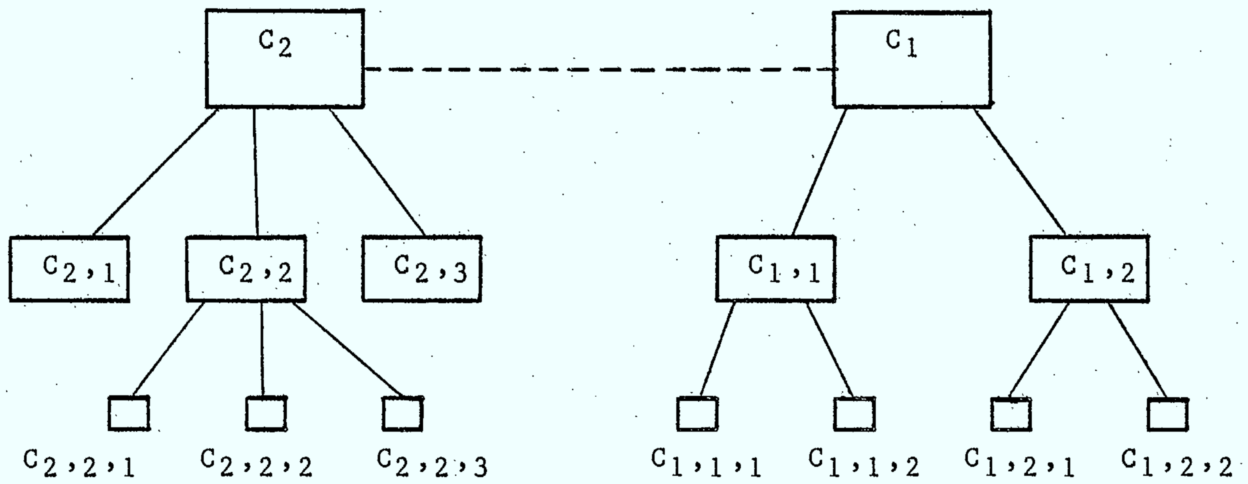


Figure 2.3
Functional Decomposition in a Top-down Approach

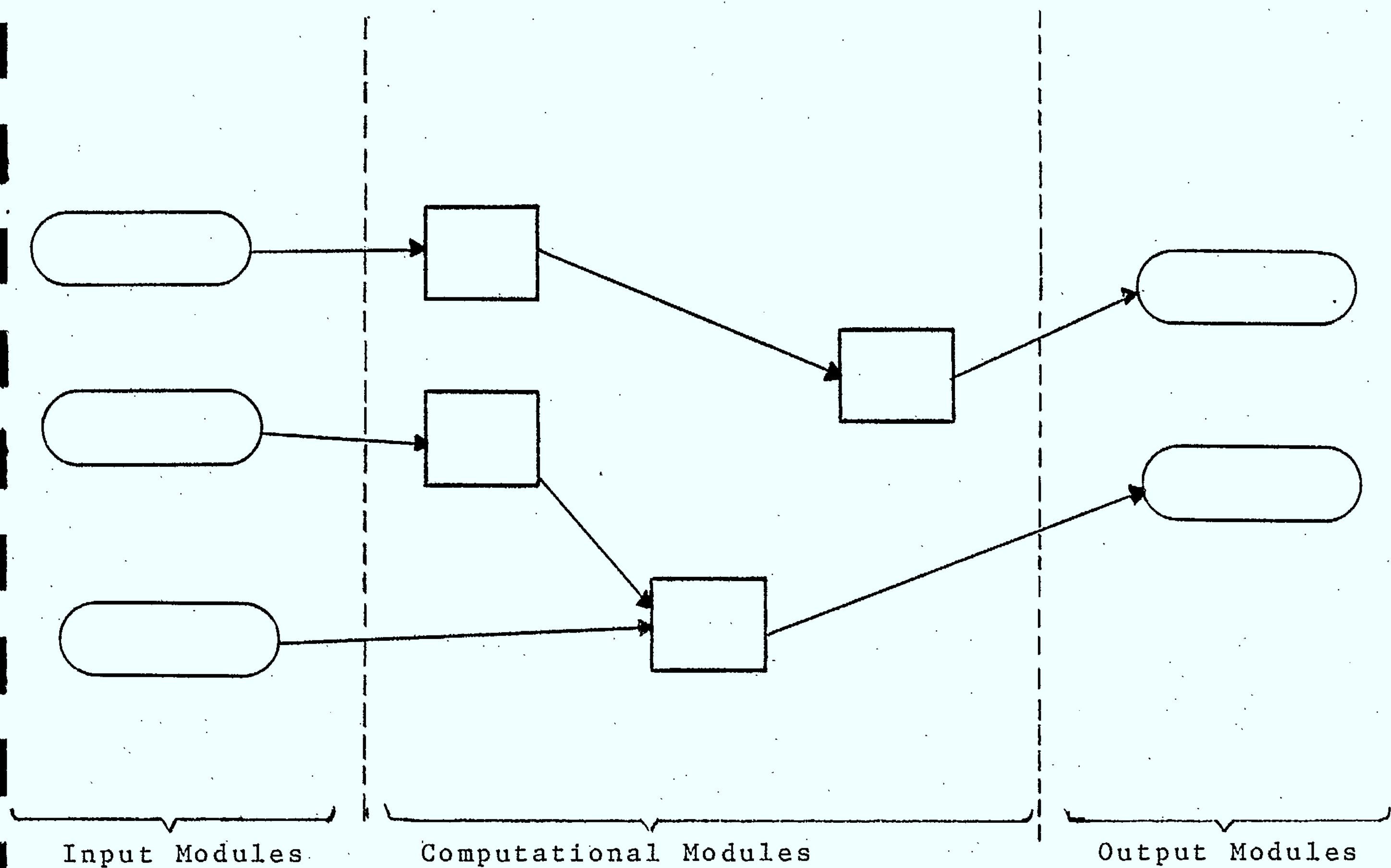


Figure 2.4
Example of a Data Flow Model

3. SPECIFICATION/VALIDATION

3.1 INTRODUCTION

3.1.1 Problem Definition

Traditionally, computing systems have been designed by a team of individuals. The design process was initiated with an effort to determine the requirements of the system to be built. The term "requirement" takes on different interpretation depending on the background of a given individual in the design team. Hardware and software induced misunderstandings abound in that early design phase. Nonetheless, what emerges is a list of items (often conflicting) depicting in detail what the system should do, how fast and how often it should do it and how reliable its performance should be.

The next phase is a mapping of the requirements, usually stated in a very informal fashion, to formal and complete specifications. This mapping is not trivial and, once done, may require validation; in other words, going back and checking that the formally specified system meets the requirements.

What follows is a series of decomposition or refinement steps on the original specifications which are normally at a very high level. Keeping in

mind that specification should lead to implementation, the high level specifications have to be translated into more manageable, lower level specifications. In this manner, a transition into a complete software description of the intended behaviour of the system is accomplished. At this point, validation may also be needed.

With the behaviour of the system formally described, a partitioning of some functions into hardware/software or dedicated hardwired controller can be attempted. The necessary guidelines to assist in this process are obtained through performance analysis and simulation.

A design methodology encompasses all the stages that have been described, namely: requirements, specification, refinement/decomposition, implementation, and performance analysis. This section is devoted to describing the high level specification activities and the ensuing series of refinement steps. The output of this process will be a formal description of the system's behaviour in a high level language; this description will also be in a form suitable for further processing by other levels.

3.1.2 Spacecraft Environment

A design methodology catering to general purpose environments would be very difficult to specify precisely. This is due to the multifarious nature of the tradeoffs involved in system design. Fortunately, working within a spacecraft environment allows for some assumptions to be made so as to restrict the scope of the methodology. The simplication thus achieved should significantly reduce the complexity of the methodology, and in particular of the specification/decomposition process.

A computer system aboard a spacecraft is, by definition, a dedicated controller. It overseas most of the current activities and may also be called upon to perform complicated computations. A basic representation of such a system is shown in Figure 3.1 where the input/output characterization of a spacecraft is illustrated. The controller has the capabilities to perceive the outside world through its sensors and to influence and to act

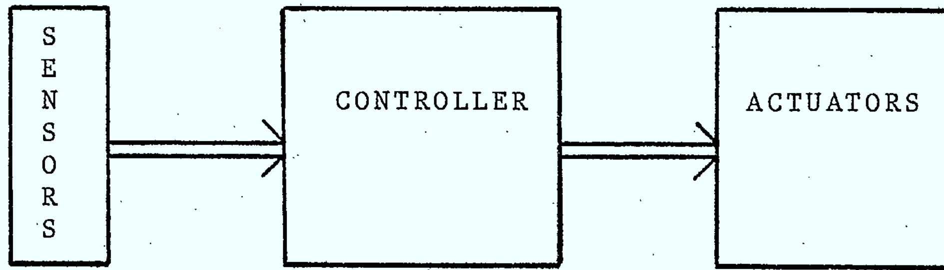


Figure 3.1: A Spacecraft Control System

upon the physical environment through the actuators. The attitude of an antenna, for example, can be sampled by some sensors and the controller (or computer system) can be made aware of it. Based upon stored directives or upon remote commands, the controller may decide to change the orientation of the antenna. The Actuators, (or servomotors) would then be used to effect this change on the physical environment.

There is yet another aspect to be considered: the real time nature of these actions. It may be required, for the sake of accuracy perhaps, that the sampling of the position of the antenna be done every milisecond. Similar constraints may also exist on how often and how fast the antenna can be moved. These observations lead to the conclusion that most tasks performed by onboard computer are periodic. The literature on such systems would seem to substantiate this view.

The set of requirements for an onboard controller would reflect the flow of data and its rate. The sensors would be characterized by an output data type, and other information such as average data rate and peak data rate. Similarly for the actuators, requirements concerning the input data type, the maximum permissible data rate, the minimum data rate, etc., would be given. The control tasks to be performed by the computer can be described by a transformation of various input types into some output types, according to some algorithm. These functions would also have time constraints imposed on them to determine their execution speed. These input, output, and processing constraints are easily expressed in a data flow framework.

3.1.3 Design Philosophy

As indicated before, the design process is a translation of informal requirements into formal specifications, followed by gradual refinement steps on these specifications. This stepwise refinement approach is analogous to a hierarchical system of machines and programs (as shown in Figure 3.2).

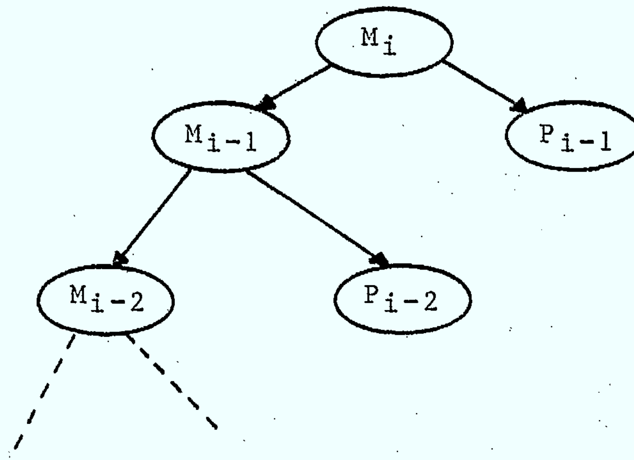


Figure 3.2: Hierarchy of machines and programs

In Figure 3.2, M_i is the highest machine. It would correspond to a machine which would perform all the system functions in one high level program instruction. Such a program, P_i, is a trivial program and is not shown. When considering the problem of designing a complex system, it may be advantageous to decompose the original design of M_i into the design of M_{i-1} and P_{i-1}. The machine M_{i-1} supports the computations performed by P_{i-1} and is the computational structure of the system to be

the computational structure of the system to be designed. The Program P_{i-1} is the computational behaviour of the system. The fact that M_{i-1} does not exist is of no concern; it can always be created by further decomposing M_{i-1} into M_{i-2} and P_{i-2}.

The methodology presented here, first assumes the existence of M_i. It decomposes M_i into M_{i-1} and P_{i-1}. Subsequently, P_{i-1} is specified using a high level language and leaving out lower level details. At that stage, machine M_{i-1} is assumed to exist. To proceed further, M_{i-1} is decomposed into M_{i-2} and P_{i-2}. Since all the instructions of P_{i-1} were directly executable by M_{i-1} (i.e., were written in M_{i-1}'s native language), it is easy to see that P_{i-2} is, in fact, a stepwise refinement on certain sections of P_{i-1}. At that level, the combination of P_{i-1} and P_{i-2} executing on M_{i-2} still corresponds to M_i, (as shown in Figure 3.3).

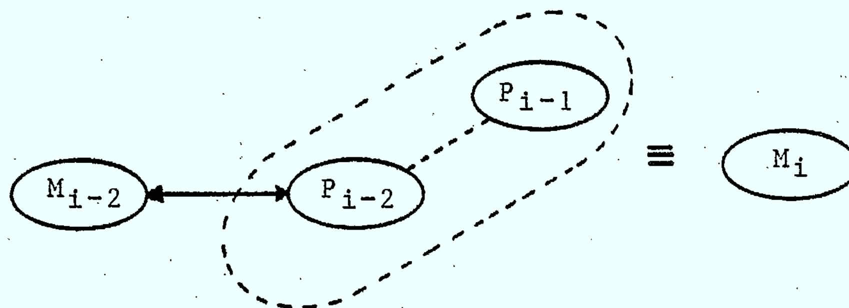


Figure 3.3: Machine equivalence

The methodology calls for this process to be applied iteratively so that a hierarchy of programs and machines is created.

The question of when to stop and implement physically a given machine (or parts of a machine) is not answered by this level of the methodology. The goal of the Specification/Decomposition process is to specify and refine down to a level where other tools such as performance evaluators can be applied. The results of such tools should help answer the implementation questions.

3.1.4 Overview of the Section

This section comprises two main parts: specification/decomposition and validation. In the first part, the concept of creating a hierarchy of programs is covered in details and an example of specification and decomposition is introduced. The second part addresses the problem of validation which was briefly mentioned in the introduction. A survey of validation techniques and of automated verification tools constitutes most of the subsection.

A summary of the section, together with concluding remarks will also be found at the end of the section.

3.2 Specification/Decomposition

3.2.1 High Level Specification

The first step of the methodology involves a translation of the requirements into very high level specifications. The nature of this translation process is not easy to document since it is mostly accomplished by systems analysts (i.e., humans) and relies on their intellectual capabilities. Human intervention in the translation process may introduce errors in this first attempt at specification. Substantial research activity has been generated, notably, [RAMA81] in which a dual design team, dual specification approach is advocated. The superiority of their technique has yet to be firmly established. Requirements definition has also been studied in [ROSS77a], and [ROSS77b] in which a structured analysis approach was proposed.

As mentioned earlier, the first specification attempt, as well as all others, embodies the hierarchical concept of machines and programs. It is important to realize that a machine program representation is isomorphic to a representation involving structure and behaviour. The latter type of representation has been used extensively in assisting hardware design endeavours [HILL79].

3.2.2 Methods of Decomposition

Stepwise decomposition is the series of activities that will transform the high level specification into an acceptable implementation. Although there are several methods to do this, only two of the most popular techniques will be described. (Also see [BERG81]).

1. Functional decomposition

The technique of functional decomposition involves a divide and refine approach. The problem is first considered as a whole and then divided into more manageable sub-problems. Those sub-problems can, in turn, be decomposed using the same technique. The result is a tree-like structure as shown in Figure 3.4.

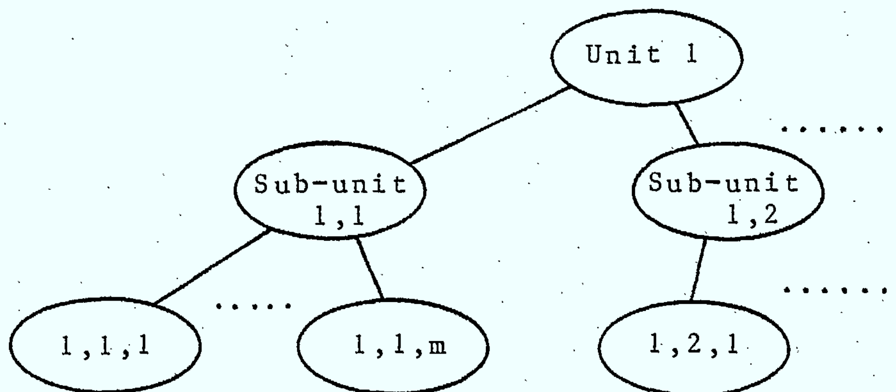


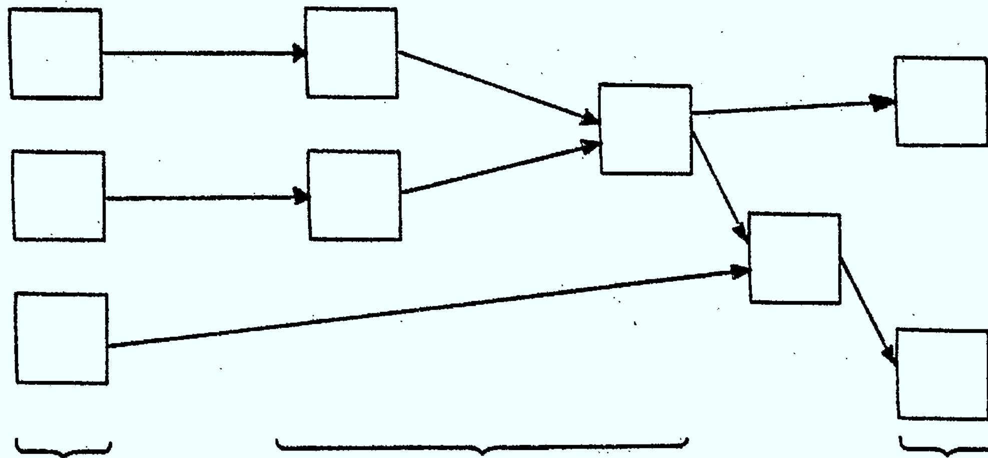
Figure 3.4: Tree structure resulting from Functional Decomposition

Functional decomposition, (as described in [DIJK76] and others), has been in use for a long time, and lends itself well to hierarchical development of the type proposed herein. The approach is not without difficulties however. One of the major problems associated with its use is the lack of similarity among independent decompositions of the same problem. Functional decomposition requires a concept (e.g., time, dataflow, groups of functions) with respect to which decomposition will be done. Lack of uniformity in choosing this concept causes the discrepancies previously mentioned.

2. Data Flow decomposition

An alternative to functional decomposition is data flow analysis. The problem to be solved is reduced to a flow problem in which afferent* modules collect various data and transmit these data to a network of computing modules. Those modules transform and alter the data and in so doing also change the flow. The end result, still in a flow form, is then given to the efferent modules for interfacing with the application. This method is described in Figure 3.5.

* In data flow terminology, afferent and efferent modules are meant to be input and output modules leading into or away from a network of computing elements.



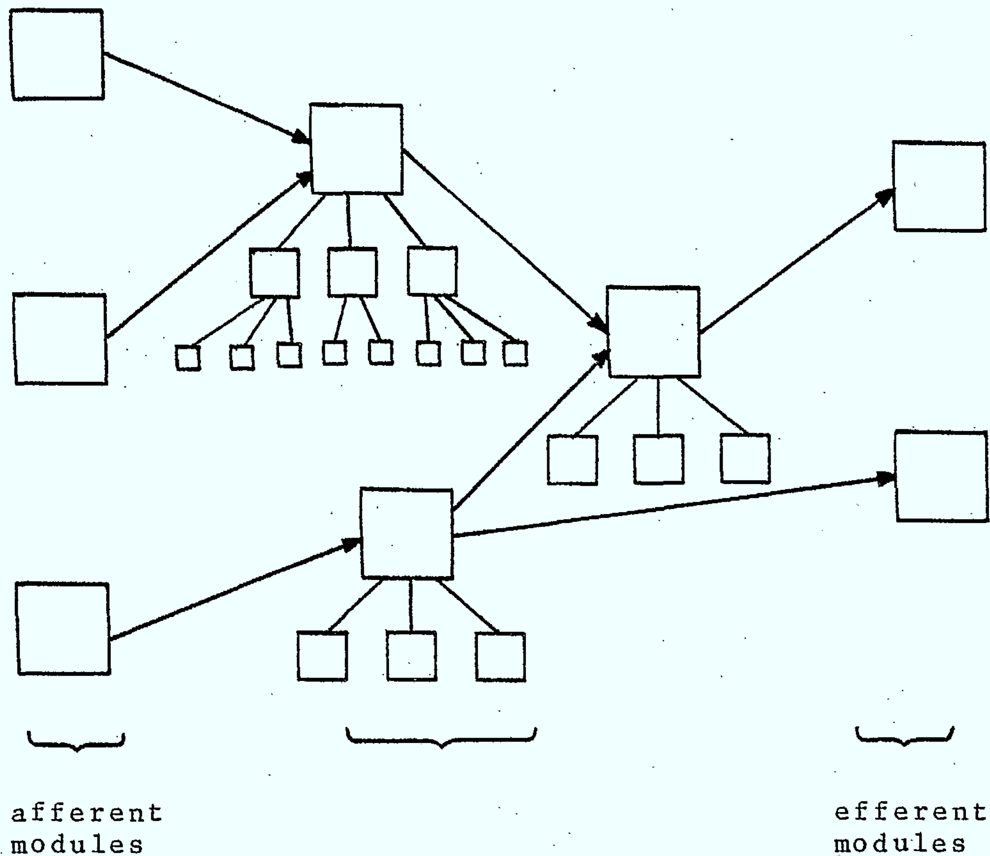
Afferent Modules   Network of Computational Modules   Efferent Modules

Figure 3.5: Data flow example

The data flow technique is well documented in [YOUR75]. It lends itself very well to the dynamic flow of data model of a spacecraft. It is, however, unwieldy to use at times, since some problems are not amenable to this functional decomposition with respect to data flow.

3. A mixed approach

In view of the affinity of the data flow analysis with the spacecraft design problem, this method was adopted, at least in the first few attempts at decomposition. To remedy some of the data flow analysis shortcomings, a functional decomposition method will also be used after the initial data flow decomposition. Figure 3.6 shows a hypothetical system decomposition carried out along those lines.



network of computing modules subjected to further functional decomposition

Figure 3.6: Result of the mixed approach

It is expected that this mixed approach will be able to capture the inherent data flow aspect of the role of an on-board controller and to help in creating the hierarchy of machines and programs which is essential to the general methodology. The mixed approach to decomposition will be used extensively in an example which will be worked out in detail in Section 3.2.5.

3.2.3 Formalizing the Specification/Decomposition Process

The specification and decomposition process requires formalism to establish a common frame of reference. This goal can be achieved by using either specification languages or procedural languages. Each option has its own merit and is adequate in describing the system.

1. Specification languages

   Specification or functional languages are non-procedural languages, often used for the purpose of specifications. HISP (Hierarchically Structured Specification Processor) [OKAD80] is such a language. HISP manipulates objects which are represented as:

   $$P = (Q, S, O, E)$$

   where P is an object, Q is a set of objects, S is a set of "sorts", O is a set of operators and E is a set of equations. In HISP, sorts are representation of data items relevant to the system. HISP also defines operations on objects (creation, construction, renaming, substitution, refinement) which allow a system to be completely specified. It is also possible to use a flowchart-like type of

approach as formalized in [ROSS77b]. Although not a language as such, this method is, in fact, a graphical representation of a functional language.

The advantages of using functional languages are as follows:

i) their semantics are easy to define,

ii) they lend themselves readily to expressing mathematically certain properties of the system. (In other words, proofs of correctness are facilitated.),

iii) they are not encumbered by lower level details which may detract from precise specifications.

The major difficulty associated with their use is that they eventually require translation to a procedural language like Ada* or Pascal.

*Ada is a trademark of the U.S. Department of Defense.

2. Procedural languages

The major complaint associated with the use of procedural languages is the presence of lower level details in the specification process. These details have been minimized to a great extent in newer languages such as Pascal and specially Ada. It is now quite feasible to use the data flow arcs of a high level "mixed approach" decomposition as basic data structures and to use procedures (either specified or stubs) to represent the functionality of the system. Stepwise refinement is, of course, possible on procedures not completely specified.

Using procedural languages for specification obviates the need for translation and imposes a strict formalism on the description of the system. It is always possible to introduce a mathematical model by means of assertions.

3.2.4 Ada as a specification tool

Ada is a programming language built to the specifications of the U.S. Department of Defense. It is a powerful language [WEGN80], [PYLE81],

[DOD80], [COMP81] with facilities for data typing, data and procedure encapsulating, support of concurrency, etc.

Some of those features are very helpful in specification/decomposition work:

1. Packages

A package is a module encapsulating data and a set of associated procedures. An Ada package is shown in Figure 3.7.

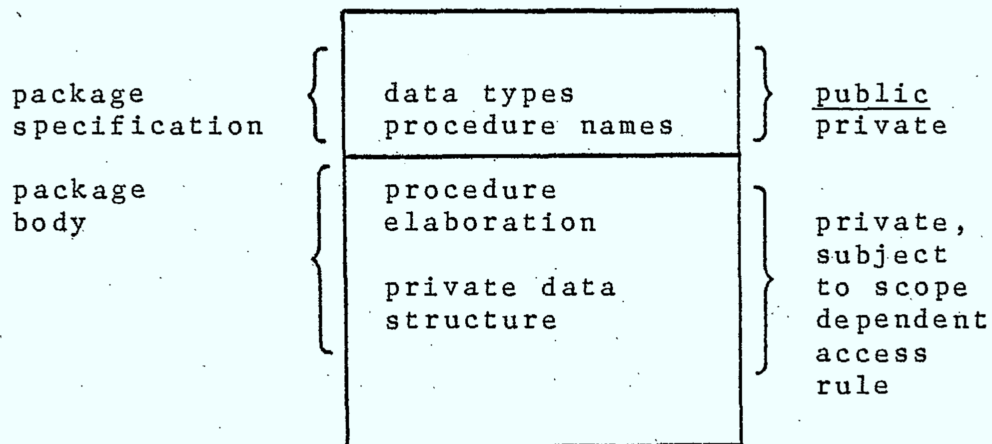


Figure 3.7: An Ada Package

A package comprises two parts: a specification in which the interface to the outside is defined and a body in which the actual processing is done. The body does not have to be completely coded in the early stages of the design; all is required is a complete specification part. The Ada compiler will make the necessary linking adjustments.

2. RendezVous

Ada supports concurrency and therefore has the necessary mechanisms to allow several tasks to execute in parallel. Concurrently executing tasks will often require some means of synchronizing their activities. To this end, Ada provides a RendezVous capability, graphically illustrated in Figure 3.8.

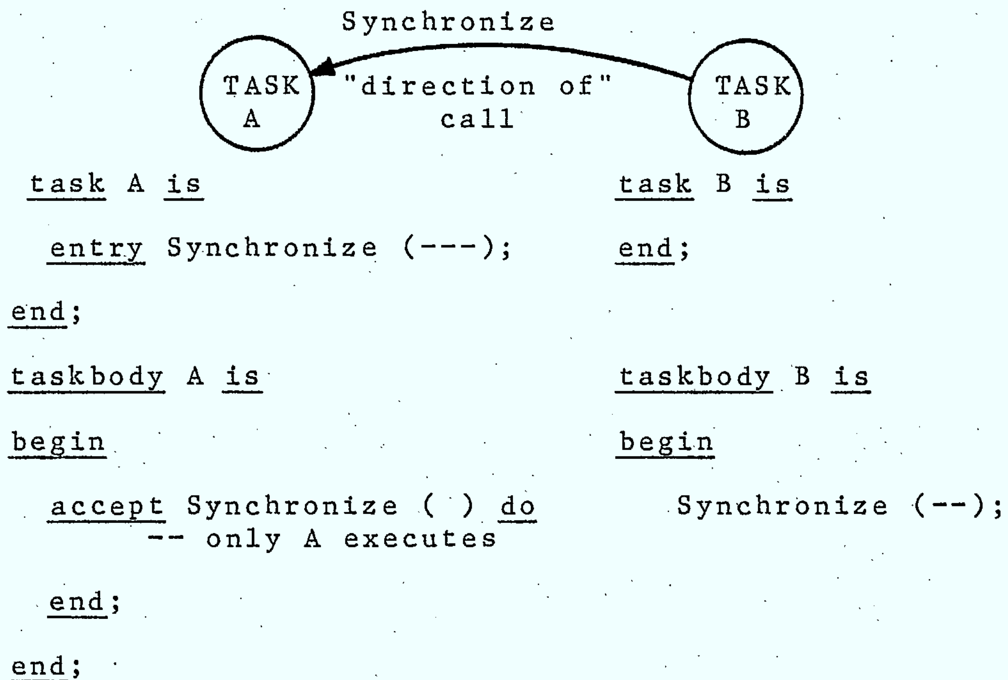


Figure 3.8: An Ada RendezVous

Task B is the active part in the RendezVous of Figure 3.8 and it calls task A. In Ada, tasks have the same specification and body structure than packages. In the specification part of a task, potential RendezVous are listed in the form of entries. A RendezVous is initiated

when either task B calls the procedure Synchronize or task A accepts it. If task B is first, it waits until task A accepts the RendezVous. If the reverse occurs, task A is the one waiting. The actual RendezVous takes place when both tasks A and B are ready. During RendezVous, the parallel executions of A and B will be reduced to a serial execution and at the end of the RendezVous, both tasks will resume their parallel execution.

3. Separate Compilation Units

There are extensive facilities in Ada for the support of separate compilation units. Separate compilation is possible for library units, package bodies and procedure bodies. Some of the rules governing separate compilation may seem at first a bit intricate, but this feature is extremely useful when attempting stepwise refinement during the decomposition phase. In fact, only the procedure or package interface need be specified initially. The body of a procedure, for example, can be left as a stub and refined later on.

When it comes to specifying and applying the "mixed approach" decomposition to a system, it can be very advantageous to use Ada as the specification language. As a procedural language, Ada would provide the formalism required of such an exercise. The translation from specification to implementation would be done in such a way as to result in code that could very often be of immediate use.

The end result of the "mixed approach" decomposition is a network of computing functions which were further subjected to functional decomposition. Despite its powerful features, Ada cannot be applied directly. What is needed is a construct which would facilitate the specifying of computing functions while at the same time allow for the representation of the functional decomposition process. Such a construct, called a "Specification Block", is shown in Figure 3.9 and is elaborated on next.

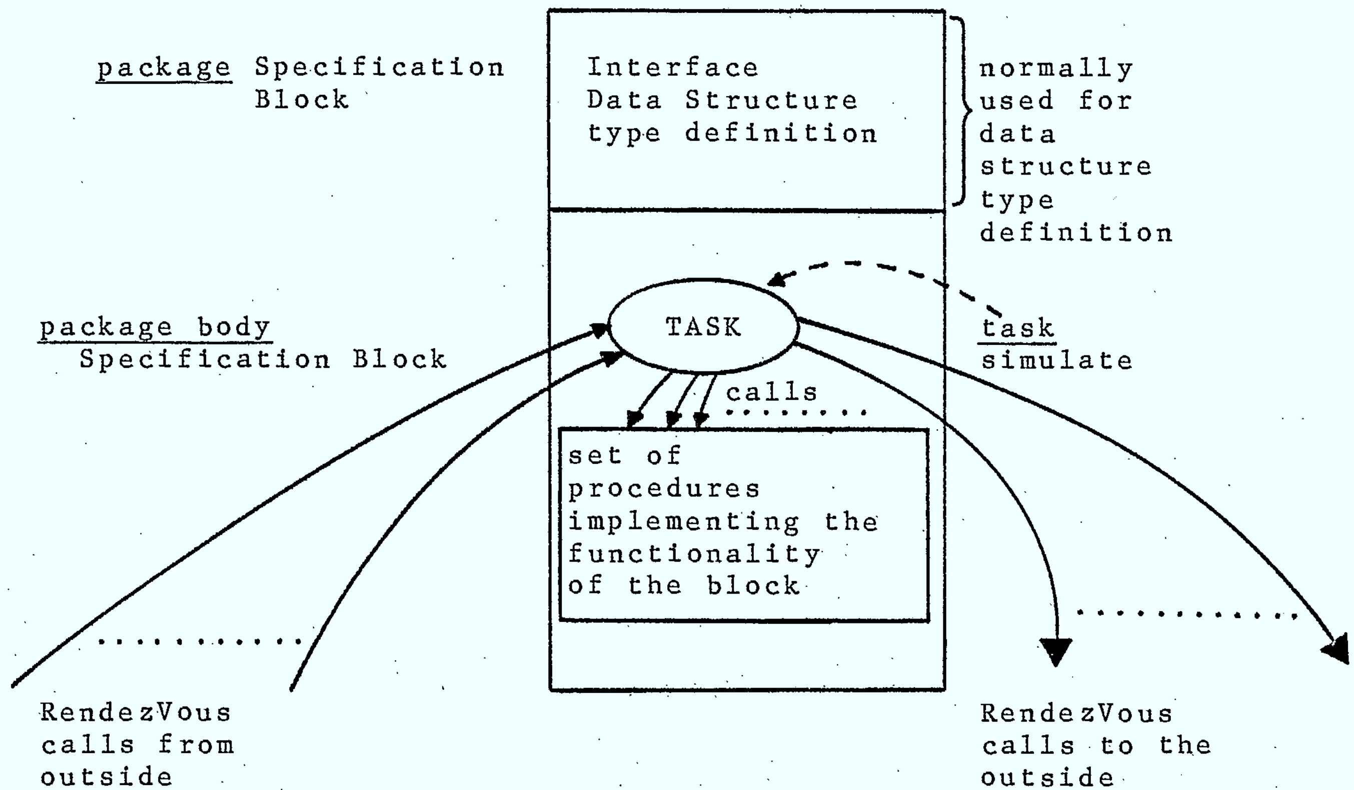


Figure 3.9: Basic Structure of a Specification Block

A specification block is expressed by means of a package; its interface is defined in the specification part of the package. Data structure types are elaborated at that level although similar results can be achieved in the specification part of the "Simulate" task.* The package body of the specification block contains the "simulate" task and

* For that reason, the simulation block package will be usually represented only by the package body.

a set of procedures implementing the functionality of the system. Those procedures will be the object of stepwise refinement. The "simulate" task fulfills two functions:

a) Through the RendezVous mechanism, it is linked to the data flow network of the decomposition model. The "simulate" tasks is called and calls other tasks, thereby emulating the flow of data of the model.

b) The "simulate" tasks embodies the algorithmic structure of a functional block. This embodiment is the result of the sequence of procedure calls the "simulate" task is going through. Functional refinement is made easier when dealing with such a construct.

The usefulness of the Ada specification block is not limited to the representation of computing functions. Even lower level devices can be accommodated by Ada, as shown in Figure 3.10, in which an interrupt driven device is interfaced to an Ada module.

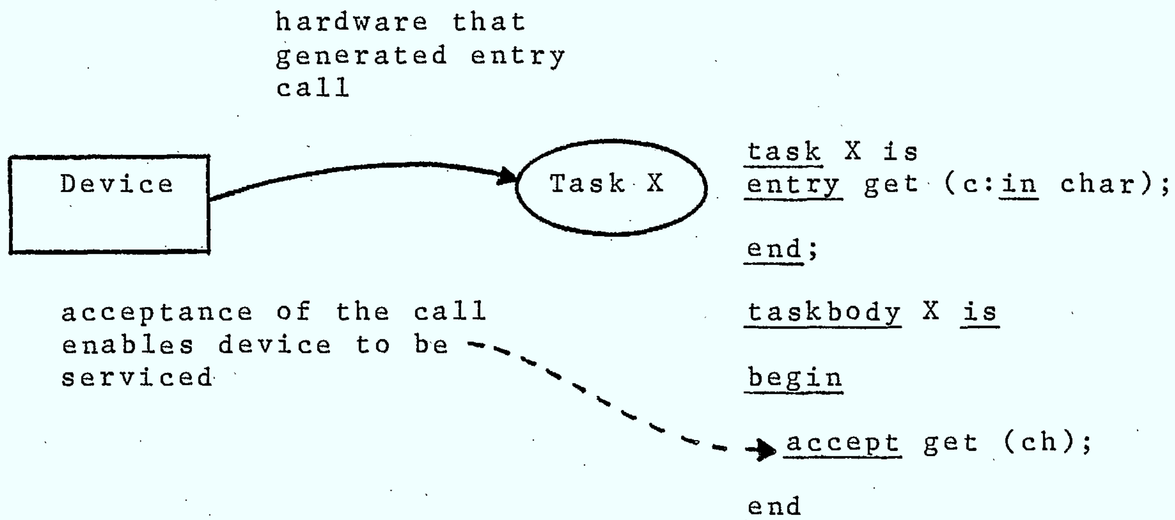


Figure 3.10: Device Servicing in Ada

It can be easily seen that, given a system decomposition, a translation can be made of all the functional modules to a network of Ada specification blocks. This process, although awkward and tedious to explain, is relatively straightforward. It is best illustrated by a thorough example of a small part of a system.

3.2.5 A Decomposition Example

In this example, the decomposition process will be applied to an antenna attitude control module. The example, shown in Figure 3.11, consists of a remote controller for the positioning of a spacecraft antenna. The on-board controller is made aware of the actual position of the antenna through sensors so as to implement a closed loop control. To complement the sensors, a television camera with digital output is also provided.

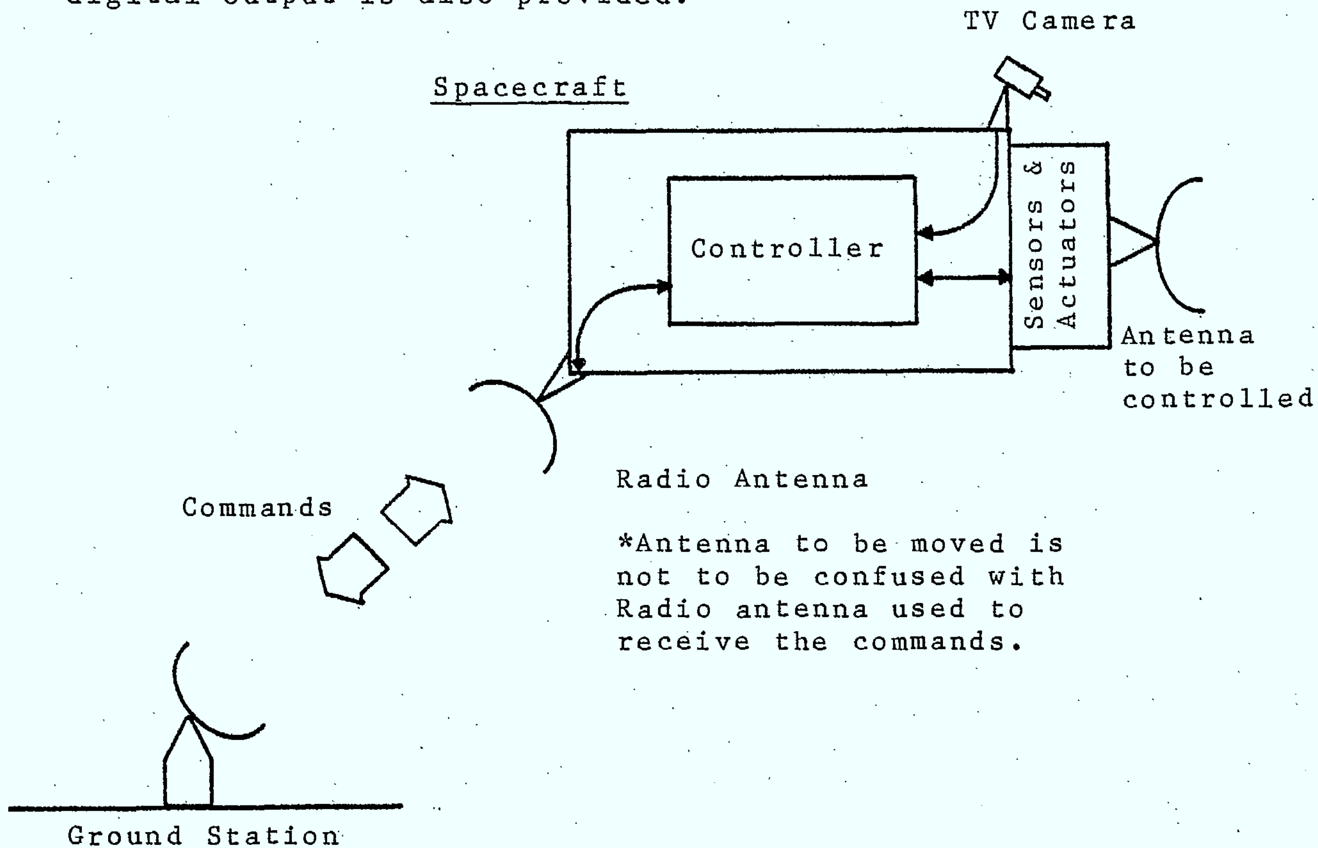


Figure 3.11: Description of the example

3.2.5.1 Various decomposition levels

The first attempt at decomposition is shown in Figure 3.12. Three types of modules are visible in that figure:

1- Input modules

InRadio, InSensors, InTv

2- Output modules

OutRadio, OutActuators, Outcalib

3- Functional modules

Antenna attitude control module

The functional module can also be further decomposed as shown in Figure 3.13. In that figure, the data flow model will be the last level of such decomposition. The functions of each module are outlined below, in preparation for functional decomposition (NOTE: numbers correspond to those of Figure 3.13.):

1- InRadio is the input module dedicated to the radio receiver. It can be thought of as a device handler.

2- InSensor is the input module dedicated to the X,Y,Z co-ordinate sensors.

3- InTv is the input module dedicated to the TV camera.

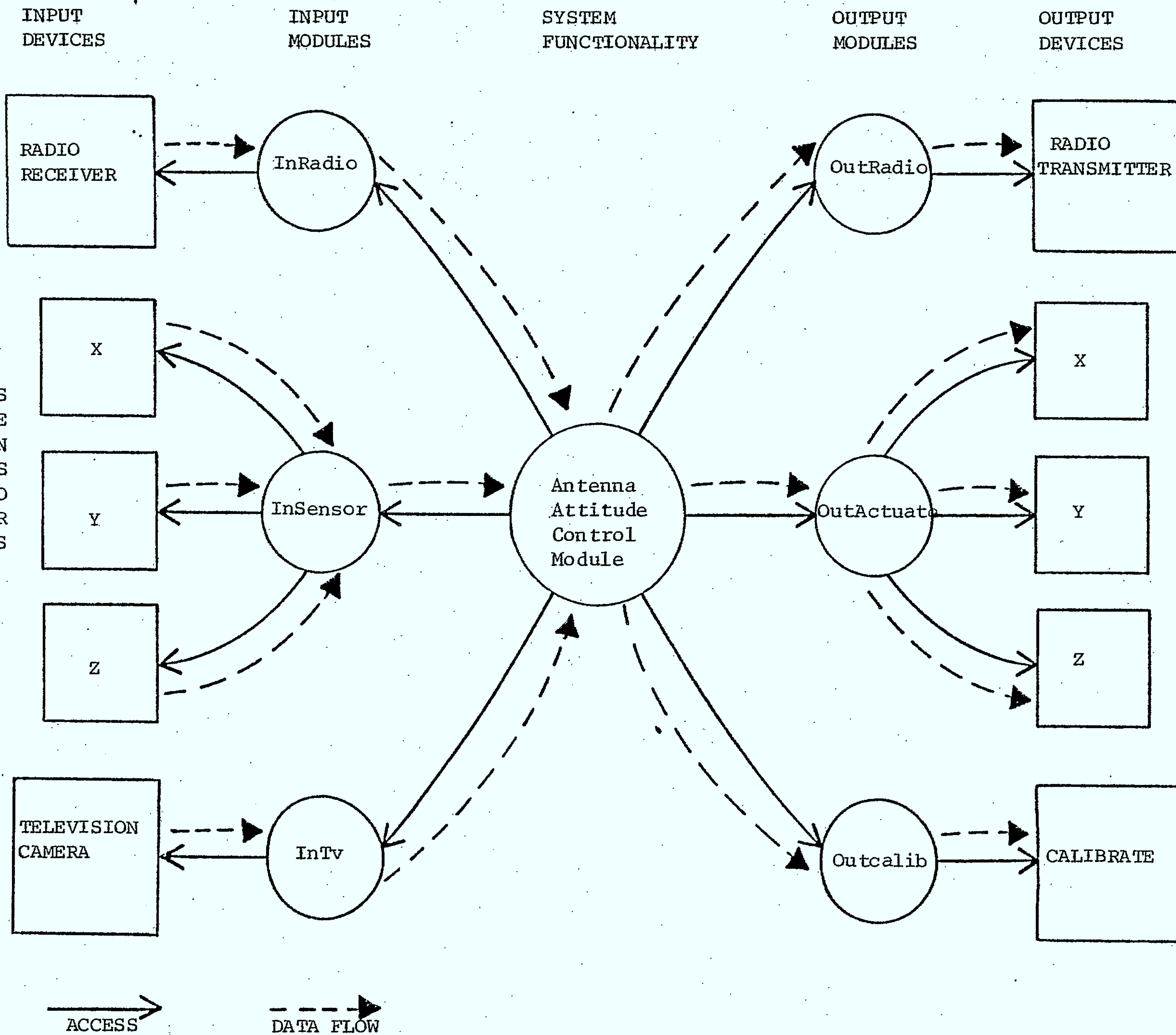


Figure 3.12: A first attempt at decomposition

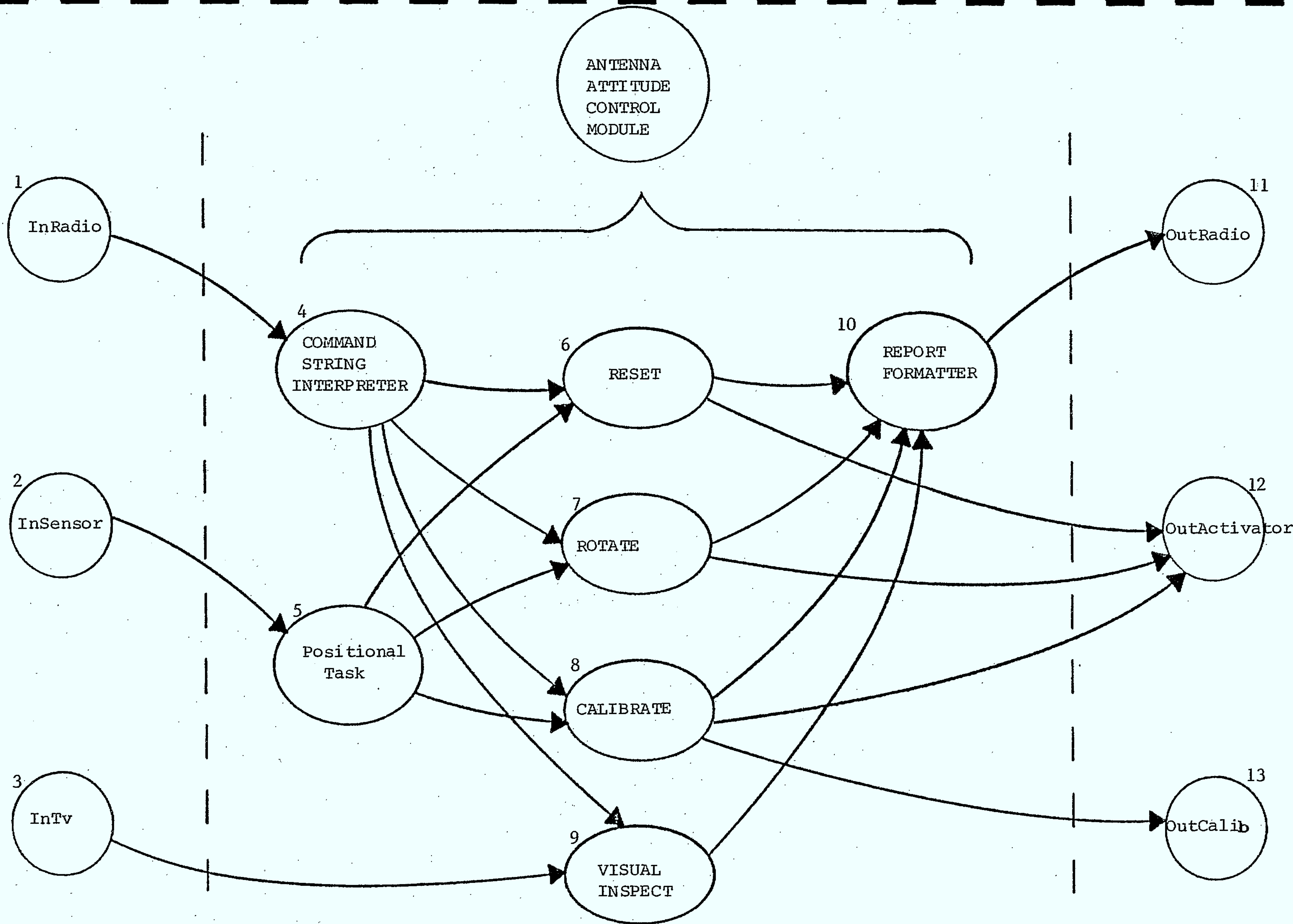


Figure 3.13: Data Flow Graph

Devices not Shown

*numbers refer to description

4- Command String Interpreter receives commands from the ground station. The commands are structured as shown in Figure 3.14.

```
type CommandString is
    record
       CommandType:  (VisualInspect, Reset, Rotate
                      Calibrate);
       case CommandType is

       when Rotate => Desired Position:  Position;
       when Calibrate => Desired Accuracy: Accuracy;
       when VisualInspect => Magnification:Integer;
                            Switch:  (on,off)
       end case;
       end record;
```

Figure 3.14: Definition of Various Commands

The Command String Interpreter will activate the desired modules (and also pass parameters when applicable) based upon the value of the Command type contained in the Command String.

5- PositionTask receives the X,Y,Z position of the antenna. Those values are the result of sampling done at regular intervals. The PositionTask also maintains an up-to-date table of statistics on the dynamics of the antenna.

6- Reset resets the antenna position to a given initial point. It then sends confirmation to the ground station, to indicate the status (success of failure) of the command.

7- Rotate rotates the antenna to a position specified in the command parameters. An indication of the success or failure of the command is also sent back to the ground station.

8- Calibrate moves the antenna to a known position and re-calibrates the sensors through the use of the Calibrate module. A Report is sent back to the ground station.

9- VisualInspect starts the TV camera and transmit back to the ground station digitized pictures. A VisualOff parameter stops this process.

10- ReportFormatter accepts various types of input data, (e.g., confirmation of successful rotation, result of calibration, digitized television pictures, etc.) and prepares them for transmission.

11- OutRadio is the output module for the radio transmitter. It is, in fact, a device handler.

12- OutActuator is the output module which takes care of the servo control mechanisms of the antenna.

13- OutCalib is the output module dedicated to the Calibrator device.

At this point, it should be noted that, although Input and Output modules have always been included, their presence is not mandatory. In actual fact, practical considerations may dictate that they be incorporated in other modules. However, the reason for their being separate entities is that the functions they implement have to exist in most cases. It is reasoned that it is easy to merge them with other functions and, besides, they add to the clarity of the decomposition exercise.

Following two stages of decomposition with respect to data flow, the system is subjected to a

functional decomposition. The result is shown in Figure 3.15. The various modules are broken down with respect to the functions they are supposed to implement. This functional decomposition is illustrated by the tree structures subtending the data flow modules. The decomposition process is also shown in various stages of completion and it should be remembered that it does not necessarily correspond to reality since its prime purpose is didactic.

In the complete design methodology, the functional decomposition, as exemplified by the diagram of Figure 3.15, is a transitory step. Its main purpose is to provide a bridge to a system representation using the Ada specification blocks. This system representation is to be found in Figure 3.16.

There are several differences between the system representations of Figures 3.15 and 3.16. The most noticeable is, of course, the use of a procedural language to describe the system. The formalism of Ada helps in solidifying the description of the system and of its various characteristics. Another extremely important point

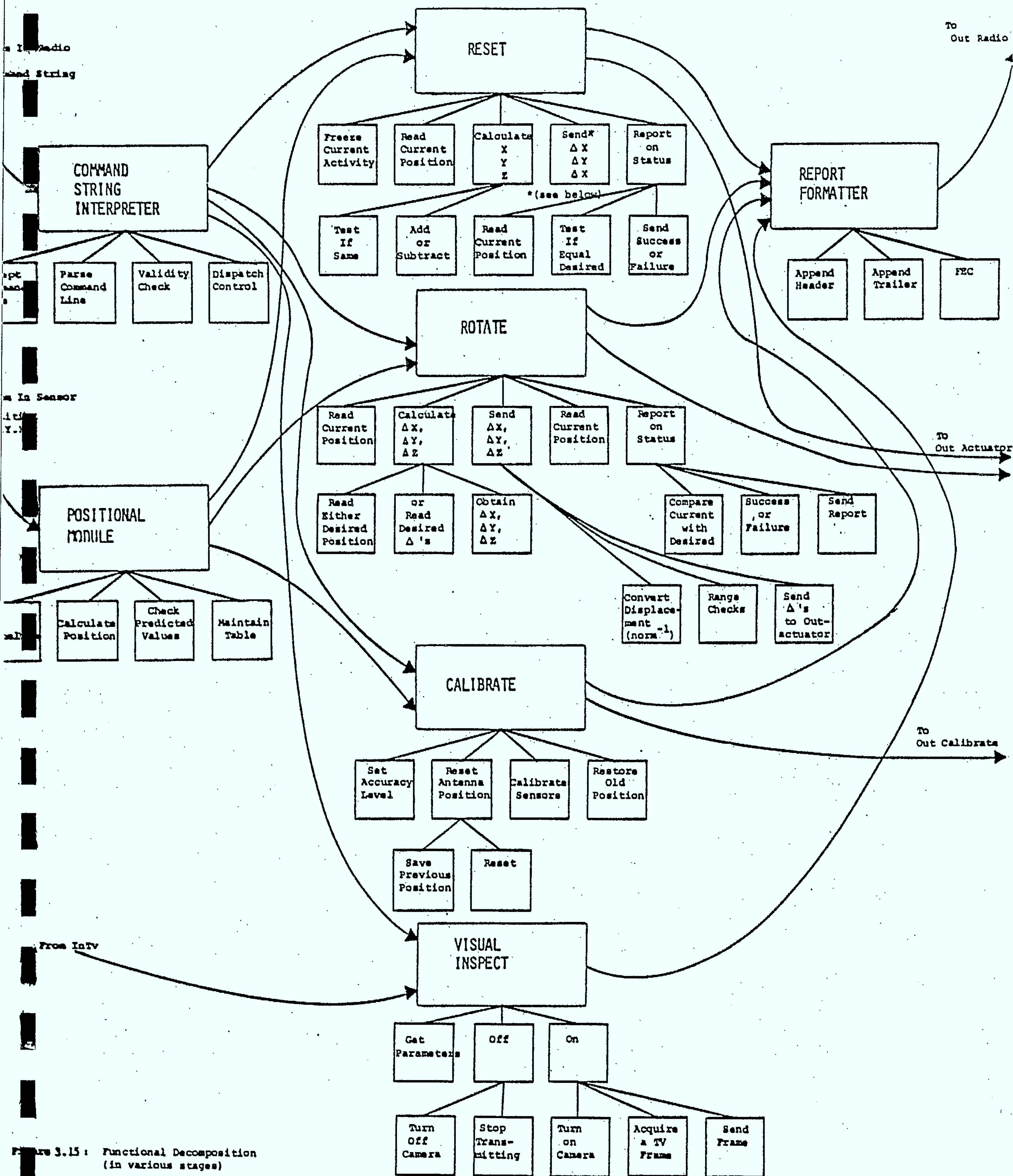


Figure 3.15 : Functional Decomposition (in various stages)

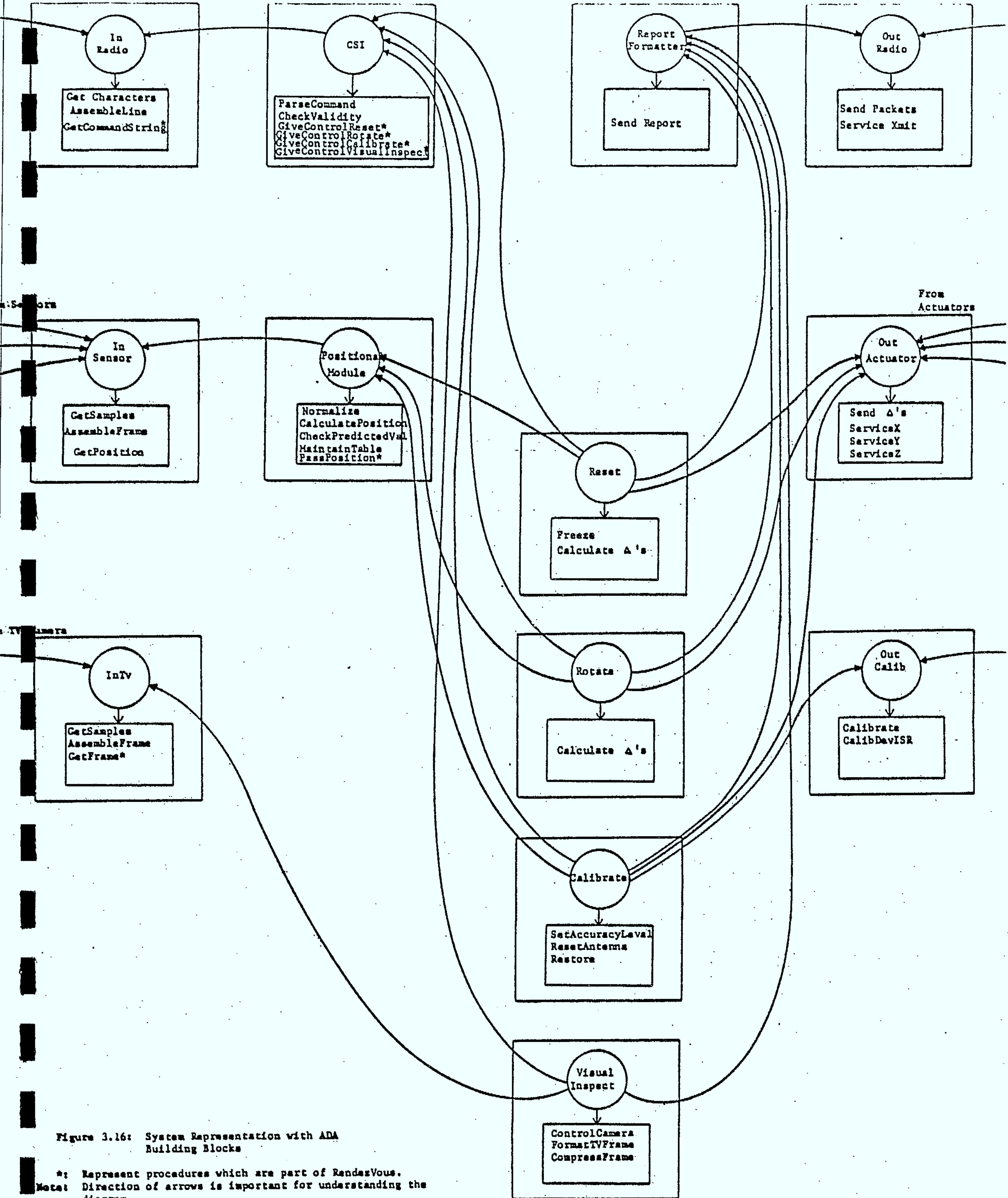


Figure 3.16: System Representation with ADA Building Blocks

*: Represent procedures which are part of RendezVous.
Note: Direction of arrows is important for understanding the diagram.

is the disappearance of data flow arcs and their replacement by procedure call arcs. It should be noted that the direction of a procedure call arc is not necessarily indicative of the direction of the flow of parameters, if and when parameters are passed. In Ada, the direction of a procedure call arc has a further significance beyond determining the calling and called parties. In fact, it plays a role in establishing how control is passed, and if not properly set up, may become the proximate cause of deadlock. Considerations on how the direction of those arcs affects deadlocks will be found in Section 3.2.5.3.

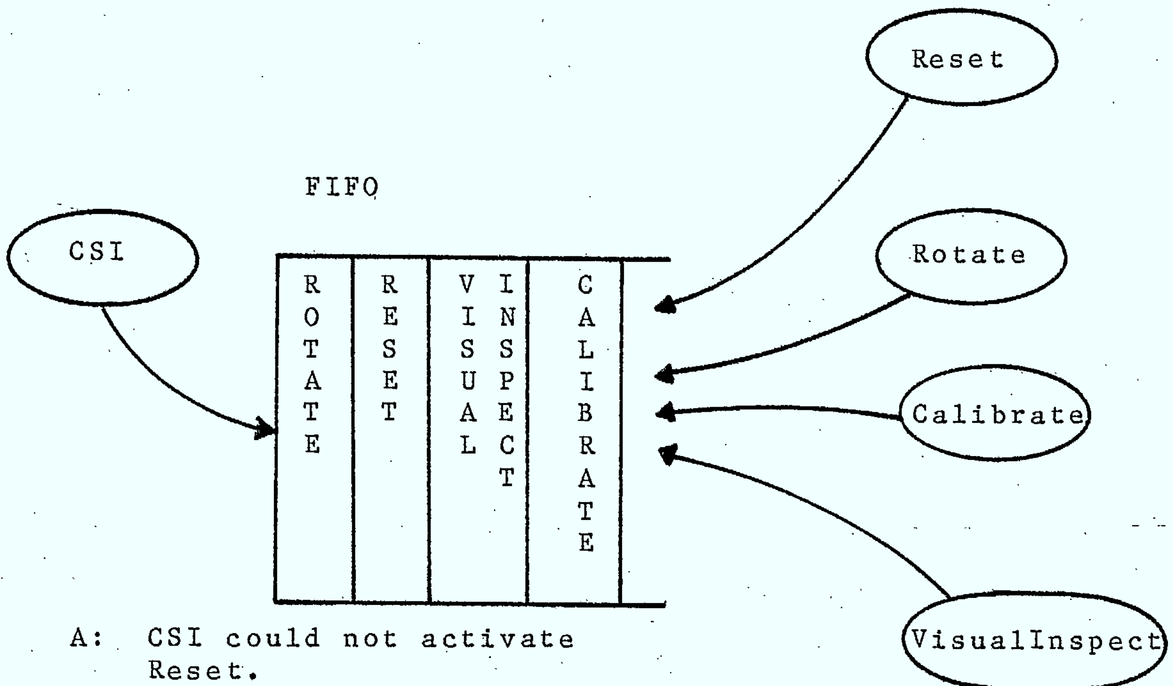
Other aspects of Figure 3.16 are worth mentioning briefly and are listed below:

i) There is direct linking to devices. Hardware generated interrupts are interpreted as procedure calls into one of the input or output modules.

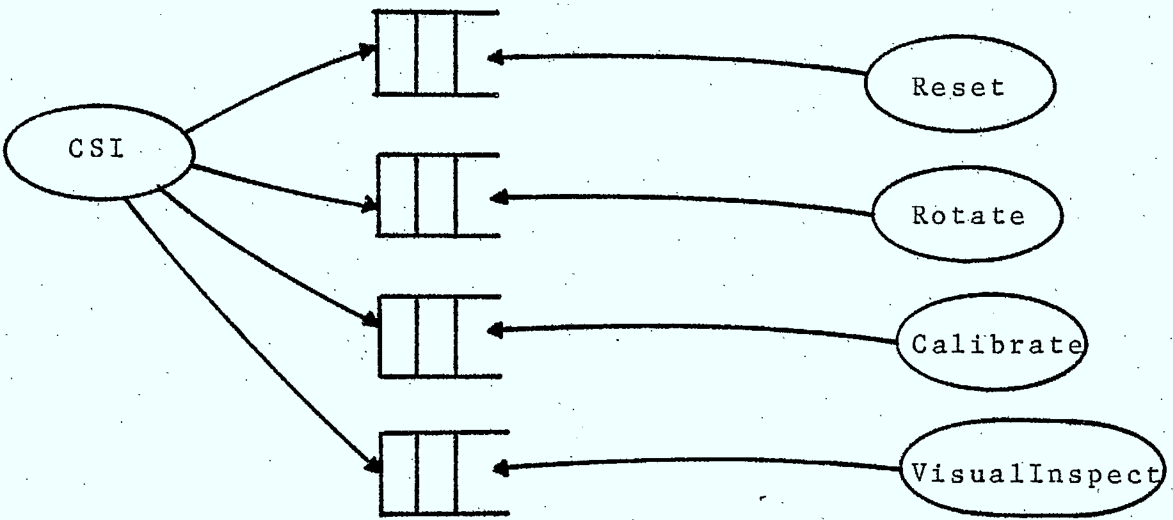
ii) The control structure of the Rendez Vous mechanism in Ada has been examined earlier. Based upon that knowledge, it is interesting to observe that, for a procedure call from a main functional module to one of the input/output modules, the body of that procedure will be in the called module.

iii) The structure of the RendezVous, [SILB8], [MAHJ81], [LAFE81], while generally very flexible, is sometimes awkward to manipulate, especially in cases where control is to be dispatched to other modules. Two cases should be considered:

Case 1: The Command String Interpreter (CSI) passes control to one of the Reset, Rotate, Calibrate, or VisualInspect modules depending on the Command String, the CSI got from the InRadio module. The procedure call arcs afferent to the CSI module are distinct, (i.e., Give Control Reset, Give Control Rotate, Give Control Calibrate, Give Control Visual Inspect). Were they not distinct, the CSI module would not be able to activate the desired module. This situation is shown graphically in Figure 3.17 (a & b).



A: CSI could not activate Reset.



B: The control structure is more intricate but allows flexibility.

Figure 3.17 (a & b): Ada Rendez Vous and Passing of Control

Case 2: Calls to the Positional module from other modules (in order to get co-ordinates) are not distinct. In this case, the need for distinct procedures is obviated by the structure of the system. The calling modules, (Reset, Rotate and Calibrate) will normally be waiting for the CSI module to give them control. Only one of them will execute at any one time and, as such, no contention is present, hence no need for separate calls.

Further functional decomposition can be carried out easily with the system model of Figure 3.16. This will be demonstrated by specifying a subset of modules such as the CommandStringInterpreter and the Reset modules. To keep the example simple, the packaging of those modules will not be shown; Ada manuals [DOD80], [WEGN80], [PYLE81], describe how to build packages quite well. The description will proceed from tasks to procedure stubs. The reader is also referred to Figure 3.14 which contains a full type definition of a command string.

The first task to be specified is the Command String Interpreter task which interfaces with most of the tasks in the system. The CSI task is shown in Figure 3.18, with the function "GiveCommandString" specified as separate. Figure 3.19 describes how the link up to the CSI module will be done. The Reset module is of greater complexity than CSI and, as such, makes an interesting example. It is shown in Figure 3.20. No entry specifications are found in task Reset since it is the active party in RendezVous all the time.

```
task CSI is
  entry GiveControlReset (CS: out CommandString);
  entry GiveControlRotate (CS: out CommandString);
  entry GiveControlCalibrate (CS: out CommandString);
  entry GiveControlVisualInspect (CS: out CommandString);
end CSI;

task body CSI is
  LN: CommandString
  function GiveCommandString (Y:CommandString) return
           CommandString is separate


begin
   loop
       GetCommandString (LN);
       case LN.CommandType is

           when Reset => accept GiveControlReset (X:outCommandString)do
                                X:=GiveCommandString(LN);
                         end GiveCommandReset;

           when Rotate => accept GiveControlRotate(X:outCommandString)do
                                 X:=GiveCommandString(LN);
                          end GiveControlRotate;

           when Calibrate => accept GiveControlCalibrate(X:outCommandString)do
                                    X:=GiveCommandString(LN);
                             end GiveControlCalibrate;

           when VisualInspect => accept GiveControlVisualInspect(X:outCommandString)do
                                        X:=GiveCommandString(LN);
                                 end GiveControlVisualInspect


       end case;
   end loop;
end CSI;
```

NOTE:
It should be realized that in this case the function GiveCommandString is redundant. It was included to show how, if such a function became necessary, link up is possible either to inside or outside the package.

Figure 3.18 Command String Interpeter (CSI)

```
separate(CommandStringInterpreter);

function GiveCommandString(Y:CommandString)returnCommandString is
  Test:boolean;
function CheckValidityOf(X:CommandString)returnboolean is separate;

begin
     Test:=CheckValidityOf(Y);
     if Test
     then return Y;
     else raise "exception";
     end if;
end GiveCommandString;
```

Figure 3.19: Expansion of a separate procedure

```
task Reset;

task body Reset is
  CS: CommandString;
  CST: CommandStatus;
  CP: CurrentPosition;
  Mssg: Message:
  DV: DisplacementValue;
  InitialPosition: Position:=0,0,0;

  procedure GiveControlReset(X:CommandString) is separate;
  procedure GetPosition(X:Position) is separate;
  procedure NewPositionDisplacement(X,Y:Position) is separate;
  procedure MoveAntennaPosition(X:DisplacementValue;Y:CommandStatus) is separate;
  procedure SendMessage(X:Message) is separate;
  procedure AbortSystemActivities is separate;

begin
   loop
       GetCommandString(CS);          -- part of RendezVous
       AbortSystemActivities;
       GetPosition(CP);               -- part of RendezVous
       DV:=NewPositionDisplacement(CP,InitialPosition);
       MoveAntennaPosition(DV,CST);
       if CST=Success
       then Mssg:=Success;
       else Mssg:=Failure;
            raise actuator failure;
       end if;
       SendMessage(Mssg);
   end loop;
end Reset;
```

Figure 3.20: Description of the Reset Module

Task Reset also exemplifies the use of separate procedures. Those procedures can either be in the package itself or in a different package which may be compiled separately. In the latter case, many stages of refinement may have taken place; proper procedure specification ensured that the rest of the model was not affected by changes in the separate procedure body.(*)

3.2.5.2 Observations on the Model

The use of the decomposition methodology yields a system model which exhibits several desirable properties. The most important advantage is that the system is described formally as opposed to an informal description whose meaning may be subjected to misinterpretation. The model is also easy to visualize. Procedures and functions which serve similar purpose may be grouped together in anticipation of a complete software design. Such grouping of subprograms would optimize the coding of the system tasks. It should be realized, however, that the system tasks obtained through decomposition are not necessarily optimal nor are they the only set of tasks which could be obtained. Different designers or analysts will more than likely arrive at different task decomposition.

(*) The Ada Manual [DOD80] has a lot more on that issue.

The Ada representation of the model makes an early compilation possible. This compilation, although not intended to produce executable code, can check the validity and consistency of data types and most importantly of procedure calls. The network of Ada building blocks can also be incorporated in a testbed, early in the design phase. This will be covered shortly in more detail.

The importance of properly designing the model with respect to procedure call arcs was stressed before; a few guidelines will now be provided. Three cases are of interest and are shown in Figure 3.21.

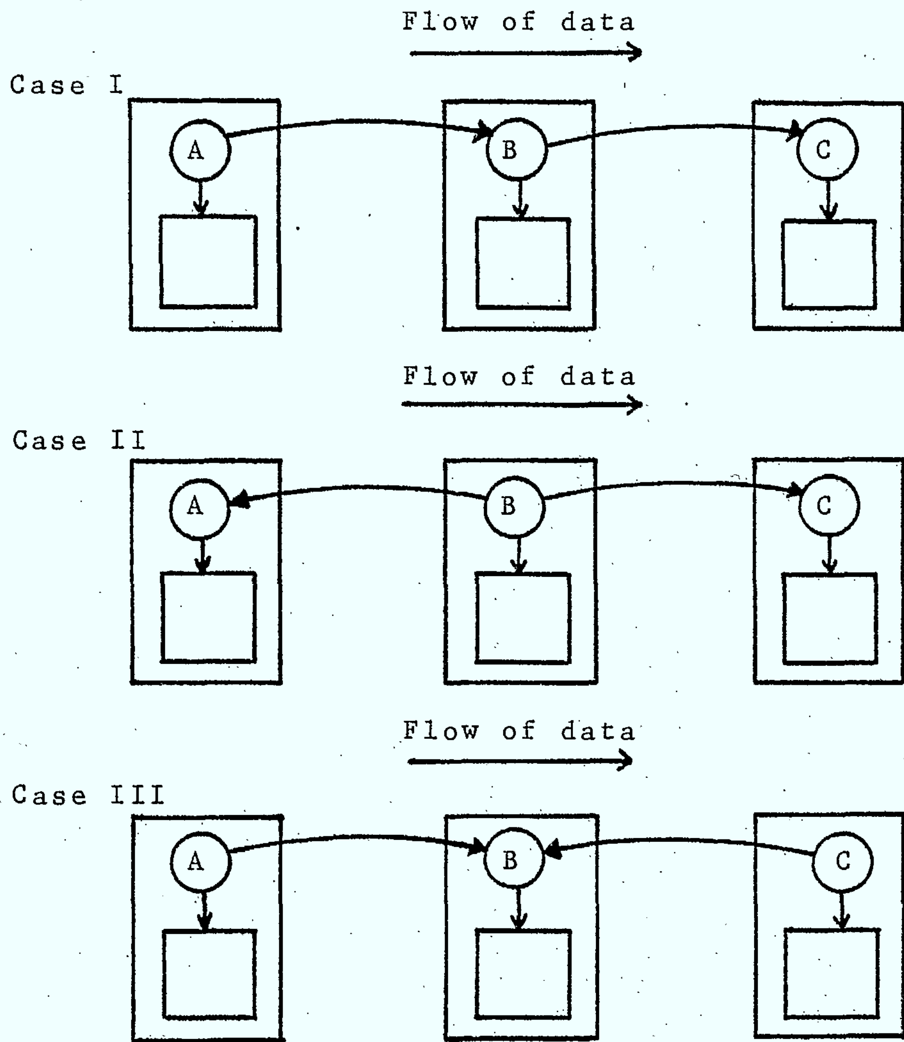


Figure 3.21: Procedure call arrangements in Ada

Case I is the most likely situation to be encountered. Task A calls Task B and passes parameters to the latter. The same process takes place again for B and C. The direction of the data flow also coincides with that of the calls. This case of innocuous appearance can be a potential cause of deadlock if a cycle is allowed to form, (e.g., A calls B and is delayed because B called C and B was delayed because C had called A and had been delayed, etc.)

Case II is introduced to lessen the possibilities of deadlock when two tasks (e.g., A & C) have to communicate with each other but, at the same time, cannot afford to wait on one another. The reason for this unwillingness to wait may be that both A and C are tasks offering general purpose services to other tasks. It should also be noted that agent task B will convey the parameters of the call from A to C only. If a reverse flow is desired, another agent task is required.

Deadlocks are not likely to occur in this arrangement because task B will be alternately waiting on A to get a message, and on C to give the message. The actions involved in each case are very short and

should hinder neither A nor C, as task B is doing the waiting. It is easily seen that the overhead in terms of executive services (e.g., context switching) may be substantial for this configuration.

Case III also lessens the possibilities of deadlock and is used in cases such as:

a) An input (or output) module which has to service hardware interrupts while at the same time making the data available to other tasks in the system.

b) A module which provides various services to other tasks, and as such, cannot be delayed.

An example of an arrangement typifying Case III is shown in Figure 3.22 which depicts an input module servicing a hardware device. The input module makes the data that it collected available to the processing task.

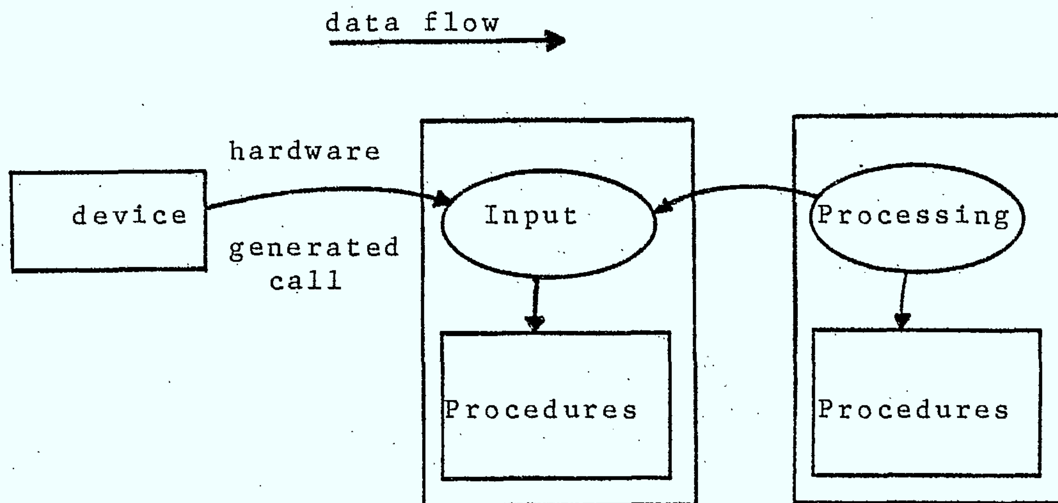


Figure 3.22: Representation of an Input Module

The reason behind the call structure of Figure 3.22 is that the Input task should always (ideally) be able to schedule a hardware generated entry call from the device. If that is not the case, a loss of input data will result, caused by either too fast a device or an inadequate call structure. The proposed structure, however, affords the Input task the best potential for quick scheduling of the service requests from the device.

Case II and Case III help design systems with lower probability of deadlock. They do not completely eliminate the risk of deadlock. Total elimination of deadlock can only be achieved by a careful design and, if need be, some simulation to increase the confidence in the operations of the system.

3.2.5.3 System Simulation and Testing

Preliminary simulation and testing can be attempted on a model which is completely (or even only partially) specified by Ada specification blocks. Figure 3.23 shows the modelled system as a black box surrounded by device simulator modules.

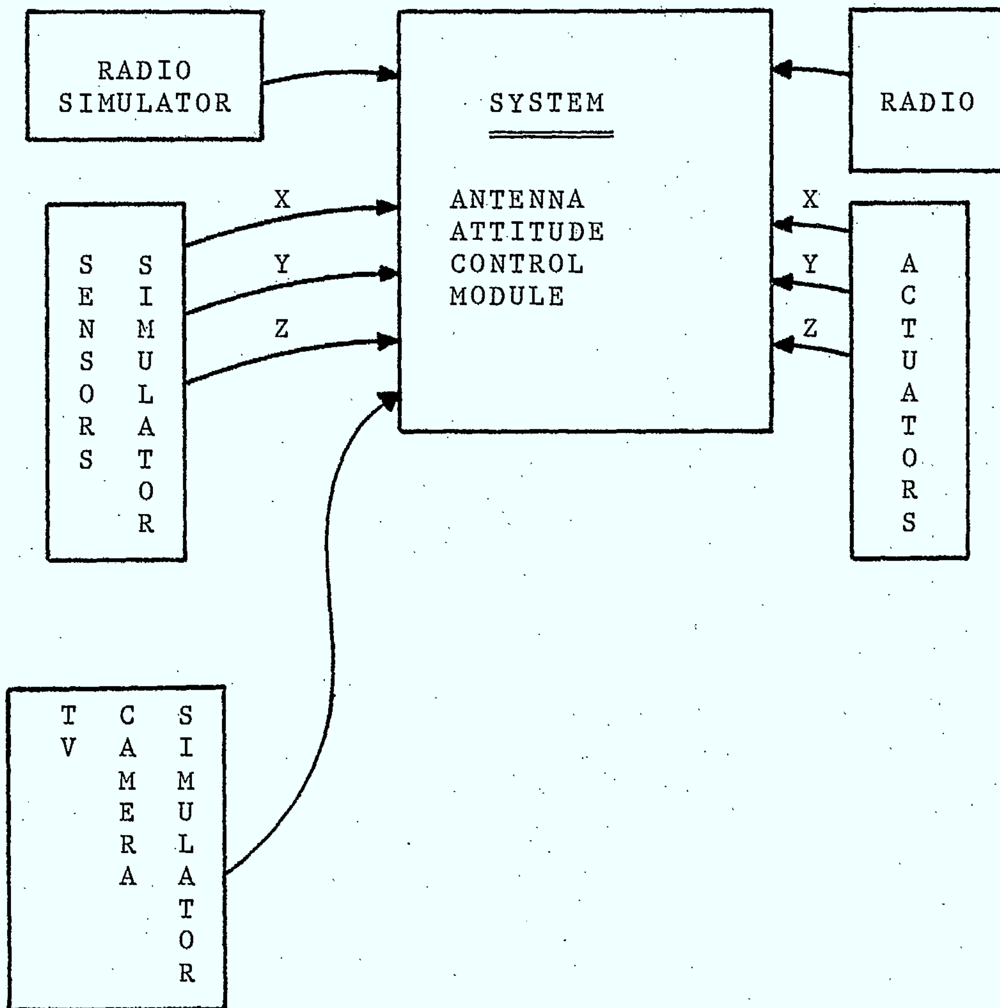
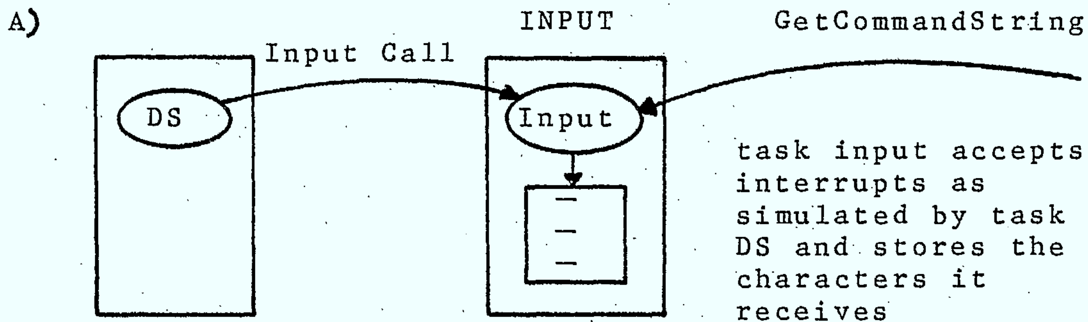


Figure 3.23: System Model and Test Bed

Those device simulator modules are outside the model and serve to simulate the behaviour of the devices they represent. Such a device simulator is shown in Figure 3.24 (a & b).

A)




```
package DeviceSimulator
```

B)

```
task DS ;
task body DS is
ch: character; DL:=Delaytime;
begin
   loop
       ch:=SelectCharacter;
       Input(Ch);                --functions SelectCharacters
       DL:=SelectDelay;          --and SelectDelay implement
       delay DL;                 --the behaviour of
   end loop;                     --the device
end;
```

Figure 3.24: Device Simulator Module

The complete specification/simulation/testing process is shown in Figure 3.25. The specification block is a manual operation; the system model it produces is the main input to the simulation and testing package. The Device simulators are also given to the simulator package.

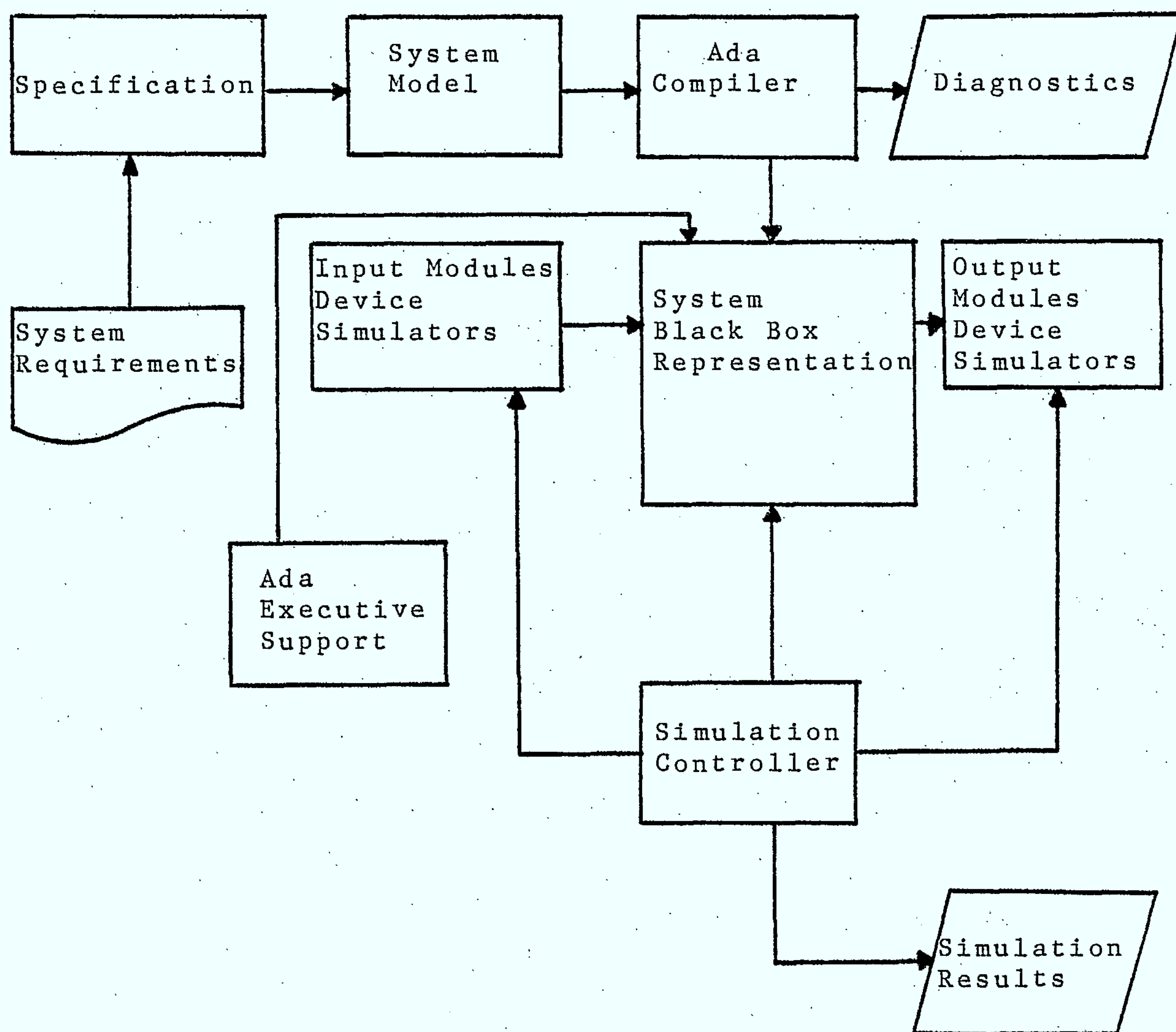


Figure 3.25: Simulation & Testing Package

Several steps are necessary to complete the simulation and testing activities; they are summarized below:

a) The system model is compiled. Data type checking is performed and some run-time checks are embedded in the load module. The load module is a combination of the compiler output, the Ada standard executive and the compiled device simulator modules.

b) If any compile time error is detected, the compiler will inform the user through the diagnostic file.

c) The load module is allowed to run. Run-time checks can detect deadlock, verify if subtypes are used consistently and that variable ranges are within bounds.

Depending on how sophisticated the simulation controller is, it might be possible to collect data on the comparative execution speed of each module. This information, while not truly representative of the speed of the finished product, is extremely useful when hardware/software decisions have to be made.

When it comes to evaluate the results of the simulation, it should be realized that the whole system was implemented in Ada. This may not necessarily be the case in real systems; special functions may be coded in machine language for the sake of efficiency. Nevertheless, from the execution time figures, execution time limits or bounds can be derived. Once the functions of a module are known and the maximum time allowed to execute those functions is established, it becomes possible to decide whether to:

a) Decompose the module further since no combination of hardware/software or hardware alone can satisfy the speed requirements of the module.

b) Implement the module as a specialized hardwired unit.

c) Implement the module (or merge the module with another and implement then both) on a given processor with a given software algorithm. The actual allowable time to execute the module functions is influenced by such factors as processor speed, processor power (i.e., how good is the instruction set), the type of algorithm chosen to implement the functions, etc.

It should be stressed again that decomposing the system with the help of Ada specification blocks facilitates the eventual coding of some functions either as Ada modules or as assembly language modules. It is also obvious that the high level system model is not immutable. In fact, practical considerations may dictate that modifications be made to it. Different task partitioning may also prove to be necessary.

3.2.5.4 Computer Aided Tools and the Specification Process

A computer aided simulation and testing tool was proposed in the last section. However, this tool took as input the complete specification of a system. The question is: Is it possible or desirable to develop computer aided tools for the specification process? In the affirmative, what functions should those tools have and how can they be built?

The answer to the first question would appear to indicate the desirability of computer aided specification tools. A further examination reveals that the tools that can be built would be restricted to a small scope. The specification process takes requirements and transform them into an abstract

form. This is still largely the province of the human mind. As such, it is difficult to simulate completely.

A system called SADT [ROSS77a], [ROSS77b] was mentioned earlier, in the context of functional languages. It should be stressed that SADT is also a complete methodology and as such should be discussed here. SADT is an elaborate set of rules and guidelines with the purpose of establishing a common language and method for specification. SADT is applied in the early phase of data flow decomposition and functional analysis. SADT or a derivative could be used advantageously in the preliminary stages of the decomposition process. The resulting diagrams could then be translated into Ada specification blocks as before. Some work has also been done towards integrating SADT with a simulation tool. The result called SAINT [BACH81] includes a dynamic simulation tool.

A tool that would also be very useful is a graphics package that would relieve the analyst from the burden of drawing boxes and connections. The topology of the graph should be remembered by the graphics package so that a comparison could be made

with the output of the compiler after an Ada model has been compiled. Any mismatch would be detected and the analyst could then correct it.

3.3 Validation of Specifications

3.3.1 Validation

Validation is a process whose purpose is to check the validity of a transformation. Figure 3.26 shows two stages of program development linked by a transformation T.

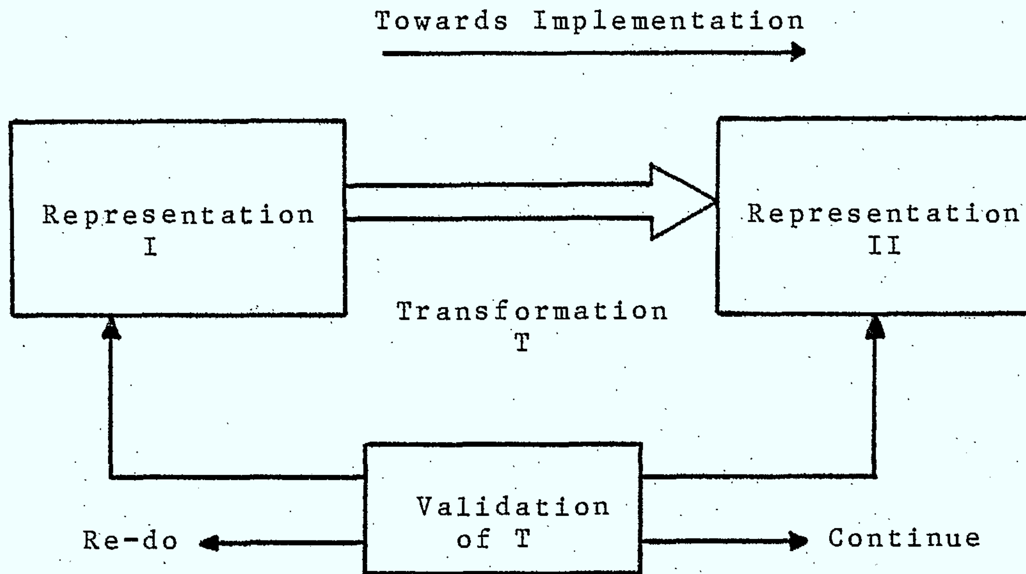


Figure 3.26: Example of Transformation

The validation of T in this case is the checking of Representation II against Representation I with respect to functionality. If the functionality of Representation II does not quite correspond to that of Representation I, then the transformation is not entirely correct.

To put the concept of validation in the perspective of the design methodology of the previous section, three main steps should be considered. Those steps are: Requirements, Specification and Implementation, as shown in Figure 3.27.

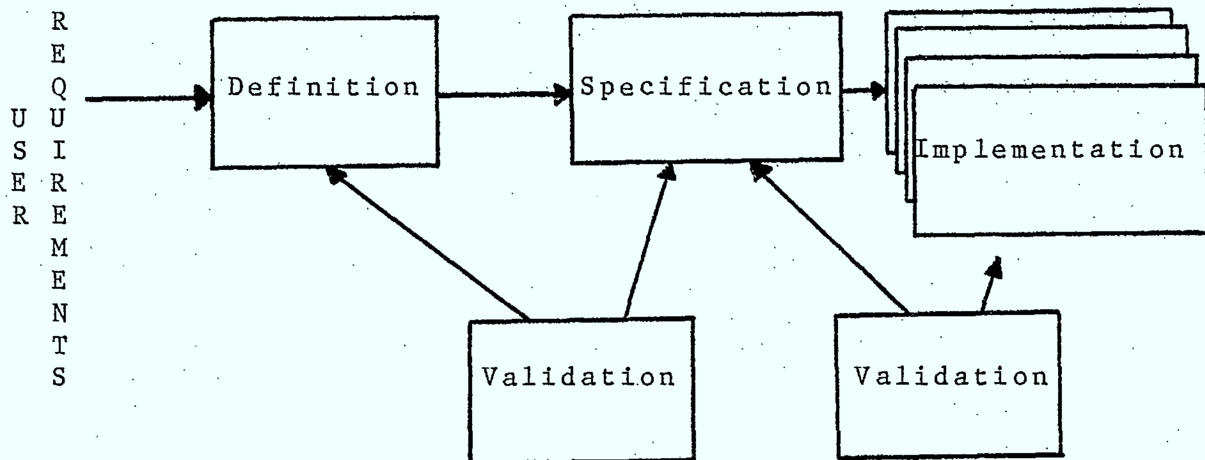


Figure 3.27: Validation and the Design Process

The role of validation is to check the correctness of each transformation, from the early requirements to the final implementation. Several advantages accrue from this exercise:

1. The transformation is verified.
2. Existing flaws are uncovered so that they can be rectified.
3. The behaviour of the output of the transformation is verified.

Figure 3.27 shows validation in two places:

a) Between Definition of Requirements and Specification.

At this stage, validation is based upon a mathematical model of the system. Validity of the specification is carried out through high level testing or proving (that is, symbolic execution).

b) Between Specification and various levels of implementations.

Validation is based on the implementation and the language used to describe it. If the language is completely axiomatized, assertions can be derived in order to make the symbolic

execution of the program possible. A successful symbolic execution of the assertions in the program attests to the correctness of the implementation. This method is called verification.

Another method to obtain validated implementation is testing. Testing involves traversing each of the program branches and at the same time, checking the output thus obtained. Although simple and easy to understand, testing may not always be the best alternative due to the difficulty of choosing a meaningful set of test data.

This section on validation will concentrate on validation (verification and testing) of implementations of systems. Validation of specification is more nebulous at present. The difficulty lies in representing the requirements in an acceptable mathematical form. Research activities so far have, therefore, been concentrated mostly on implementations of systems.

This section will cover three topics of importance in validation:

a) Testing,

b) Verification, and

c) Automated Verification Systems (AVS).

In the course of the discussion, full definitions will be provided as well as indications of how applicable and relevant those concepts are to the current work.

3.3.2 Testing

Testing [HEND77], [GOOD77] is a methodology which can be applied to a program to determine its validity. The degree, that is, the thoroughness of testing is under user control. Testing rests upon basic observations on the behaviour of programs, be they at a high level or at a low level. Figure 3.28 shows the skeleton structure of an imaginary program with all the control paths being given numbers. The purpose of testing is to select a set of test data so that:

a) all the paths of the program have been traversed at least once and

b) the output thus obtained is valid.

Another way of explaining testing is to consider a program F as a transfer function. Figure 3.29 depicts the domain and range of the transfer function F.

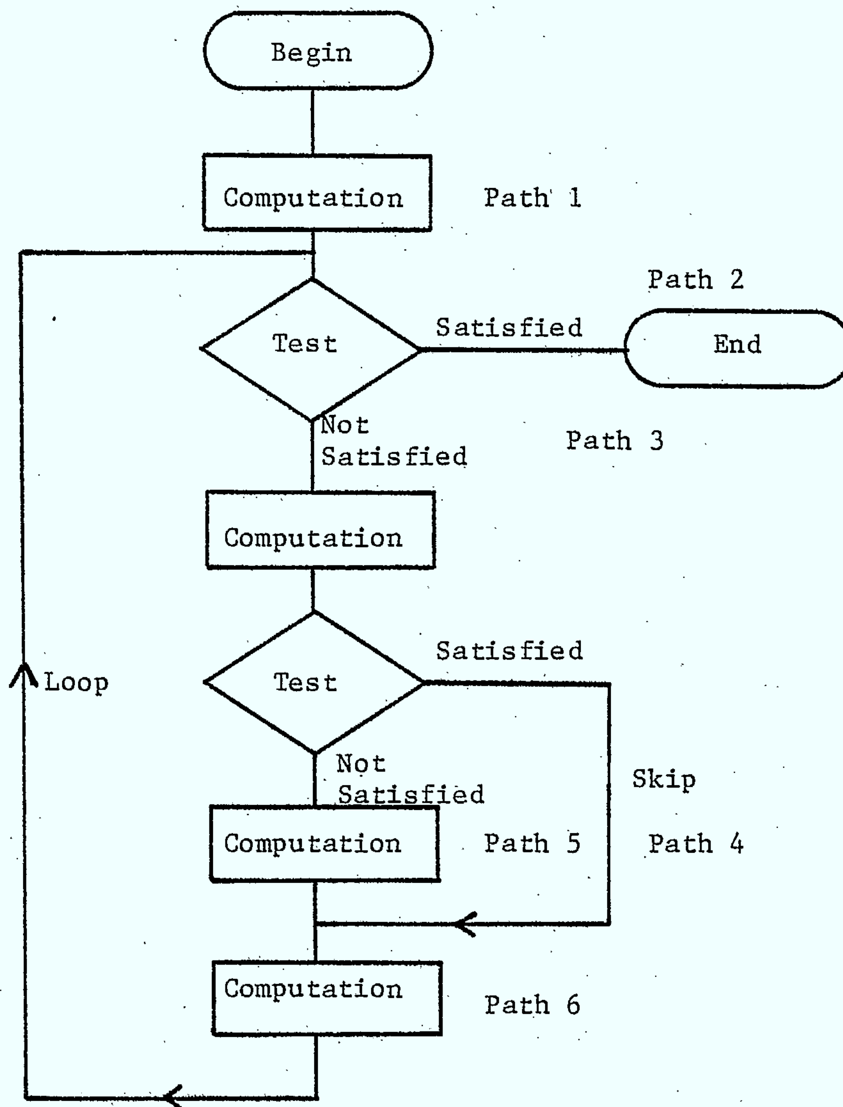


Figure 3.28: Skeleton of a program control structure

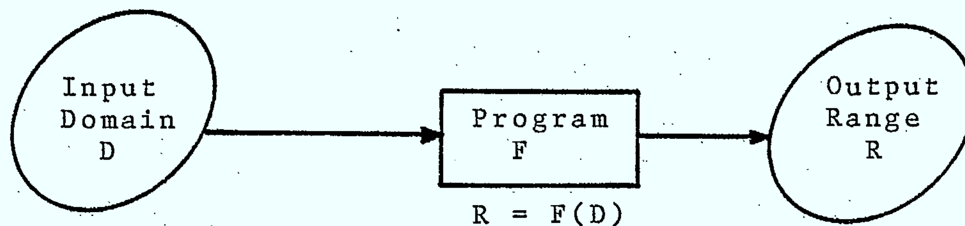


Figure 3.29: A Program's Domain and Range

An element d, $d \in D$, will produce $r = F(d)$ and $r \in R$. If this is the case for all $d \in D$, then the program is said to be valid.

Using that definition of testing, it is clear that the set of test data that is required is the set of d's, $d \in D$ so that:

i) $F(d) \in R$

ii) the set is minimal

iii) all the control paths of the program have been traversed once.

The main advantage of testing and paradoxically its principal weakness lies in its simplicity. Testing is easy to carry out in terms of computer resources and packages. It has the advantage of using not only real data but also data that is meaningful. The program is, therefore, tested in its working environment. However, unless all the paths of a program are known and unless the set of test data is such that complete traversal of those paths is achieved, testing will not provide a guarantee of correctness. To put this differently, testing can uncover the presence of flaws but not prove that there are not any. Given the multiplicity of paths in a program of even moderate size, it is not reasonable to expect that testing will

cover all possibilities. Nevertheless, testing has its usefulness in increasing the level of confidence one has in a program.

3.3.3 Verification

As mentioned before, a system specification and its implementation can be represented mathematically. Given this mathematical model, it is possible to simulate this system by what is known as symbolic execution. What is accomplished is, in fact, the traversal of all control paths in the program or, more precisely, the testing of the program for all possible input data.

Program Verification is a research endeavour which is relatively new. It is nevertheless well documented as surveys and tutorials on verification [HANT76], [LOND77], [GRIE76], [KING80] attest. Another area of research is the automation of the verification process. Section 3.3.4 is devoted to Automated Verification Systems.

The basis of program verification can be defined with respect to the simple diagram of Figure 3.29. An assertion, called pre-assertion, is placed at the input of F. The pre-assertion is true for all d's in D and false otherwise. Similarly, an

assertion, called post-assertion, is placed at the output of F. The output of the program satisfies the post-assertion if the output falls in the range of F. The difficulty most often experienced is in choosing the pre- and post-assertions.

Refinements are, therefore, needed to make this choice easier. The approach usually taken is to break down the program into smaller paths that can be enclosed by a pair of pre- and post-assertions. This is illustrated in Figure 3.30a. The next problem is concerned with the handling of loops. Loops can be considered as a set of simpler serial paths with the parameters changing from path to path. Figure 3.30b shows such a loop and the proposed assertion, called a loop invariant. A loop invariant combines the concepts of pre- and post-assertions. The assertion is true, i.e., satisfied, at the beginning of the loop and similarly at the end, hence the term "loop-invariant".

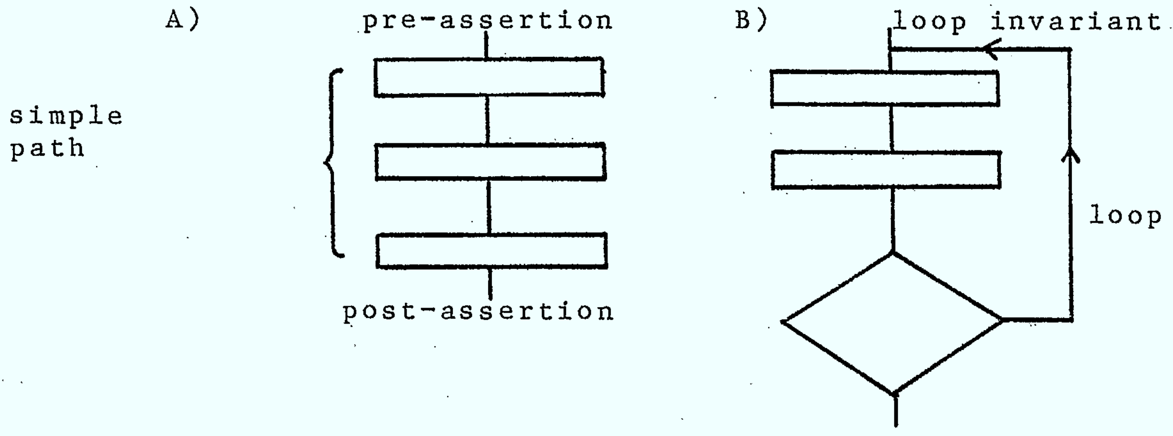


Figure 3.30: Paths and Assertions

The thesis of program verification is that, given a program and a set of assertions mathematically describing the purpose of the program then, if all the assertions are proved to be true, the program is correct. This is, in fact, a symbolic execution of the program since real input data has been replaced by algebraic symbols. This method is also called "Inductive Assertion" method since all the program paths (or segment) are proved independently and an induction argument is then used to establish the correctness of the whole program [FLOY67], [ROBI77], [REYN76]. At this point, a basic requirement of such a verification process manifests itself. The language used to write the program has to have a regular behaviour, easily expressed in mathematical terms. In other words, the language has to be completely axiomatizable (see [HOAR69], [HOARE73]. The reason for this require-

ment is easy to visualize. A statement S can be preceded by a pre-assertion P and a post-assertion Q. Given that P is true, some means of going from P to Q has to exist. If the language of which S is a statement has been completely axiomatized, then S is the tool that will transform P into Q. Therefore, a complete definition of S renders possible the proving of Q.

The preceding paragraphs described the most widely used verification method: inductive assertion. There exists other variations such as "predicate transformers" and "sub-goal induction". In those methods, the basic principles of verification are not altered. Assertions are still used to describe the behaviour of the program and an axiomatized language is still necessary; interesting peculiarities exist, however.

1. Predicate Transformers

The concept of predicate transformers [DIJK75], [YEH77] has been developed to be used in conjunction with "predicate calculus". Predicate calculus is a program methodology destined to help programmers construct their programs with a strong mathematical

base. Going one step beyond leads to verification which is made easier by the already existing mathematical description.

The theory behind Predicate Transformation is based on the state space of a program. On this state space, predicates P, Q, ..., can be formulated. (A predicate is taken to be an assertion.) Associated with a given predicate P, there exists a set of program states, P*, for which P is true.

$$P^* = \{\text{states} \mid P\}$$

Given a program S, it is possible to have a pre-assertion and a post-assertion. These will be respectively P and Q and the following relation is held to be true:

P[S]Q.

In other words, P is true for the input data of S and following the execution of S, Q is also true for the output data produced by S. A function WP will now be introduced and its effects are as follows:

WP(S,Q) = weakest pre-condition such that after the execution of S, Q is true.

WP(S,Q)* = largest set of initial states of S for which S terminates and Q is true.

The function WP is called a predicate transformer since it takes a post-assertion and transforms it into a pre-assertion. In [YEH77], some predicate transformer theorems are stated and some examples are given, illustrating the predicate calculus and transformer methods.

2. Sub-goal Induction

Sub-goal induction [MORR77] is a proof method that can be used to complement the general inductive assertion method. In the latter, loops are handled through loop invariants, while in the former the correctness of loops is proved directly from their input-output specification.

Figure 3.31 shows a simple loop which will be analyzed using sub-goal induction.

```
begin
  :
  while  not P(x) do
         x:= N(x);
  :
end;
```

Figure 3.31: Simple loop example

A post-assertion Q will be used to represent the desired state of the program after the execution of the loop. In this example, the predicate $Q(x,z)$ relates a given input x to the desired output z (*).

Two cases have to be considered in connection with the loop:

Case 1: When the loop is executed for the first time, $P(x)$ is true; the loop is ended with the value of x unchanged. Expressed mathematically, this case amounts to:

$$P(x) \rightarrow Q(x,x) \qquad \text{(c1)}$$

Case 2: When the loop is executed for the first time, $P(x)$ is false. This implies that $x := N(x)$ and thus becomes x'. This second case reduces to:

$$\underline{\text{not}}P(x)\ \underline{\text{and}}\ x'=N(x)\ \underline{\text{and}}\ Q(x',z) \rightarrow Q(x,z) \qquad \text{(c2)}$$

(*) This example is that of [MORR77].

The two cases, namely, c1 and c2, are called verification conditions. In fact, proving c1 and c2 amounts to proving the correctness of the loop. Sub-goal induction is, therefore, a method to obtain those verification conditions based upon the specification of the loop.

To resume the description of the process of verification, the example of the "Quotient-Remainder" will be presented. This example has first been given by [HOAR69], and then by [WIRT73] and [LOND77]. The example is based on the simple division program using the successive subtraction method (shown in Figure 3.32). In a first step, assertions are introduced in the program, as shown in Figure 3.33. It should be realized that the assertions are not executable statements and could be specified separately in a specification language.

```
function Divide (var x: integer ; var y:integer):integer
var r,q: integer;
begin
   r:=x;
   q:=0;
   while y >=r do
        begin
           r:= r-y;
           q:= q+1
        end;
   Divide:=q
end  {of function Divide}
```

Figure 3.32: Sub-program for simple division

```
function Divide(var x: integer;var y:integer): integer
var r,q:integer;
begin
   pre    true; {no restriction upon entry}
   r:=x; q:=0;
   while y >=r do
     begin
       assert x= r+(y*q);
       r:=r-y;
       q:=q+1
     end;
   Divide:=q;
   post x=r+(y*q) and r < y
end {of function Divide}
```

Figure 3.33: Sub-program with assertions

From the assertions, three lemmas can be obtained which correspond to the three assertions. In deriving the lemmas, it is assumed that the programming language has been axiomiatized.

1- Lemma I

true <u>and</u> r=x <u>and</u> q=0 → x=r+(y*q)

2- Lemma II

x=r+(y*q) <u>and</u> y<=r <u>and</u> r'=r-y <u>and</u> q'=q+1 →
x=r' +(y*q')

3- Lemma III

x=r+(y*q) <u>and</u> <u>not</u> y<=r → x=r+y*q <u>and</u> r < y

This axiomiatization is necessary in order to be able to transform a pre-assertion into a post-assertion. For example, in the lemmas, it can be seen that := has been replaced by =, since the behaviour of the assignment construct had been axiomatized.

The next step in the verification process is to prove the three lemmas. Fortunately, it is relatively easy to do in this case (*).

(*) Obviously, it will not always be that easy! See [POLA79].

1. Lemma I requires substituting r for x and 0 for q and $x=r+(y*q)$ is thus verified.

2. Lemma II requires substituting r-y for r' and q+1 for q'. The equation $x=r'+(y*q')$ is verified since $x= (r-y) + (y*(q+1))$ which reduces to $x= r+(y*q)$.

3. Lemma III is the exit assertion. The term $r < y$ is equivalent to <u>not</u> $y <= r$ and the term $x= r+y*q$ is true.

The proofs of those three lemmas have now been completed and the correctness of the sub-program Divide has been established. The lemmas themselves are interesting because they could have been stated differently. Hereto, the lemmas have been associated with the forward execution of the program. Backward execution can also be considered. The lemmas that it yields are slightly different.

1- Lemma I

true $\rightarrow$ $x = x+y*0$

2- Lemma II

$x = r+y*q$ <u>and</u> $y <= r \rightarrow x=(r-y) + y*(q+1)$

3- Lemma III

$x = r+y*q$ <u>and</u> <u>not</u> $y <= r \rightarrow x=r+y*q$ <u>and</u> $r < y$

The forward and backward methods of generating assertions are discussed comparatively in [KING 76].

Once a proof of correctness has been obtained for a given program, the question that comes to mind is: how correct and how reliable is the program? Obviously, the program itself is correct provided the proof was done correctly and the language was axiomatized properly. For the program to run correctly, extra factors have to be considered. Compiler correctness will influence how reliable the "correct" program will be. In this case, bugs in the compiler would jeopardize the correct execution of the program. Hardware correctness has to be considered as well. In fact, in order to establish the correctness of the hardware, the behaviour of the processor and the other components has to have been axiomatized. Based upon the material presented in this section, one can see that hardware axiomatization is necessary if one aims at proving the correctness of a system.

It would appear that even if a program has been proven correct, its correct execution is not automatically guaranteed. However, the risks of software related failure have been greatly minimized.

To some extent, this is what testing achieved, but not to the same degree. In practical situations, testing tries to traverse most of the paths likely to be used during the execution of a program. Verification, of course, traverses all of them. The cost of testing being considerably less than verification, there exist situations where, for both practical and economic reasons one alternative would be preferable.

A very important point that has to be emphasized is that Verification (just like testing) is an activity that should be planned and carried out in parallel with program development. The applicability of verification to the design methodology is still an unresolved issue. On one hand, it would be advantageous to have a completely validated design from the top to the bottom level. On the other hand, Verification is very costly of time and efforts and requires skilful users. In satellite systems, where reliability is paramount, verification should be a goal worth considering.

3.3.4 Automated Verification Systems (AVS)

In an effort to facilitate the task of verification, several research activities have investigated the possibility of involving the computer in the verification process. Automated Verification Systems (AVS) were the results of those efforts. The theory behind the operation of a typical AVS is no different from that of a hand proof. Of course, some steps have to be spelled out due to the different natures of the human mind and of the computer.

The process of designing and of verifying a program using a hypothetical AVS is shown in Figure 3.34. The first step towards verification, once the program is written, is to insert the assertions and loop-invariants. Those may be difficult to obtain but it will be assumed that they have been properly generated.

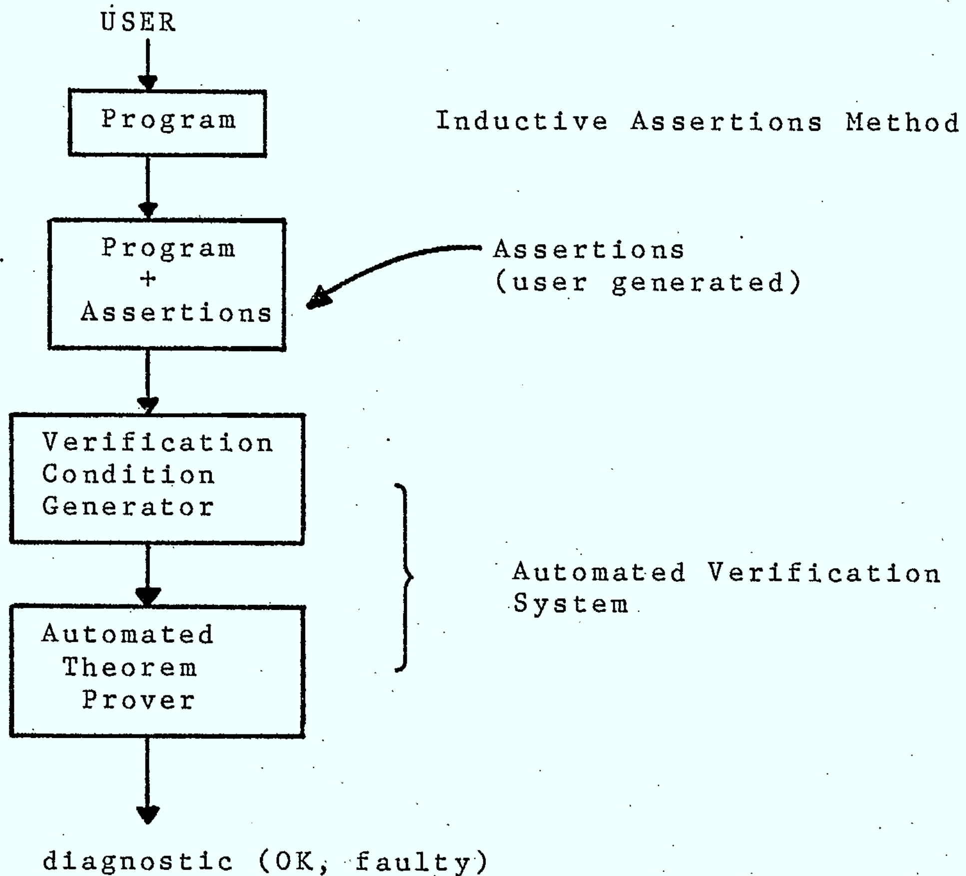


Figure 3.34: Design and Verification of Programs

The AVS accepts as input the program and its assertions. It uses the assertions to produce verification conditions which will be of a format suitable for machine proving. The verification conditions are, in fact, considered as theorems and various techniques can be used by the theorem prover (see [BOYE79]). The "inductive assertions" method can be used, sometimes supplemented by subgoal induction. It has to be pointed out that the AVS

will, in general, prove partial correctness. Total correctness is proved when the program has been ascertained to terminate. In order to illustrate those concepts, an example will be presented which will take a subprogram and perform the steps outlined in Figure 3.34.

The example to be considered is basically that of [ROBI77] and uses register modules as defined by [PARN72]. A register module is a variable size vector whose components are ordered by size in an increasing sequence. The following predicates always hold for a register module.

$0 <= \text{Length} <= 1000$
$\forall i,\ 0 < i <= \text{Length}$, RegisterModule[i] is defined,
$\forall i \mid$ RegisterModule[i] is defined,
$0 <= \text{RegisterModule}[i] <= 255$.

The subprogram to be verified with respect to the register module is shown in Figure 3.35.

```
Procedure InsertSorted(var c: integer);
 var x:integer;
 begin
              <- - - - - - - - - - - - Assertion I
    x:=1;
    repeat
              <- - - - - - - - - - - - Assertion II
         if x >= Length+1
         then Insert (x-1,c)
         else if c =< RegisterModule[x]
              then Insert(x-1,c)
              else x:=x+1

         until c is inserted
              <- - - - - - - - -      Assertion III
end  of InsertSorted
```

Figure 3.35: Procedure InsertSorted

The subprogram of Figure 3.35 refers to another procedure called Insert (i,j). "Insert(i,j)" inserts the value j after position i and moves subsequent values one position higher. Keeping this definition in mind, it is now possible to state the assertions of "InsertSorted". Three assertions are necessary and should be placed as indicated by arrows in Figure 3.35.

1. Assertion I

$\forall k \mid 1 =< k =< Length-1,\ RegisterModule[k] =< RegisterModule[k+1]$
<u>and</u> $0 =< c =< 255$
<u>and</u> $0 =< Length < 1000$

2. Assertion II

$Length = Length_o$ <u>and</u> $c = c_o$ <u>and</u>
$\forall k \mid 1 =< k =< Length,\ RegisterModule[k] = RegisterModule_o[k]$ <u>and</u>
$\forall k \mid 1 =< k =< x-1,\ RegisterModule[k] < c$ <u>and</u>
$1 =< x =< Length+1$

3. Assertion III

$Length = Length_o + 1$ <u>and</u>
$\forall k \mid 1 =< k =< Length-1, RegisterModule[k] =< RegisterModule[k+1]$
<u>and</u> $BagOf(i,1,Length,RegisterModule(i)) =$
$BagOf(i,1,Length_o,RegisterModule_o(i)) \cup Bag\ (c_o)$

Assertion I describes the state of the RegisterModule at the beginning of the procedure. The RegisterModule is sorted and has room for another character. The character to be inserted is within bounds. Similarly, Assertion III describes the state of the RegisterModule after the insertion took place. It shows that the length of the RegisterModule has been incremented and that the RegisterModule itself is still sorted.

Assertion II describes the state of the RegisterModule and of the procedure during the loop. In Assertion II, Bags and BagConstructors [KNUT68] are used. For example, Bag (a,b,c,) is the set of three elements a,b,c with Bag(a,b,c) = Bag(b,a,c). The BagConstructor is BagOf(i,a,b,expression(i)) which represents the bag of elements obtained by substituting b for a, for i in expression(i).

The next step taken by an AVS is the generation of verification conditions. Those are to be found in the following list: (The list and proof of VC's are basically that of [ROBI77]).

1. $\forall k | 1 <= k =< Length_o - 1,\ RegisterModule_o[k] =< RegisterModule_o[k+1]$
2. <u>and</u> $0 =< c_o =< 255$
3. <u>and</u> $0 =< Length_o < 1000$ <u>and</u>
4. $\forall k | 1 <= k =< x-1,\ RegisterModule_o[k] < c_o$
5. <u>and</u> $1 <= x =< Length_o + 1$
6. <u>and</u> $x < Length_o + 1$
7. <u>and</u> $c_o =< RegisterModule_o[x]$
8. <u>and</u> $Length = Length_o + 1$ <u>and</u>
9. $\forall k,\ RegisterModule[k] =$ <u>if</u> $k =< x-1$ <u>then</u> $RegisterModule_o[k]$;
   <u>if</u> $k = x$ <u>then</u> c_o;
   <u>otherwise</u> $RegisterModule_o[k-1]$

After insertion of c_o, verification conditions 10 to 16 are derived from the verification conditions 1 to 9.

10. $1 <= x =< Length_o$ <u>and</u>
11. $0 =< x-1 =< Length_o$ <u>and</u>
12. $0 =< c_o =< 255$ <u>and</u>
13. $Length_o < 1000$ <u>and</u>
14. $Length = Length_o + 1$ <u>and</u>
15. $\forall k | 1 =< k =< Length-1,\ RegisterModule[k] =< RegisterModule[k+1]$
16. <u>and</u> $BagOf\ (i, 1, Length, RegisterModule(i)) = BagOf\ (i, 1, Length_o, RegisterModule_o(i)) \cup Bag\ (c_o)$

The automatic verification system would then start proving each of those verification conditions as theorems. Most of the above verification conditions are straightforward to prove. Condition 15 will be used to illustrate the theorem proving activities of the AVS. Verification condition 15 is simplified by substituting expressions 8 and 9. The new condition is shown below, in Figure 3.36.

$\forall k \mid 1 =< k =< Length_o,$

$\{$ **if** $k =< x-1$ **then** $RegisterModule_o[k]$ (a)
else **if** $k = x$ **then** c_o (b)
else $RegisterModule_o[k-1] \}$ (c)

$=<$

$\{$ **if** $k+1 =< x-1$ **then** $RegisterModule_o[k+1]$ (d)
else **if** $k+1 = x$ **then** c_o (e)
else $RegisterModule_o[k] \}$ (f)

Figure 3.36: Verification Condition 15

The binary relation =< relates the two expressions in bracket. Nine possible cases result but with only four of them being non-trivial. Using the letters at the far right of Figure 3.36 to represent each particular case, the four cases of interest become:

1. Case 1: a,d
2. Case 2: a,c
3. Case 3: b,f
4. Case 4: c,f

Those cases are proved in the following fashion: (item i corresponds to case i).

1. $\forall k \mid 1 =< k =< x-2,\ RegisterModule_o[k] =< RegisterModule[k+1]$
2. $RegisterModule_o[x-1] =< c$
3. $c =< RegisterModule_o[x]$
4. $\forall k \mid x+1 =< k =< Length_o,$ $RegisterModule[k-1] =< RegisterModule[k]$

At this point, the AVS has completed the proof of verification condition 15 and would continue on to 16. Automated verification systems perform along similar lines as a proof by hand. Their requirements are very much the same in that they need an axiomatized language and the insertion of assertions. It is clear that assertions are very important to the AVS since they mathematically depict the behaviour of the system. The onus to produce suitable assertions is on the designer and this seems to indicate that obtaining the assertions is a rigorous activity which should be undertaken concurrently with program development.

Several AVS are being experimented with at the present time, as the following list can attest:

1. Gypsy is a verification system being developed at the University of Texas, [GOOD78], [AMBL77]. Gypsy is also surveyed in [CHEH81]. The Gypsy verification environment is shown in Figure 3.37.

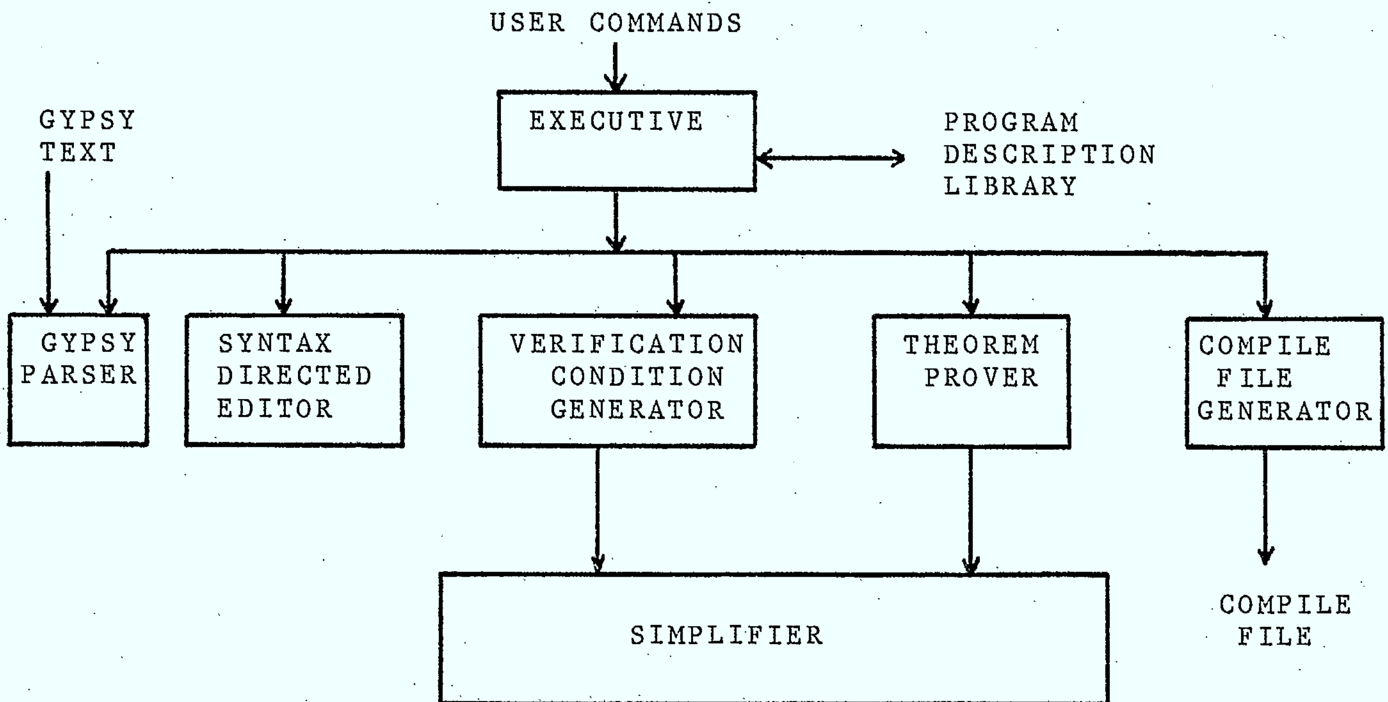


Figure 3.37: Gypsy Verification Environment (from [CHEH81])

Gypsy features a single language (i.e., Gypsy) to program the application and to specify its behaviour mathematically. The Gypsy language is a derivative of Pascal [JENS74] and supports concurrency. Detailed examples of its use are given in [AMBL77] and also [CHEH81]. Gypsy also features a designer/verifier's assistant package [MORI79] to facilitate the task of maintaining previously verified programs.

2. Hierarchical Development Methodology (HDM) [ROBI78], [ROBI79] is a complete methodology for program development from the early stages of specifications to the final stages of implementation. Mathematical representation of the system's behaviour is accomplished through Special, a non-procedural, specification and assertion language. Figure 3.38 shows some of the details of HDM.

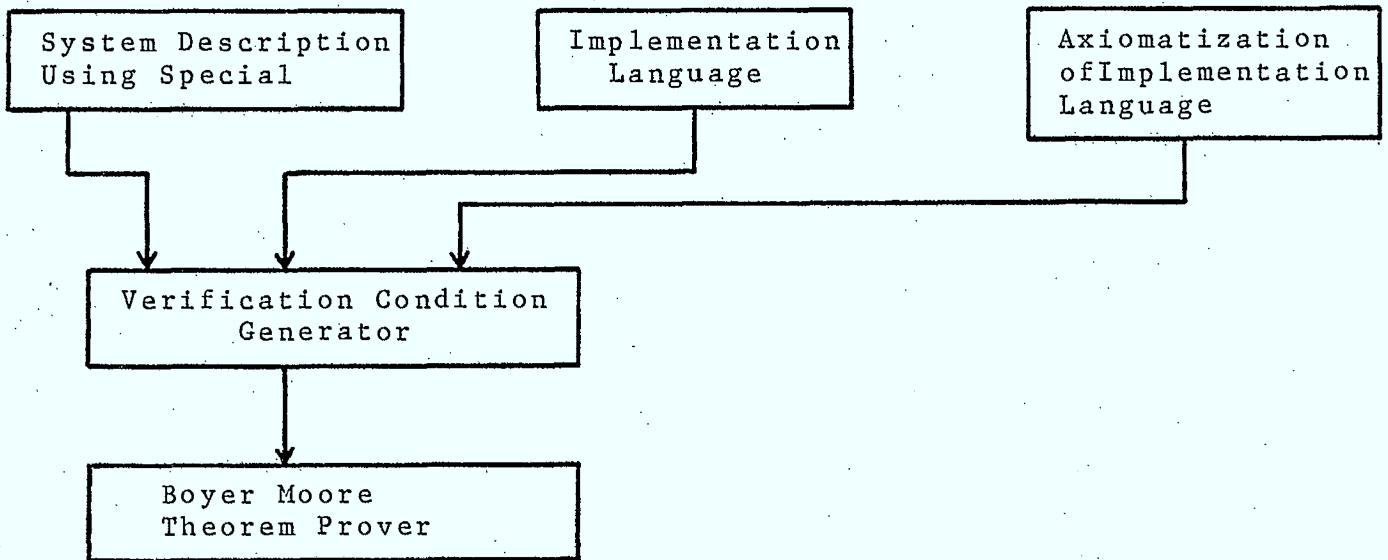


Figure 3.38: Description of the HDM System

Special is not a procedural language and, therefore, an implementation language is necessary. Complete axiomatization is obviously a requisite condition on the implementation language if the verification condition generator is to perform properly. The verification conditions thus generated are given to the theorem prover for automated proof. The theorem prover is the Boyer Moore [BOYE79] theorem prover and is among the most powerful available.

3. Affirm [AFFI79] is primarily an interactive system requiring considerable directions from the user. It uses a variant of Pascal for specification and implementation. Other facilities are also provided for data type specification and for theorem proving.

4. Other systems such as the Stanford Pascal Verifier [LUCK79], the Formal Development Methodology (FDM) [KEMM80] should also be mentioned. (Other systems also exist in the early experimental stages and are not mentioned here.)

As pointed out previously, all those automated verification systems are at various stages of experimentation. Their use is costly but, above all, requires trained programmers well versed in mathematical programming. The next section will cover the validation capabilities that can be reasonably and realistically incorporated to the design methodology.

3.3.5 <u>Proposed Validation Capabilities</u>

The specification methodology should incorporate some validation capabilities. The nature and the extent of those capabilities should be carefully chosen. An ideal system, even though not practical at present, is shown in Figure 3.39.

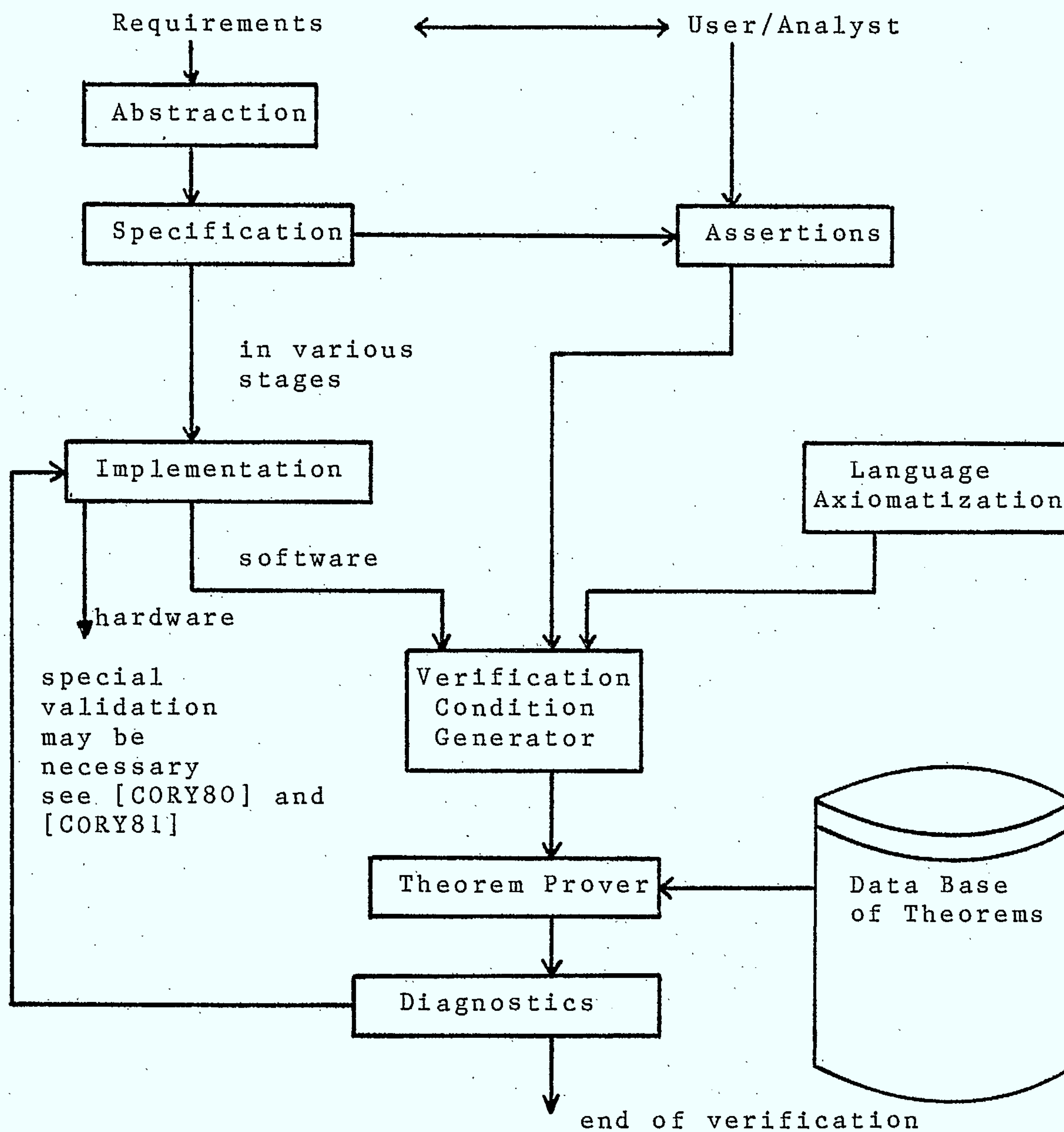


Figure 3.39: An Ideal System

Several factors make this system impractical:

1. Ada is going to be used as the specification/implementation language. Ada has not been fully axiomatized and it is not likely that it will, at least in its present form [YOUN80].
2. Concerns about code correctness may be dwarfed by concerns about compiler and especially run-time environment mechanisms correctness.
3. As pointed out before, use of those AVS tools is not necessarily complicated. However, obtaining mathematical descriptions of systems through assertions is not a trivial task. As systems become more complex and as the issues to be considered proliferate, the problem of verification will become more and more unwieldy.

As more research is pursued in the field of verification in general and in Automated Verification Systems in particular, it is reasonable to expect that verification and AVS will become more powerful and easier to use. In their present form, their overall complexity precludes their use on a

large scale, such as a validation of a complete system. Validation/verification work of a smaller scope can be undertaken; an example of such works would be the verification of small modules implementing functions which are critical to the proper functioning of a larger system. This latter alternative would be feasible for the multi-micro processor design methodology. Its use, however, should be within in a consistent and logical framework. Such a framework, shown in Figure 3.40, combines the various features outlined so far in the last two sections.

The interim verification system of Figure 3.40 is, of course, a compromise and should be augmented with new capabilities for decomposition and verification, if and when these become available. In fact, any addition to the interim verification system should tend to transform it into the ideal system of Figure 3.39. At present, the interim verification system achieves limited verification partly through checks performed at pre-compilation and pre-simulation time and partly through formal verification of some critical sections of limited scope.

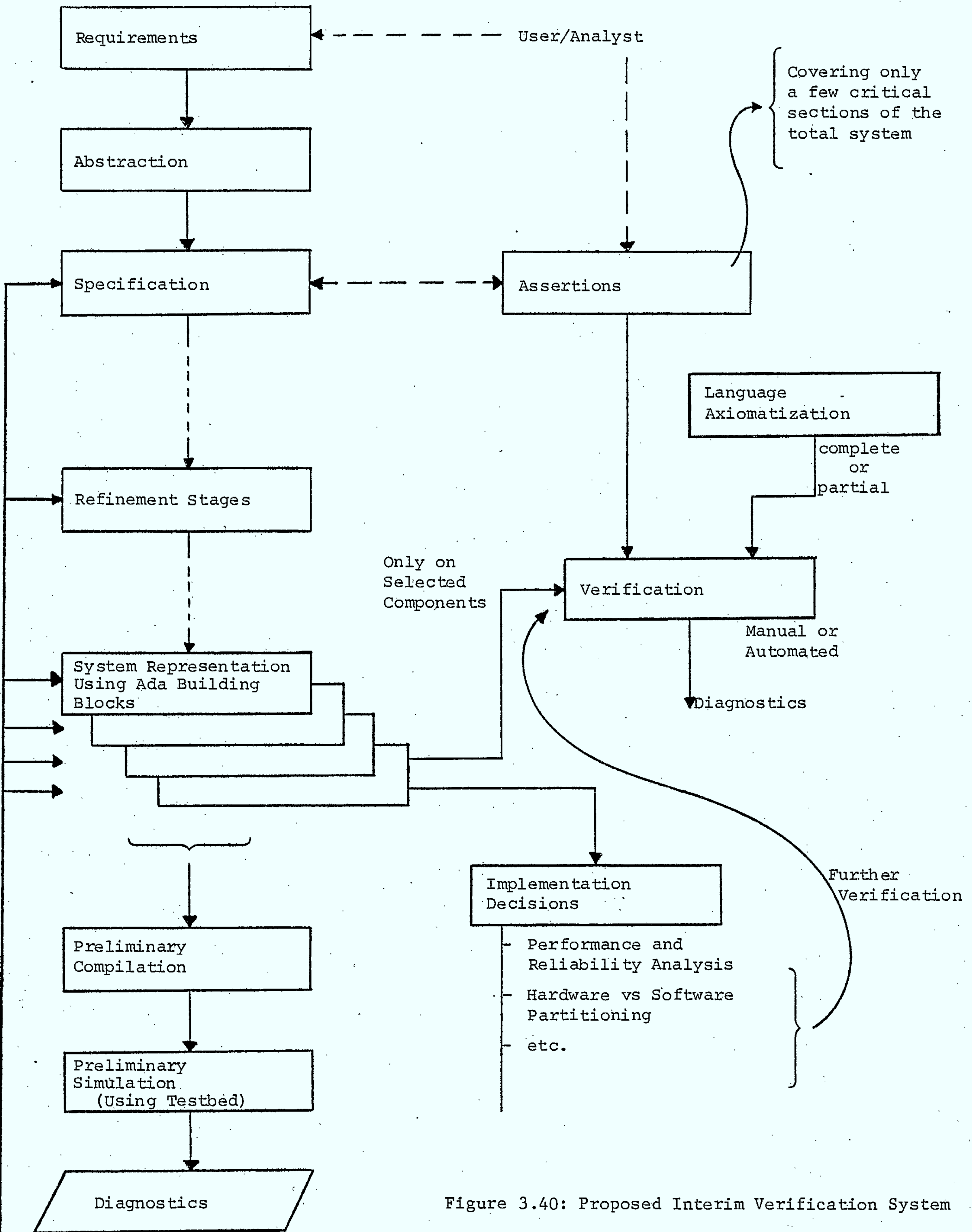


Figure 3.40: Proposed Interim Verification System

A useful addition to the preliminary compilation would be the inclusion of imports/exports constructs fashioned after those of Euclid [LAMP77]. Those constructs are not part of standard Ada and would be direted expressly at the specification blocks. Access into a block and access originating from inside a block would be tied to a source or a destination. The advantages accrueing are twofold:

1. A tight control on interconnection would be achieved.
2. Further validation work would be made easier.

3.4 Summary and Conclusions

This section on specification and validation was an attempt to describe a design process based on requirements and producing specifications leading to easy implementation. Validation of the specifications and possibly of the implementation was also covered, mostly by means of survey of existing techniques.

The results of this research can be summarized as follows:

1. The transformation of requirements into formal or even informal specifications is still largely the task of a systems analyst. The necessary abstraction power makes the automation of this activity within the near future unlikely.
2. The process of obtaining specification was described and guidelines were provided. Decomposition techniques, such as dataflow analysis and functional decomposition were introduced. Ada specification blocks were used to provide a strict formalism .

3. The specifications obtained with Ada specification blocks can be transformed naturally into an implementation. This transformation is done in various steps, with each step being a refinement of the previous one.
4. Validation techniques were investigated with respect to their applicability to the design methodology. Although no concrete validation system was specified, preliminary validation capabilities were outlined. A more powerful validation system was also described as a desirable goal for future research.

Due to its research nature, the work reported herein is not definitive. Modifications will, no doubt, be made as new problems are discovered (in the course, perhaps, of actual experimentation).

The methodology should be experimented with in the context of a large example. This would allow for improving the methodology itself and would also

provide the necessary framework for the development of computer aided tools such as:

1. A graphical aid to decomposition (outlined previously).
2. A simulation testbed for the preliminary simulation phase.
3. A verifier which could be used on selected sections of the system.

In connection with verification, it should be pointed out that, due to the very complex nature of the task, it would be desirable to gain experience with an already existing system, such as Gypsy. Theory would then be substantiated by practical experience.

4.0 PERFORMANCE AND RELIABILITY

This section examines the potential applications of CAE tools for the analysis and optimization of the performance and reliability characteristics of multi microprocessor systems design. First the primary questions which CAE tools can answer are considered. Then the application of these tools at each design stage are explored. Next the reliability and resource usage models which form the building blocks of CAE tools are examined. Based on these fundamental models, areas for the development of new or improved CAE tools are identified. The conclusions are summarized at the end of the section.

4.1 Introduction

The problems in which designers employ CAE performance/reliability tools involve the determination of one of the following three factors given that the other two are known:

(a) the architecture of the entire sytem,

(b) the performance/reliability levels of each of the components of the system, and

(c) the performance/reliability of the entire system.

The most basic CAE tools assist the designer in solving the following problem:

Given the architecture of the system and the performance/reliability levels of each of the components, what is the performance/reliability of the entire system?

By solving this problem for different design alternatives, the designer can solve the following two more difficult problems:

1. "Given the architecture and the performance/reliability requirements of the system, what is the required performance/reliability levels of each of the components of the system?",
2. "Given the performance/reliability levels of each of the components and the required performance/reliability of the system or module, what should the architecture be?"

The more powerful and sophisticated CAE tools provide additional assistance to the designer in answering these last two questions. Only in the most fully automated design environments do the CAE tools answer these questions with a minimal amount of involvement and interaction on the part of the designer.

4.2 Scope of CAE Tools in the Performance Area

This section examines the demand for performance/reliability CAE tools in the successive design stages. The potential application and scope of CAE tools are derived from the designer's needs in each of these stages. Opportunities for "pre-building" the CAE tools before a micro-computer design project starts are also explored.

4.2.1 Architecture Selection

The first stage of selecting a hardware/software architecture is primarily a "strategic" type of decision. Typical decisions include the level at which redundancy is implemented (component, assembly, module or system), the communication protocols, and the resource scheduling policies. These decisions are based mostly on experience. Important variables and decisions are often expressed subjectively. Because of the complexity associated with the design of highly reliable multiprocessor systems, the architecture decisions must be in the form of a coherent strategy rather than separate fragments of detailed solutions.

CAE tools which are used in the Architecture Selection stage are usually employed on a one shot basis to answer a specific question. Thus the automated tools which are the most useful are General Purpose Tools. Examples of these tools include general purpose simulation languages such as GPSS [GORD75] and SIMULA [FRAN77].

An example question related to the performance assessment in the Architecture Selection stage is "How does the throughput of a bus with a prioritized demand access protocol compare with a bus which has a time slot access protocol when the number of processing units and the frequency of the processing unit's access to the bus are varied?" The answer is usually found by simulating the two alternatives under varying conditions. To do this quickly and economically, the

designer can use an existing general purpose simulation tool to create a simulation model of the two alternatives. This simulation model is a Specific Tool for the question at hand. The designer first validates that the simulation model is correct by comparing its results with analytical predictions. Once satisfied with the correctness of the model, the designer then uses it to examine the performance of the alternative bus protocols under different conditions.

4.2.2 System Model (Hardware and Software Selection)

The primary objective of the System Model Design stage is to optimize the design within the confines of the Design Policies set by the Architecture Selection stage. Typical decisions made in the this design stage are the number of processing units and the allocation of software tasks to processing units and memory partitions. This optimization normally involves the handling of a considerable amount of data and the repetition of complex calculations. Therefore, this stage is an excellent candidate for computer automation.

The CAE tools which are used for performance evaluation in the System Model Design stage are usually custom tailored for the hardware/software architecture being employed. To understand the reason for this, it is useful to review the underlying methodology for performance/reliability evaluation. The basics of this methodology are shown in Figure 4.1. The process starts with a <u>Design Description</u> consisting of information such as the hardware organization

of processing units, memories, buses, and I/O ports, software organization of tasks, buffers and inter-task communications. This is followed by mapping the relevant information onto the Performance/Reliability Models. This mapping process is called Abstraction. From these models, results are obtained, interpreted and conclusions drawn. Based on the Conclusions, design changes can be made and the cycle repeated until an "acceptable" design is produced. An acceptable design is one which meets the performance requirements.

The application of this methodology to the performance evaluation and design of multiprocessor hardware is illustrated by the example of Figure 4.2(a). The Design Description shows a dual processor architecture with both local and common memory. The various cycle, access, and delay times are included in the description. By the process of Abstraction, the performance characteristics of the components are extracted from the design and configured into a Performance Model. The model, shown in Figure 4.2(b), considers the processing units, memories and I/O interface as servers in a queuing network. The conflicts and resulting delays in the access to the Common Bus are modelled by the Common Bus Queue. The delay of the Bus Interfaces and the access times of the Common Bus Devices (ROM #3, RAM #3 & I/O) are lumped into one server, the Common Memory, which has a total service time of 0.5 uSec.

From the Performance Model of Figure 4.2(b), the following Results are obtained by mathematical calculations and by running the simulation model:

1. Processing Unit #1 executes 2.0 million cycles/sec.
2. Processing Unit #2 executes 1.3 million cycles/sec.
3. Average Common Bus Queue Length = 0.6

By Interpreting these Results, two Conclusions are drawn:

1. More processing throughput is needed.
2. Too much time is wasted in accessing Common Memory.

Next, design changes are recommended:

1. Decrease the Access Time of ROM #3, RAM #3, and I/O to 0.2 uSec.

This completes one cycle of the methodology which is repeated until a hardware design, which has acceptable performance, is obtained.

The critical processes in this methodology are the Abstraction and Interpretation steps. In general, these steps can only be done by the human mind and cannot be done by computers until significant breakthroughs are made in artificial intelligence.

If the Abstraction, Interpretation and Design Change steps must be performed manually, then the CAE tool can assist directly in solving the following basic performance analysis problem:

> "Given the architecture of the system or module, and the performance/reliability levels of each of its components, what is the performance/ reliability of the system or module?"

In summary, the following conclusions can be made regarding CAE tools for this design phase:

1. CAE tools which incorporate performance and reliability models can be used to optimize many designs which employ various computer architectures. Their main advantage is to relieve the designer of the tedious and time-consuming tasks of processing (number crunching) and storing large quantities of data. Because these tools are compatible with many different architectures, they can be built before an architecture is selected.
2. CAE tools which perform the Abstraction, Interpretation and Design Change steps as well as the performance and reliability calculations can be built only after the computer architecture has been selected. These tools further reduce the amount of manual design

effort and the total time required for the initial design stage. This is particularly true if the same architecture is used in several projects since powerful CAE tools would then be available at the beginning of each project. Re-using CAE tools in many projects also helps justify their often substantial development costs.

4.3 Reliability Models

The reliability analysis models enable the designer to examine the probability of failure, hence the survival probability of a given architectural configuration. Two types of failures are considered:

(i) failure due to exhaustion of spares and

(ii) failure due to imperfect coverage.

Also a model for calculating component reliabilities is presented.

4.3.1 Component Reliability Model

In general, the failure of the electronic components follow a Poisson distribution with failure rate L. Thus the reliability of the component is equal to no failure in time [0, t], given by:

$$P_r \text{ (no failure in time } [0, t]) = e^{-Lt}.$$

4.3.2 Exhaustion of Spares Model

This model calculates the probability that a sufficient number of spares fail gradually over time so as to render the remaining parts incapable of performing the required system function. In this model, the hardware architecture of the micro-computer system is considered in terms of its Basic Modules. A Basic Module is a module whose failure is independent of the failure of other modules in the system. This approach is often applied in fault tolerant architectures where identical copies of hardware modules are employed as spares or in voting strategies. Several assumptions are made in the construction of this model:

(i) The failure probability of a particular Basic Module is independent of the failure state of other Basic Modules.

(ii) A failing Basic Module is not repaired and will be isolated from the remaining components.

(iii) The system starts from an Initial State in which all components, including the redundant ones, are functioning, i.e., the system has a perfect Initial State.

For an architecture with n Basic Module types, there is defined a state vector, S, consisting of the tuple $S = (s_1, s_2, \ldots s_n)$; where s_i corresponds to the Basic Module type i. The entry s_i in vector S is an integer whose domain is

zero to N_i; with N_i denoting the number of identical copies of Basic Module type i in the perfect Initial State of the system.

The state of the system, at any time, t, is defined by number of working (non-failed) copies of each Basic Module type. The Initial State of the system can be defined by S_o as:

$$S_o = (N_1, N_2, \ldots N_n)$$

The state of the system after some time $t > 0$ can be represented by:

$$S_t = (N_1^t, N_2^t, \ldots, N_n^t)$$

where $N_i^t \leq N_i$ for $i = 1, 2, \ldots, n$.

A <u>Minimal State</u> is one in which the system is operating with a minimum number of copies of each Basic Module in working condition so that the failure of a copy of any of the n modules will lead to a total system failure. The Minimal State is represented by S_m:

$$S_m = (N_1^m, N_2^m, \ldots, N_n^m)$$

where $N_i^m \leq N_i$ for $i = 1, 2, \ldots, n$.

Clearly, the set of <u>Operating States</u> for the system consists of all those states whose representative vector S is greater than or equal to S_m. All other states correspond to system failures.

The reliability of the micro-computer system is defined to be the sum of the reliabilities of all the Operating States. Thus the failure-to-exhaustion probability can be computed once the Operating States of the system are enumerated.

For example, consider an architecture which consists of 10 processing units, 8 shared memory modules, 6 buses and 5 clocks. The minimum operating configuration of this architecture consists of 7 processing units, 6 shared memory modules, 4 buses, and 3 clocks. Let Rp, Rm, Rb, and Rc denote the reliability of a processing unit, a memory module, a bus, and a clock, respectively. The Initial State of the system is given by:

$$S_o = (10, 8, 6, 5)$$

and then the Minimal State is given by:

$$S_m = (7, 6, 4, 3)$$

The reliability of an Operating State S' = (9, 6, 5, 3) is given by:

$$R(S') = \binom{10}{1} R_p^9 (1 - R_p) * \binom{8}{2} R_m^6 (1 - R_m)^2 * \binom{6}{1} R_b^5 (1 - R_b) * \binom{5}{2} R_c^3 (1 - R_c)^2$$

and the reliability of the system is given by the sum of the reliabilities of all states S that satisfy:

$$S_m \leq S \leq S_o$$

4.3.3 Imperfect Coverage Model

In this model, the time required to detect an error and recover from it (e.g., by isolating the failing components, reconfiguring the hardware architecture, and re-allocating its functions) is considered. Since this time is finite, it is probable that one or more other components will fail before the recovery actions are completed. This may or may not lead to a total system failure, depending on the extent and complexity of the recovery mechanism.

The above situation can be best modelled by a Markovian chain which consists of the following states:

(i) The Start-Up (all components good) State which has a given initial probability that the system is initially fault-free.

(ii) A set of Intermediate States in which one or more components are in failure but undergoing detection and recovery. It is assumed here that complete recovery will return the system to the Start-Up State, i.e., that there are sufficient spare parts. This assumption is valid as long as the period of time being analysed is relatively short and that, in the long run, system reliability is dominated by the failure-to-exhaustion.

(iii) The set of failing states are lumped together into one Failure State. This state is reached from any of the Intermediate States when an additional failure occurs which hampers recovery and leads to a catastrophic failure.

4.4 Resource Usage Models

Modelling of the resource usage in a multiprocessor system can be done at different levels of detail. However, the methodology for performing the analysis is the same at all levels of detail as shown in Figure 4.3. From the Design Description, the Loading and Resource Descriptions are obtained by constructing suitable models. These descriptions are combined with the algorithms in the Resource Usage Model to produce the Resource Usage Estimates. These results are Interpreted and Design Changes made, thus causing the design process to cycle until a design with acceptable resource usage is produced.

When the resource usage is modelled at lower levels of detail, the Design, Loading and Resource Descriptions become increasingly complex. The following sections progress from the high to low levels of detail. The higher levels provide more general information on the total amount of resources needed, while the lower levels yield more information on how individual resources are used.

4.4.1 Simple Totals Model

The simplest assessment of resources can be made by just summing the load and comparing it to the available or postulated levels of resources, without considering how the components of the load will be assigned to the individual units of resources. This type of analysis gives gross estimates of system sizing and resource utilization.

Examples of Loading, Resource Descriptions and Resource Usage Estimates are shown in Figure 4.4. The Loading Description lists all software tasks and buffer areas. For each one, the required resources such as processing time, ROM and RAM are estimated. The individual resource requirements are summed to estimate the total required amount of each resource type. In the Resource Description, the resource types are listed. The capacity of each resource component is multiplied by the number of copies to give the total capacity of each resource type. Next the total capacities are compared to the required amounts of each resource in the

Resource Usage Estimates. The resource utilization is the percentage ratio of required resources to capacity, and the spare capacity is the difference between required resources and the capacity.

4.4.2 Effects of Allocation Model

The next step is to allocate the individual loads to individual resources. The Design Policies from the Architecture Selection stage place constraints on how this allocation is done. Examples of Allocation Constraints are:

(i) One single board computer (SBC) may read the RAM of another SBC but may not write into it.

(ii) The private data of a software task must be stored in the RAM of the SBC which executes the task.

(iii) The code for all reliability critical tasks must be stored on at least two SBC's.

Further Allocation Constraints are derived from the Design Description. These constraints can be summarized in an Access Graph as shown in the example in Figure 4.5. The Access Graph shows the inter-task communication requirements and the shared buffer areas. In this example, the In TV task receives 0.25 K byte messages from the Antenna Attitude Control task and sends 0.10 K byte messages back. The In TV task also requires access to the two Video Buffers. The memory requirements for this task are estimated as 2 K bytes

ROM and 1 K bytes RAM. The resource, message passing and buffer access requirements of the other tasks are similarly described by the diagram.

In addition to the Allocation Constraints, the effects on resource usage caused by the allocation of the software tasks to hardware resources must be identified. Examples of these Allocation Effects could be:

(i) If two tasks are dedicated to the same SBC, then their inter-task messages can be stored in their SBC RAM. If their messages are stored in the common memory, then a 0.01 m sec + 0.1 m sec per 1 K byte of message will be added to the common bus load.

(ii) If tasks exchange messages by copying from one local RAM to another, then a 0.02 m sec + 0.1 m sec per K byte of message will be added to the common bus load. As well, RAM space on both SBC's must be reserved for the message.

(iii) If a processing unit executes code from common memory, then the processing unit will run 25% slower and a common bus load of 0.2 m sec per 1.0 m sec of processing time will be generated.

(iv) If the processing unit executes code from a local RAM instead of ROM, then the processing unit will run 35% slower.

Based on the Allocation Constraints and Effects, the Loading Description can be created. An example is shown in Figure 4.6(a). Compared to the Loading Description in Figure 4.4, the allocation or assignment of tasks to individual hardware components is shown. The requirements for the hardware components are summed to produce the total load on each. The Resource Description also shows more details associated with the resources (Figure 4.6(b)). The individual hardware components are identified and the common bus which was not considered in the Simple Totals Model is included. The Resource Usage Estimates then show the utilization and spare capacity of each hardware component (Figure 4.6(c).

4.4.3 Effects of Dynamic Interaction Model

Two aspects of dynamic interaction are important:

1. Process Flow - Certain actions must be taken (or events occur) before other actions take place(or events occur). The order of hardware and software tasks and their inter-task communication define the data processing flow of the system.
2. Resource Scheduling - The method for allocating resources in real time between competing tasks can have a large effect on the ability of the system to meet its real time requirements. The interaction between

Resource Scheduling and Process Flow affect the overall utilization of the resources.

There are two methods for coping with the effects of dynamic interaction:

(i) Apply Rule of Thumb Utilizations - Because the undesirable effects of dynamic interaction usually only occur when one or more of the resources are heavily utilized, a rule of thumb may be employed such as:

"No processing unit or shared bus may have greater than 70% average utilization over a system cycle".

By avoiding high loading of any resources, bottlenecks can be prevented. The maximum utilization levels can also be set by a worst case analysis of real time events and system loading [MELL80]. The advantage of Rule of Thumb Utilizations is that they are easy to apply. Given that resource requirements are usually not known precisely until the near completion of the system implementation stage, comparisons to rules of thumb often provide as much precision as is possible in the early design stages.

(ii) <u>Simulate the Dynamic Behaviour</u> - A simulation model of the system can be constructed and timing and statistical performance measures obtained from it. This method of evaluating the resource usage is only appropriate when there is accurate data on the resource requirements for each software task. Thus this method of estimating resource requirements is most appropriately used in the later design and implementation stages when this data is available. The advantages of simulating the dynamic behaviour are: (1) more accuracy is obtained in the resource usage estimates, and (2) the ability to discover hidden flaws in the design due to actual timing and resource scheduling problems.

The Rules of Thumb Utilizations can be built into the Effects of Allocation Model (Section 4.4.2) to:

(i) flag (warn the designer) when the utilization of a resource exceeds the approved threshold, and/or

(ii) recalculate the resource utilizations based on first order effects of Resource Scheduling and bottlenecks.

If more accurate information on resource usage is required, then a CAE tool which simulates the micro-computer system is needed. There are two possible types of simulator tools which could be provided to the designer:

(i) A General Purpose Simulation Language - The designer could use this tool to build a simulation model which corresponds to his design and then obtain the resource usage information from this model.

(ii) A Special Purpose Resource Simulator - The designer could feed his Design Description directly into this tool and then automatically receive the resource usage information. Because this type of CAE tool is specialized for a particular architecture, it can only be built once the architecture has been selected.

4.5 Areas for New or Improved CAE Tools

In this section, areas for the development of completely new performance/reliability CAE tools, and the improvement of existing ones are identified. These tools are separated into two categories:

(i) tools which are independent of any particular multiprocessor architecture, and

(ii) tools which are customized for a given multi-processor architectural scope (Architecture Dependent Tools).

Both categories of tools are useful and it is desirable that the designer would have access to a full complement of tools from both categories. The Architecture Independent tools are particularly useful in the Architecture Selection stage when the computer architecture and Design Policies are being formulated. These tools can also be employed in the Detailed Design stage to optimize the design. The architecture independence feature means that these tools can be re-used in wide variety of computer design projects which employ different architectures.

The Architecture Dependent tools are useful primarily in the Detailed Design stage. Because they are "customized" for particular architectures, they can perform a larger portion of the Abstraction, Interpretation and Design Change steps, and thus provide a more automated design environment.

4.5.1 Architecture Independent CAE Tools

4.5.1.1 ADA Based General Purpose Simulation Language

Many General Purpose Simulation Languages, GPSS, SIMULA, SIMSCRIPT, GASP, ..., already exist. Improving on these languages and incorporating an ADA base could lessen the cost and time needed to implement specific simulation models. If ADA is also the language used for system specification and/or software implementation, then the incorporation of the ADA

syntax and language constructs into the simulation language would mean:

(i) less time wasted on learning a multitude of computer languages, and

(ii) easier and more reliable translation of the simulation results into the specifications and implementation.

An ADA Based General Purpose Simulation Language could also be used to construct the Dynamic Resource Usage Analysis Tool (Section 4.5.2.3) which is useful in the later design and implementation stages.

4.5.1.2 Exhaustion of Spares Analysis Tool

This tool incorporates the Exhaustion of Spares Model (Section 4.3.2) in an automated package. The designer supplies the parameters for the model and the CAE tool performs the calculations.

4.5.1.3 Imperfect Coverage Analysis Tool

This tool incorporates the Imperfect Coverage Model (Section 4.3.3) in an automated package. As with the previous CAE tool, the designer supplies the parameters for the model and the tool performs the calculations.

4.5.1.4 General Reliability Analysis Tool

This tool is a more comprehensive combination of the previous two CAE tools. By integrating all three reliability models, Component Reliability, Exhaustion of Spares, and

Imperfect Coverage, this tool can provide a more complete service to the designer. In particular, the designer is relieved of the chore of transferring data between CAE tools for the individual models. As before, the designer supplies the parameters for the models and the tool performs the calculations.

4.5.1.5 Resource Allocation Analysis Tool

The Simple Totals Model (Section 4.4.1) and the Effects of Allocation Model (Section 4.5.1) are automated in this tool. One of the main purposes of this tool is the generation of up-to-date management reports on the expected usage of the micro-computer resources. This information is crucial to the resource management decisions [LARM77] which must be made as the hardware and software development teams progress through the design and implementation phases.

The Resource Allocation Analysis Tool automates the production of the tables shown in Figures 4.4 and 4.6. The user first enters a list of software tasks and buffers, and another list of hardware resources. The CAE tool then builds the tables for Loading and Resource Descriptions and prompts the user for the data entries. From this data, the tool performs the mathematical calculations and outputs a table showing the Resource Usage Estimates. As the project progresses from the Architecture Selection stage through Initial and Detailed Design to implementation, the user can

update the lists of software tasks, buffers and hardware resources as well as the entries in the tables. The tool then produces up-to-date summaries of the resource usage. Two types of summaries are produced: (1) a simple comparison of total requirements compared to total capacity (Figure 4.4), and (2) a detailed analysis of the loading of each hardware component (Figure 4.6).

4.5.2 Architecture Dependent CAE Tools

4.5.2.1 Hardware Reliability Analysis Tool

With this tool, the designer assembles a hardware model from a database of pre-defined components such as processing units, memories, majority voting circuits, and bus structures. The tool can then use the pre-defined reliability characteristics of the components to:

(i) calculate the overall reliability of the hardware, or

(ii) select the number of redundant modules which are necessary to meet the overall hardware reliability specifications, or

(iii) calculate how reliable the modules must be to meet the overall hardware reliability specifications.

A major feature of the Hardware Reliability Analysis Tool is the support of Structure/Behaviour Design Partitioning. The Structure is the component and module

interconnections specified by the designer. In Figure 4.7(a), the Structure is illustrated in a "bottom-up" order. The lowest level of detail, the Processing Unit, ROM, and RAM Modules are described first in Sections (a), (b) and (c). These modules then are configured into the Single Board Computer Module in Section (d) (Figure 4.7(b)). Finally, the top level view of the Structure is shown in Section (e) (Figure 4.7(c)). In a "top-down" design approach, the order of the previous design sections is reversed. The top-level section (e) is drawn first and is subsequently decomposed into the lower level sections.

The Behaviour, which is analysed by this CAE tool, is the reliability characteristic of the components, modules and system configurations. The Behaviours of the components and modules are stored in the CAE tool's data base. By combining the system structure specified by the designer with the component, and module Behaviours from the data base, the tool formulates (by a pre-programmed algorithm) the parameters of the reliability models (from Section 4.3). The tool then computes the reliability characteristics of the complete system.

The most powerful version of this tool would also adjust the Structure (such as the number of redundant processing units), or calculate the component reliabilities (Behaviour) which are necessary to meet the overall system reliability

specification. The modelling power of the tool can be expanded by adding more components and modules to the data base.

4.5.2.2 Static Resource Usage Analysis Tool

This tool automates the Simple Totals Model (Section 4.4.1) and the Effects of Allocation Model (Section 4.4.2). The major enhancement of the tool over the Resource Allocation Analysis Tool (Section 4.5.1.5) is the automatic production of the Loading and Resource Descriptions from the Design Description. Since the calculation of the data entries in the Loading Description (Figure 4.6) is the most laborious step in analysing the resource usage, its automation is a significant improvement. When using the Static Resource Usage Analysis Tool, the user inputs the Design Description and receives from the tool the Loading and Resource Descriptions as well as the Resource Usage Estimates.

The most powerful version of this tool would also:

(i) optimize the allocation of the load to the individual resources,

(ii) flag resource utilizations which exceed specified thresholds,

(iii) refine the Resource Usage Estimates based on the first order effects of Resource Scheduling and bottlenecks, and

(iv) select the amount of each resource type which is required to meet the resource usage specifications.

As the project progresses from the Architecture Selection stage through Detailed Design to implementation, the user can update the lists of software tasks, buffers and hardware resources as well as input new Design Descriptions. The tool then produces up-to-date summaries of the resource usage. Two types of summaries are produced: (1) a simple comparison of total requirements compared to total capacity (Figure 4.4), and (2) a detailed analysis of the loading of each hardware component (Figure 4.6).

4.5.2.3 Dynamic Resource Usage Analysis Tool

The purpose of this tool is to estimate the resource usage by simulating the dynamic interactions of the software tasks and hardware resources. It automatically constructs the simulation model from the Design Description provided by the designer. Based on the results of executing the model, the tool estimates:

(i) the utilization of individual resources,

(ii) the timing diagram which shows when each hardware resource is used by each software task,

(iii) the unused resource capacity caused by tasks holding onto some resources while waiting for other resources,

(iv) the location of system bottlenecks.

The most powerful version of this tool would also:

(i) optimize the allocation of the load to the resources, and

(ii) select the amount of each resource type which is required to meet the resource usage specifications.

In order to produce accurate and useful results, this tool requires that the user provide accurate estimates of the resource requirements of individual software tasks, as well as a moderately detailed Design Description. Since this information is not normally available until later in the design process, this tool is primarily useful in the Detailed Design Stage.

4.6 Summary

When choosing a CAE tool or a package of CAE tools for the evaluation of performance and reliability, the major factors to be considered are the:

(i) Selectivity and the

(ii) Degree of Automation of the tool(s).

The Selectivity refers to range of multi-processors designs for which the tool can be used. Tools which are architecture dependent are specialized for a small range of multi-processor architectures while tools which are architecture independent are useful in a wide range of computer designs. The Degree of Automation refers to the amount of the design process which is performed by the tool.

In this section, the proposed CAE tools (Section 4.5) were classified as either Architecture Dependent or Architecture Independent. The classification was not intended to imply a sharp boundary. Quite the contrary is true. The tools of both classifications are built on the same fundamental performance/reliability models (Sections 4.3 and 4.4). The difference between the two classifications is the amount of abstraction, interpretation, and design changes performed by the tool. When these steps are performed primarily by the designer, an Architecture Independent tool is defined. The amount of architecture dependency and thus the level of Selectivity can be changed by varying the responsibility for performing the abstraction, interpretation, and design change Steps between the designer and the CAE tool.

In a complete CAE tool package, the performance/reliability tools would be integrated with the specification and verification tools. The point of integration is the design description documentation which is produced by the specification tool and used by the performance and reliability tools. There must be enough information in the design description documents to perform the Abstraction and parameter estimation for the performance/reliability models. If the Abstraction step is performed manually, then the precise contents of the design description documents is not critical so long as they are clear and easy to read. But if

the Abstraction step is performed automatically by the performance/reliability tool, then the accuracy and completeness of the design description is critical to the integration of the CAE tool package.

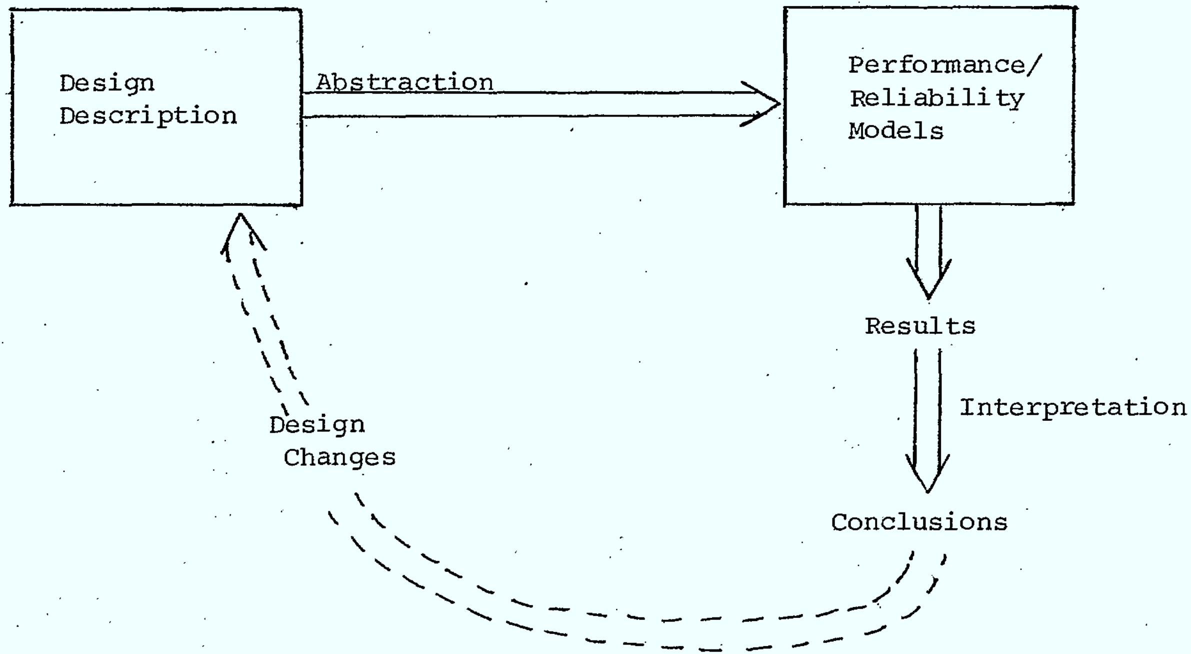
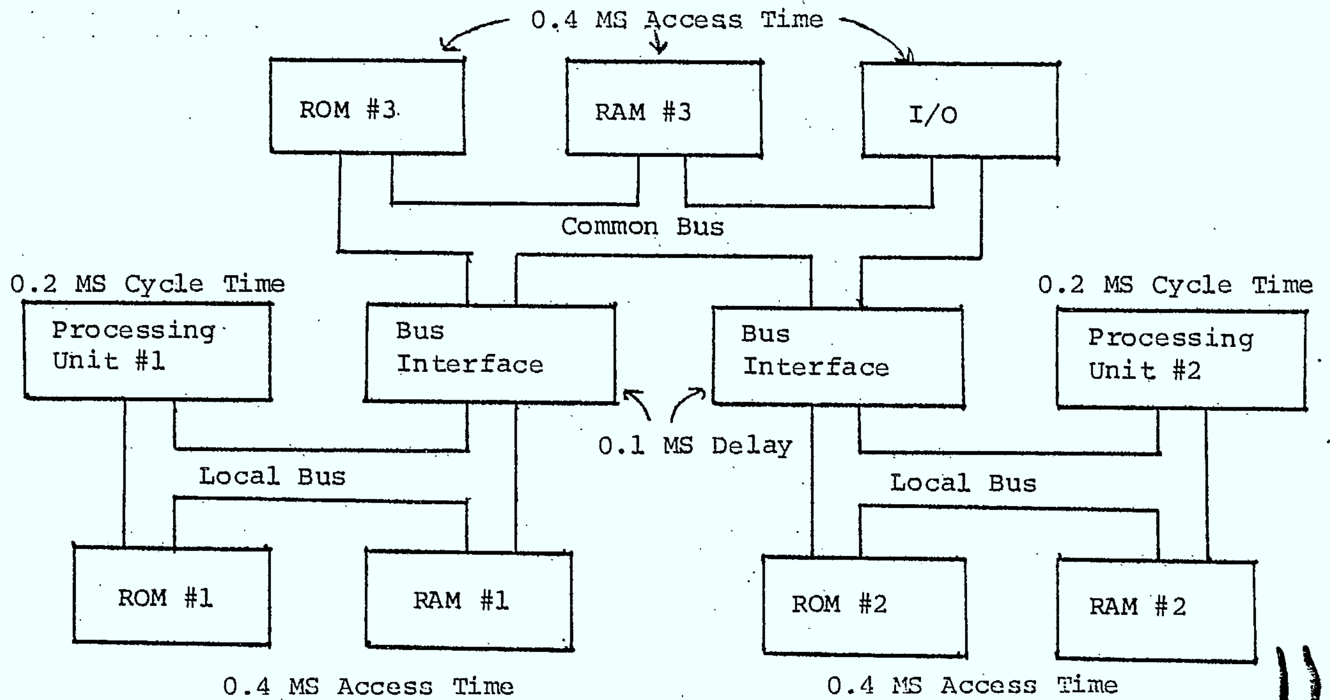
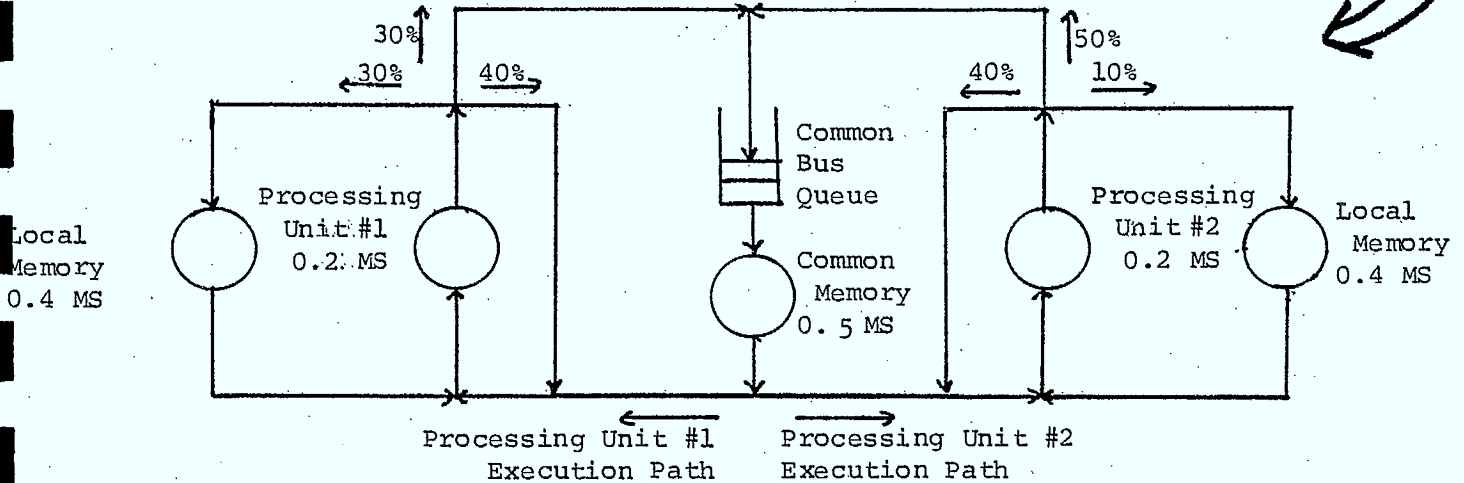


FIGURE 4.1 - PERFORMANCE/RELIABILITY DESIGN METHODOLOGY

(a) Design Description



(b) Performance Model



(c) Interpretation

Results:
- Processing Unit #1 executes 2.0 million cycles/sec
- Processing Unit #2 executes 1.3 million cycles/sec
- Average Common Bus Queue length = 0.6

Back to Design Description

Conclusions:
1. More processing throughput is needed.
2. Too much time is wasted accessing common memory.

Design Changes: Decrease the Access Time of ROM #3, RAM #3 and I/O to 0.2 MS.

FIGURE 4.2 - HARDWARE PERFORMANCE DESIGN METHODOLOGY EXAMPLE

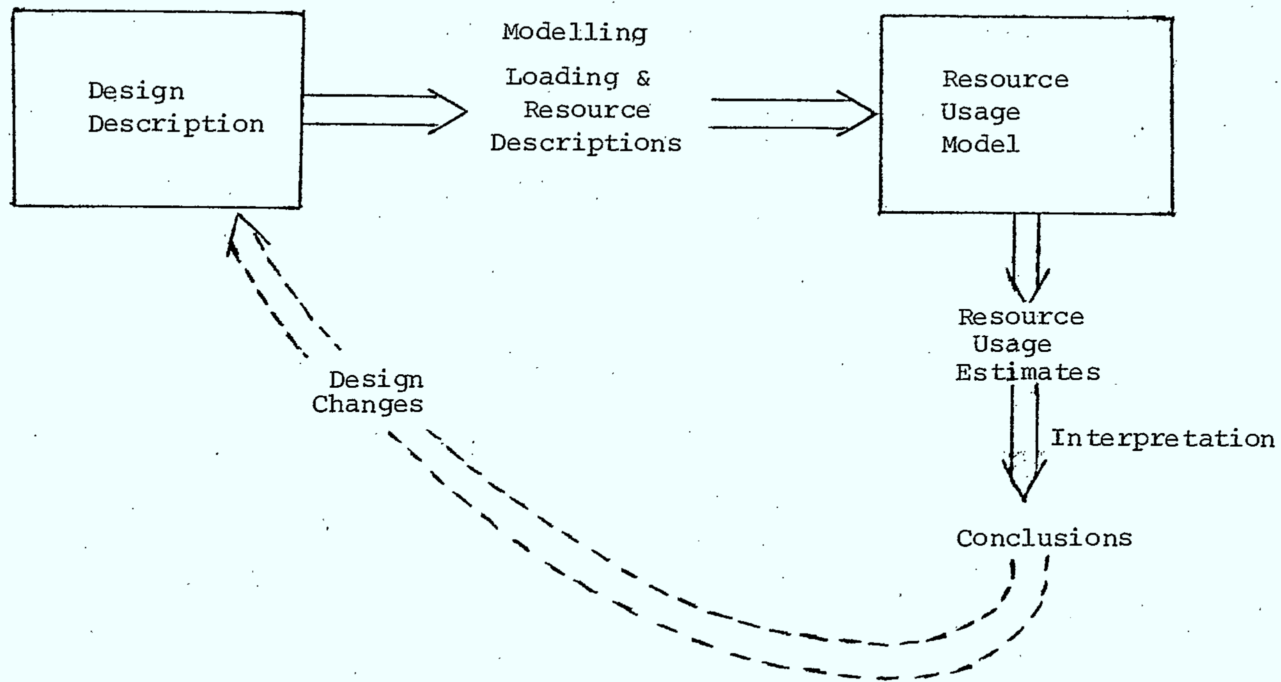


FIGURE 4.3 - RESOURCE USAGE DESIGN METHODOLOGY

(a) Loading Description:

Task or Buffer	Processing Time	ROM	RAM
1. Antenna Attitude Control	6 msec/cycle	6.0K	2.0K
2. In Radio	10	10.0	5.0
3. Out Calib	2	2.0	1.0
4. Out Actuate	1	0.5	0.5
5. In Sensor	2	0.5	0.5
6. In TV	8	2.0	1.0
7. Out Radio	4	1.5	2.0
8. Messages			2.75
9. Video Buffers			24.0
	33 msec/cycle	22.5K	38.75K

note: 1 cycle = 25 msec

(b) Resource Description:

Resource Type	Number	Total Resource
Processing Unit	2	50 msec/cycle processing time
8K ROM Module	3	24K
16K RAM Module	3	48K

(c) Resource Usage Estimates

Resource Type	Utilization	Spare Capacity
Processing Units	$\frac{33}{50}$ X 100 = 66%	50 - 33 = 17 msec/cycle
ROM	$\frac{22.5}{24}$ X 100 = 93%	24 - 22.5 = 1.5K
RAM	$\frac{38.75}{48}$ X 100 = 81%	48 - 38.75 = 9.25K

FIGURE 4.4 - EXAMPLE SIMPLE TOTALS MODEL

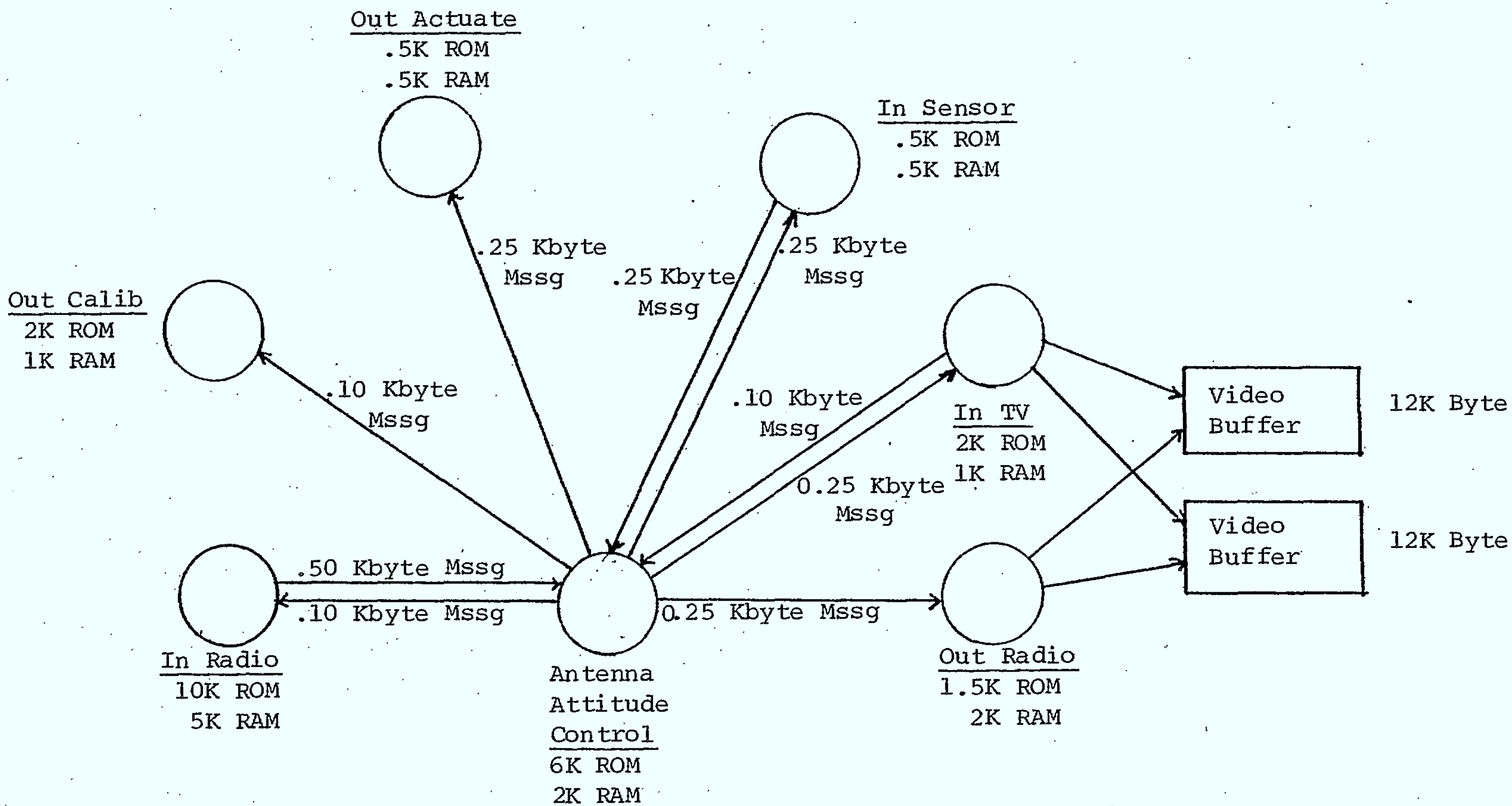


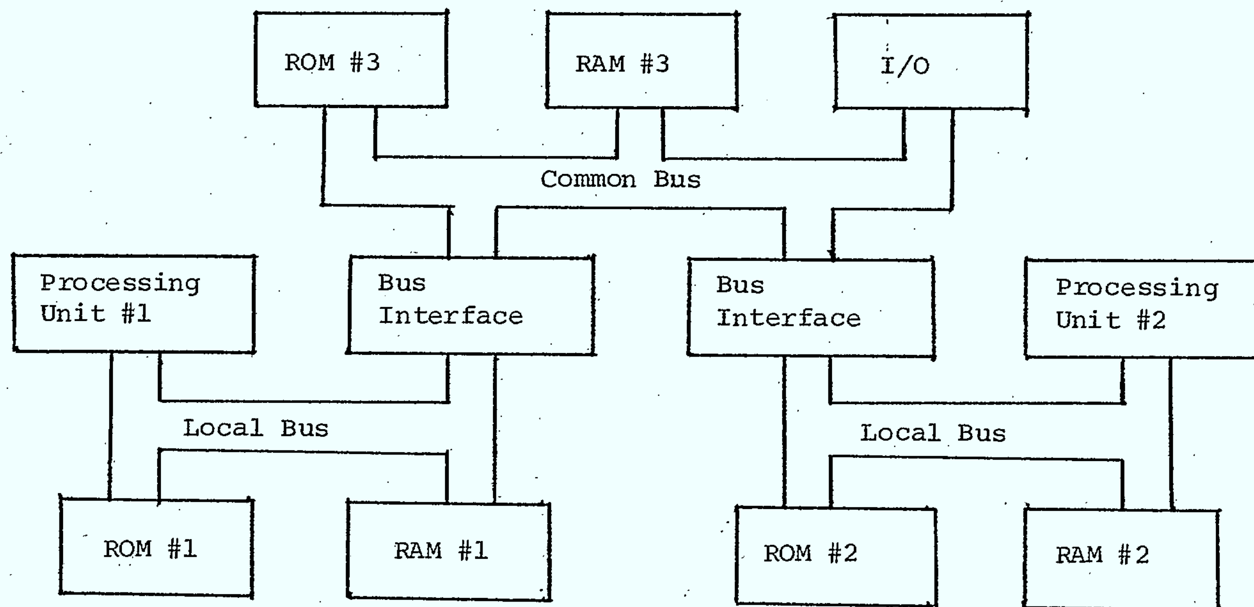
FIGURE 4.5 - EXAMPLE ACCESS GRAPH

(a) Loading Description

Task or Buffer	Processing Unit #1	Processing Unit #2	ROM Module #1	ROM Module #2	ROM Module #3	RAM Module #1	RAM Module #2	RAM Module #2	Common Bus
1. Antenna Attitude Control	6		6.0			2.0			
2. In Radio		10		8.0	2.0		4.0	1.0	1.5
3. Out Calib	2		2.0			1.0			
4. Out Actuate	1				0.5	0.5			0.5
5. In Sensor		2			0.5	0.5			1.0
6. In TV		8			2.0			1.0	8.0
7. Out Radio	4				1.5	2.0			6.0
8. Messages						.75		2.0	3.0
9. Video Buffers							12.0	12.0	
	13 msec/ cycle	20 msec/ cycle	8.0K	8.0K	6.5K	6.75K	16.0K	16.0K	20.0 msec/ cycle

Figure 4.6(a) - EXAMPLE EFFECTS OF LOADING MODEL

(b) Resource Description



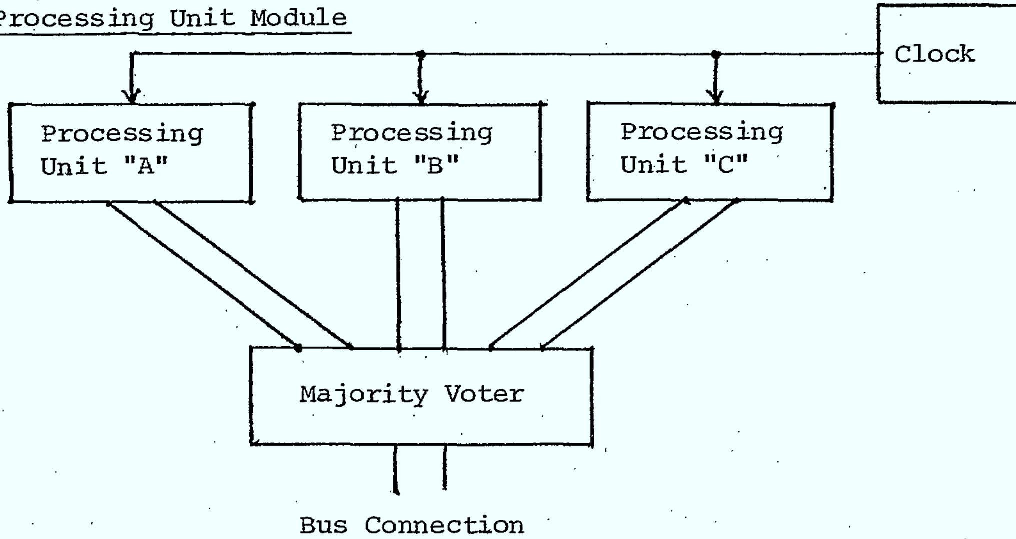
Resource Unit	Amount of Resource
Processing Unit #1	25 m.sec/cycle
Processing Unit #2	25 m.sec/cycle
ROM #1	8K
ROM #2	8K
ROM #3	8K
RAM #1	16K
RAM #2	16K
RAM #3	16K
Common Bus	25 m.sec/cycle

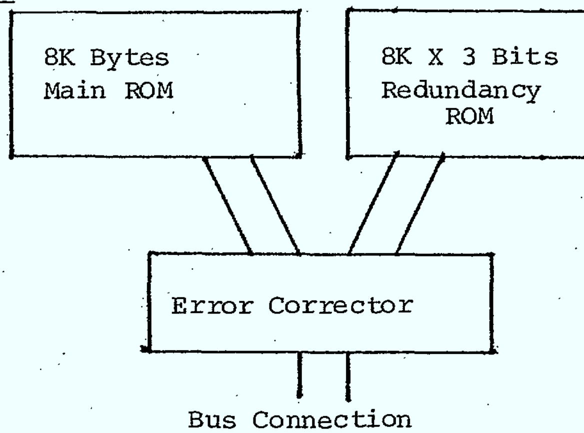
Figure 4.6(b) - EXAMPLE EFFECTS OF LOADING MODEL

(c) Resource Usage Estimates

Resource Unit	Utilization	Spare Capacity
Processing Unit #1	13/25 X 100 = 52%	25 - 13 = 12 m.sec/cycle
Processing Unit #2	20/25 X 100 = 80%	25 - 20 = 5 m.sec/cycle
ROM #1	8.0/8.0 X 100 = 100%	8.0 - 8.0 = 0
ROM #2	8.0/8.0 X 100 = 100%	8.0 - 8.0 = 0
ROM #3	6.5/8.0 X 100 = 81%	8.0 - 6.5 = 1.5K
RAM #1	6.75/16.0 X 100 = 42%	16.0 - 6.75 = 9.25K
RAM #2	16.0/16.0 X 100 = 100%	16.0 - 16.0 = 0
RAM #3	16.0/16.0 X 100 = 100%	16.0 - 16.0 = 0
Common Bus	20.0/25.0 X 100 = 80%	25.0 - 20.0 = 5.0 m.sec/cycle

Figure 4.6(c) - EXAMPLE EFFECTS OF ALLOCATION MODEL





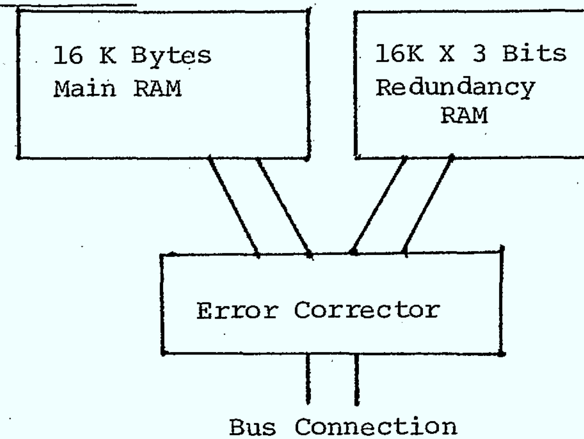


Figure 4.7(a) - HARDWARE RELIABILITY ANALYSIS TOOL EXAMPLE

(d) Single Board Computer (SBC) Module

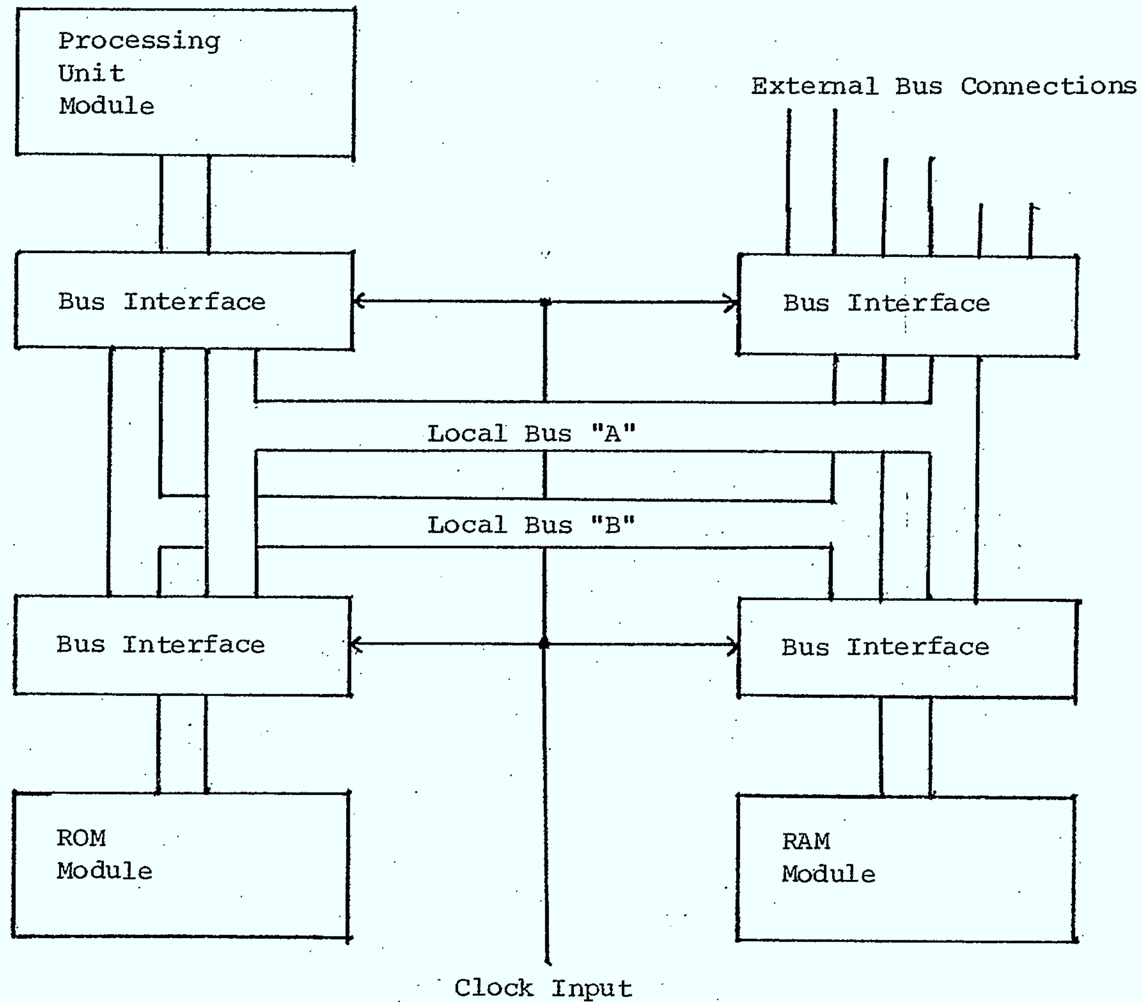


Figure 4.7(b) - HARDWARE RELIABILITY ANALYSIS TOOL EXAMPLE

(e) Multi-Microcomputer System

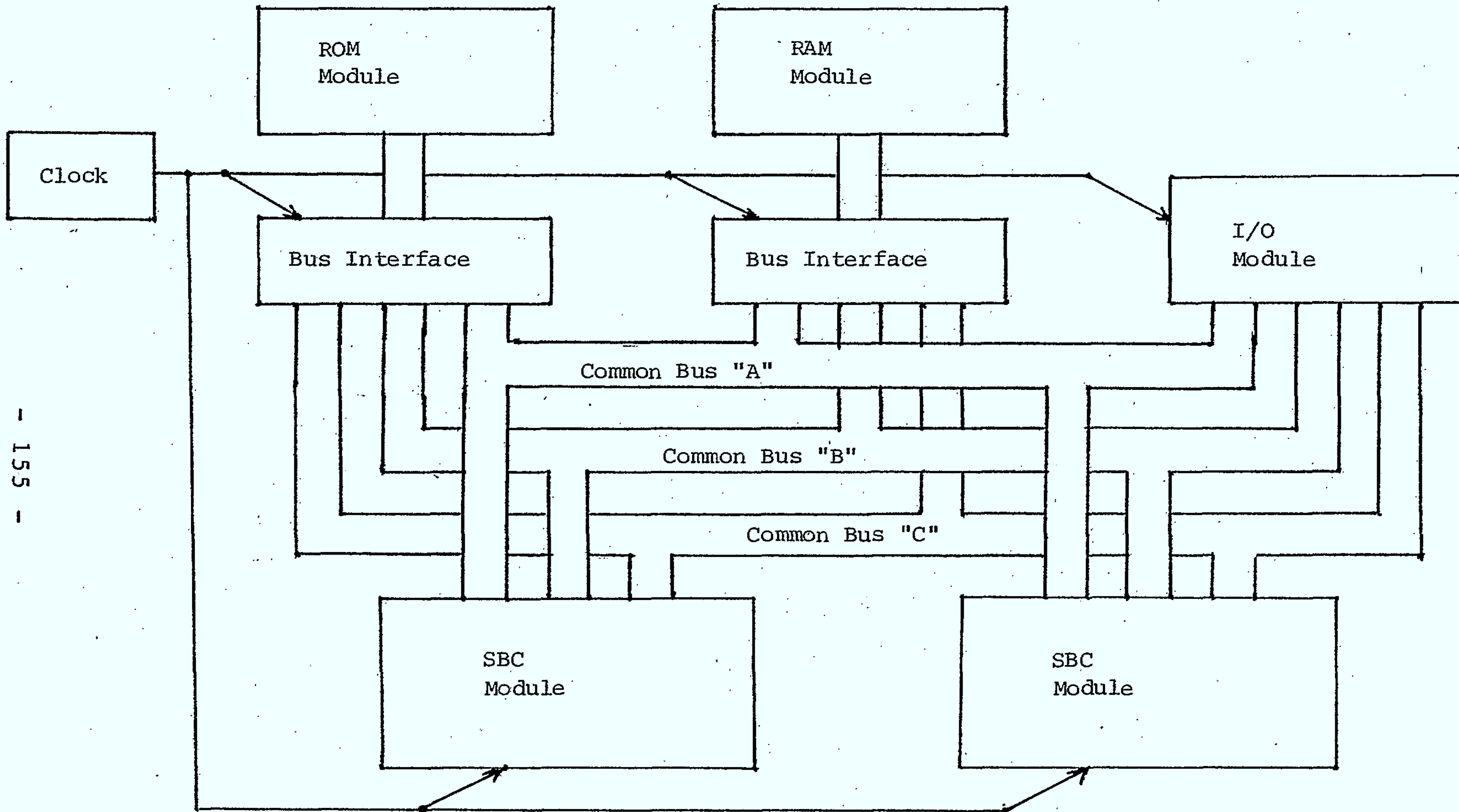


Figure 4.7(c) - HARDWARE RELIABILITY ANALYSIS TOOL EXAMPLE

5.0 INTEGRATION WITH EXISTING TOOLS

5.1 Introduction

As indicated earlier, the development of a fully integrated set of CAE tools requires significant effort and can be accomplished in a reasonable period of time only if built over a solid foundation of existing tools. This section examines closely the basic characteristics of existing tools at the architecture design level and the levels below it. The main objectives are:

- to determine the suitability of one or more of the existing tools for possible integration with the high level specifications tool;
- to determine the necessary procedures and interfaces for integrating these tools; and
- to select one or more of the existing CAE tools, if possible, for procurement and implementation.

The use of existing tools to build a foundation for the integrated set of tools would provide two significant advantages. First, they would shorten the development cycle of the CAE tools, perhaps by as much as half of the total anticipated four or five years required for complete system development. Second, the use of these tools would provide significant input into the design of higher level tools by broadening the knowledge base of those systems.

5.2 Transition between Different Tools

Once a design engineer has completed his functional specification using the tool proposed in Chapter 3 and is satisfied with the simulation and performance results, as discussed in Chapter 4, he must then refine his design to a lower level of detail: the architecture level. At this level, the engineer will have to use one of the existing tools that were defined in a previous report [MAHM82]. Given a specific tool, the engineer must rewrite the definition of his design in the language of the new tool; in particular, he must:

- rethink his design in terms of the concepts used by the tool; for example, some tools are based on state models, others on data flow models, etc.;
- augment the design by the specifying to the level required; i.e., if progressing from the functional to the architectural levels, the designer must now specify some of the hardware components and their interconnections;
- rewrite the design in the language of the tool;
- re-simulate and re-evaluate this design.

This conversion effort from one tool to another could perhaps be automated. As an example, program translators exist that allow almost fully automatic conversion of programs from RPGII to COBOL or COBOL to PL/I. In our case,

the objective will be to find a translator that accepts the ADA specification and translates it into the input language of an existing tool. This translator would be feasible if:

- the language of the second tool is sufficiently close to ADA (i.e., Pascal);
- the definition concepts or the second tool is sufficiently close to the hybrid data flow/top down decomposition approach used in the specification level.

There are a number of pitfalls to these transition efforts which cannot be fully avoided. These are:

- the overall design effort is multiplied by the necessity of learning several environments and of recoding the definition of the design for each tool;
- translation errors can be introduced in between levels during the conversion process; these errors may be difficult to notice since there are no mechanisms to measure the correctness of the translation;
- it may be difficult to determine the source of errors detected during the evaluation of the architecture level, for example, if the errors were caused by an improper translation or by an incorrect architecture design feature added at that level;

- results of simulations performed in each tool may be difficult to compare since simulation measurements may be based on a different approach; this would make tracking the design from top to bottom difficult.

5.3 Selection of Existing Tools

Various existing tools are described at length in [MAHM82] and will be reviewed here in order to select the best tools possible for the implementation of a complete integrated design environment. Our aim is to select tools that will be useful at all the levels of design in order to complement the specification tool described in Section 3. Tables 5.1 and 5.2 outline some of the characteristics of each tool while Table 5.3 summarizes the comparison.

The functional characteristics of each tool is reviewed in Table 5.1. The characteristics of concern are:

- the aim of the tool: these existing tools were designed for specific purposes: the closer that purpose is to ours, the likelier the tool will be useful;
- the level of usage: tools can cover one or more levels of design; obviously wider ranging tools are preferred since they minimize the transition efforts;

- multi-level capability: tools that span more than one level are especially useful if the designer is not forced to define all system components at the same time, i.e., he should be able to work out some portions of the design to lower levels while leaving other segments at a higher level.
- performance evaluation: a good tool should include performance evaluation mechanisms.

The implementation characteristics of each tool is detailed in Table 5.2. The characteristics of concern are:

- design specification language: the language used to specify the design;
- tool implementation language: the language used to program the tools;
- operating system and processor type under which the tool operates;
- developer: the organization where the tool is available from;
- availability: whether the tool can be obtained or not.

Finally, in order to select a tool, three primary criteria were chosen: Table 5.3 rates the tools with respect to each criteria. The selection criteria is based upon the characteristics described earlier and are:

1. Completeness: a good tool should offer a complete working environment. Therefore, it should cover as many levels of design as possible, it should provide multi-level capability and it should provide performance evaluation mechanism.
2. Ease of Interface: a good tool should provide for an easy transition between the functional specification level and the other levels. Therefore, the tool should have been designed for architectural modelling and simulation. It is also desirable that the language in which the tool is written be compatible with Ada as an implementation language.
3. Implementation Potential: the tool should be relatively easy to install on the same computer as the one selected for the development of the functional specification tool, (e.g., DEC's VAX 11/780). Ideally, the tool should also be well documented and fully supported by the vendor.

A review of Table 5.3 shows that the AIDE package represents the best choice. However, the Bell Laboratories have decided not to release it at this time. The next attractive alternative is the ADLIB/SABLE facility which will be available commercially in the near future and will be run on the VAX minicomputer. Once augmented by some of the performance and reliability tools discussed in Section 4, the ADLIB/SABLE will provide system engineers with an excellent environment to pursue the design efforts after the functional specification phase is completed.

TOOL	AIM OF TOOL	LEVEL OF USAGE	MULTILEVEL CAPABLE	PERFORMANCE EVALUATION
AIDE	modeling and simulation for development of computer architectures	architectural and lower	Yes	Yes
CASL	design and documentation for VSLI implementation	register transfer	No	No
N.mPc	multi-processor design and evaluation	register transfer	No	Some
ADLIB/ SABLE	multi-level design	architectural and lower	Yes	Some

Table 5.1 Existing Tool Functional Characteristics

TOOL	DESIGN SPECIFICATION LANGUAGE	TOOL IMPLEMENTATION LANGUAGE	OPERATING SYSTEM	CPU	DEVELOPER	AVAIL-ABILITY
AIDE	C augmented	C	UNIX	VAX11/780	Bell Laboratories	No
CASL	CASL	-	-	B1800	U. of Utah	No
N.mPc	ISP' and assembler	C	UNIX	PDP 11/70	Case Western U.	Unknown
ADLIB/ SABLE	Pascal augmented and SDL	Pascal	TOPS or VMS	DEC 20 or VAX11/780	Stanford U. or Commercial Company	Yes

	Completeness	Ease of Interface	Implementation Potential
AIDE	Excellent	Good	Medium
CASL	Poor	Poor	Poor
N.mPc	Medium	Medium	Unknown
ADLIB/ SABLE	Good	Good	Good

Table 5.3 Existing Tool Selection Evaluation

6. Summary and Further Work

6.1 Summary

The study reported here examined the role of existing computer assisted engineering tools in supporting the application of current design methodologies used in the development of multiprocessor systems. Specifically, the study focused on the issue of augmenting and enhancing existing tools to generate an integrated set of multi-processor design and simulation tools that can be useful throughout the various phases of the design. The following is a summary of the major results and conclusions of the study.

The design process of multiprocessor systems can be described, in a top-down approach, as consisting of six phases:

1. The Requirements Specification phase,
2. The Functional Components Definition phase,
3. The Architectural Design phase,
4. The System Model phase,
5. The Processing Element Partitioning (Register Transfer Level phase), and
6. The Logic Design (hardware) phase.

A survey of existing tools indicated the availability of many design and simulation tools which satisfy different design needs, depending on the design level (or levels) for

which it is developed. Unfortunately, no one simulator was found to be useful throughout all specification and design phases. This multiple simulator approach has two advantages and several disadvantages. The advantages are:

1. Each simulation can be written in a language tuned for one particular level, and
2. Each simulation tool can optimize its runtime organization for one particular task.

The disadvantages include the following:

1. The design effort is multiplied by the necessity of learning several simulator systems and writing a design in each.
2. The possibility of error is increased as more human manipulation is involved.
3. As the design becomes increasingly fragmented, it becomes impossible to simulate an entire multiprocessor system at a low level of abstraction. Therefore, only small fragments can be simulated at any one time.
4. Each fragment needs to be driven by a supply of realistic data and its output must be interpreted. This may make the software written to serve these needs extremely costly.

Several tools have been developed to overcome the above difficulties and provide the designer with a uniform simu-

lation approach starting at the architecture design level and going down to the register transfer simulation level. Our study indicated that the utility of these tools can be improved substantially by augmenting them with a high level specification package which allows the designer to describe the functional components of the system being designed and to interface this high level description to existing tools at the architectural level. In addition, two design aspects were addressed in augmenting existing tools:

1. Analysis of redundancy and fault-tolerance characteristics must be provided at the architectural levels if the tools are to be useful in the design of spacecraft multiprocessor systems. Several performance analysis models were introduced to serve as basis for reliability and resource scheduling evaluation both at the end of the functional specification level and during the architecture selection phase.

2. A high level specification and verification tool was introduced to bridge the gap between the requirement specification phase and the architecture design and simulation phase. It is proposed that the implementation of this tool be based on the ADA language. The selection of ADA was made for several reasons: it seems to gain wide acceptance and support in the programming community; it supports top-down design and implementation procedures; and

it is capable of describing concurrency and multitasking through a set of designated constructs. The main results of our study of the high level specification and verification tool are:

- The process of obtaining specification was described and guidelines were provided. Decomposition techniques, such as dataflow analysis and functional decomposition were introduced. Ada specification blocks were used to provide a strict formalism.
- The specifications obtained with Ada specification blocks can be transformed naturally into an implementation. This transformation is done in various steps, with each step being a refinement of the previous one.
- Validation techniques were investigated with respect to their applicability to the design methodology. Although no concrete validation system was specified, preliminary validation capabilities were outlined. A more powerful validation system was also suggested as a desirable goal in future research.

The methodology proposed here should be experimented with in the context of a large example. This would assist in improving the methodology itself and would also provide the

necessary framework for the development of computer aided tools such as:

- A graphical aid to decomposition (outlined previously).
- A simulation testbed for the preliminary simulation phase.
- A verifier which could be used on selected sections of the system.

It was also pointed out that in view of the complex nature of the verification process, it would be desirable to gain experience with an already existing system, such as Gypsy. Theory would then be substantiated by practical experience.

Finally, the interface between the proposed high level design tool and existing tools at the architectural level was examined. Several existing tools were investigated in the process of selecting a candidate tool which can be implemented and used later to support the development of an integrated set of tools. The selection was based on several criteria including the ease of interface, completeness, run-time environment and availability and support by the vendor. A design and simulation tool, developed by Stanford University and known as SABLE/ADLIB was identified as a feasible candidate to be used at the architecture and register transfer levels of the design.

6.2 Further Work

The ultimate objective of the work in this area is to assimilate an integrated set of CAE tools which can be utilized in all specifications and design stages of multiprocessor systems, with particular emphasis placed on spacecraft applications. This objective can be achieved through the utilization of existing tools, provided that they are augmented by a high level functional specification tool and a set of performance evaluation packages. To achieve the stated objective, we propose the following work as a logical next step to the definition and specification study reported here:

1. Based on preliminary analysis, a package developed by Stanford University (SABLE/ADLIB) and used as a design and simulation tool for general purpose processors at the Architecture and Register Transfer levels was selected. A detailed study is needed to determine its suitability and utility as a design tool in the special application of spacecraft multiprocessors. If selected following the detailed study, the package must be installed and checked out. It should be noted that the use of this package has been limited so far to research and development applications. The package will be available commercially in the near future.

2. Design of the functional component specification tool defined in this study must be completed in detail prior to the implementation of the tool. The general design will be based on ADA constructs and will follow a general top-down specification approach. The output of this tool must be structured to permit the definition of hardware/software boundaries of the architecture. Initially, the results of this phase will be manually interpreted and used to generate the input to the architecture design and simulations tool.

3. Design of reliability and resource utilization analysis modules to be used with the high level specification tool (at the stage where the hardware/software boundaries are defined). A more refined form of these modules will also be used to augment the design and simulation tool at the architecture selection stage.

REFERENCES

[AFFI79] "The Affirm Reference Library", Gerhart, S., Editor, ISI Program Verification Group, International Science Institute, Marina Del Rey, California, 1979.

[AMBL77] Ambler, A.L. et al "GYPSY: A Language for Specification and Implementation of Verifiable Programs", Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, vol. 12, no. 3, pp. 1-9, March 1977.

[BERG81] Bergland, G.D. "A Guided Tour of Program Design Methodologies", IEEE Computer, vol. 14, no. 10, pp. 13-37, October 1981.

[BOYE79] Boyer, R.S. and Moore, J.S. "A Computational Logic", Academic Press, New York, 1979.

[CHEH81] Cheheyl, M.H. et al "Verifying Security", ACM Computing Surveys, vol. 13, no. 3, pp. 279-340, September 1981.

[COMP81] IEEE Computer, Special Issue on Ada "Ada Programming in the 80's", IEEE Computer, vol. 14, no. 6, June 1981.

[CORY80] Cory, W.E. and VanCleemput, W.M. "Developments in Verification of Design Correctness: A Tutorial", Design Automation Conference, pp. 156-164, 1980.

[CORY81] Cory, W.E. "Symbolic Simulation for Functional Verification with ADLIB and SDL", Design Automation Conference, pp. 82-89, 1981.

[DIJK75] Dykstra, E.W. "Guarded Commands, Non-Determinancy and a Calculus for the Derivation of Programs", Proceedings of the 1975 International Conference on Reliable Software, pp. 2.0-2.13, 1975. (also in CACM, vol. 18, no. 8, 1975).

[DIJK76] Dykstra, E.W. "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[DOD80] United States Department of Defense "Ada Programming Language", Military Standard, MIL-STD-1815, Decembre 1980.

[FLOY67] Floyd, R.W. "Assigning Meaning to Programs", Proceedings of the American Mathematical Society Symposium in Applied Mathematics, vol. 19, Providence, R.I., American Mathematical Society, pp. 19-31, 1967.

[FRAN77] W.R. Franta, "The Process View of Simulation", North-Holland, N.Y. 1977.

[GOOD77] Goodenough, J.B. and Gerhart, S.L. "Toward a Theory of Testing: Data Selection Criteria", in "Current Trends in Programming Methodology", Vol. II, Yeh, R.T. (Editor), Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[GOOD78] Good, D.I., Cohen, R.M. and Hunter, L.W. "A Report on the Development of Gypsy", Technical Report ICSCA-CMP-13, University of Texas at Austin, October 1978.

[GORD75] G. Gordon, "The Application of GPSS V to Discrete System Simulation", Prentice-Hall, N.J. 1975.

[GRIE76] Gries, D. "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs", IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 238-244, December 1976.

[HANT76] Hantler, S.L. and King, J.C. "An Introduction to Proving the Correctness of Programs", ACM Computing Surveys, vol. 8, no. 3, pp. 331-353, September 1976.

[HEND77] Henderson, P. "Structured Program Testing", in "Current Trends in Programming Methodology", Vol. II, Yeh, R.T. (Editor), Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[HILL79] Hill, D.D. "ADLIB: A Modular, Strongly Typed Computer Design Language", Proceedings of the 16th Annual Design Automation Conference, pp. 75-81, 1979.

[HOAR69] Hoare, C.A.R. "An Axiomatic Basis for Computer Programming", CACM, vol. 12, no. 10, pp. 576-583, October 1969.

[HOAR73] Hoare, C.A.R. and Wirth, N. "An Axiomatic Definition of the Programming Language Pascal", Acta Informatica, vol. 2, pp. 335-344, 1973.

[JENS74] Jensen, K. and Wirth, N. "Pascal: User Manual and Report", 2nd edition, Springer-Verlag, New York, 1974.

[JENS78] E.D. Jensen, "The Honeywell experimental distributed processor - An overview", Computer, pp. 28-37, Jan. 1978.

[KEMM80] Kemmerer, R. "FDM-A Specification and Verification Methodology", Proceedings of the Third Seminar on the Department of Defense Computer Security Initiative Program, National Bureau of Standards, Gaithersburg, Maryland, November 1980.

[KING80] King, J.C. "Program Correctness: On Inductive Assertion Methods", IEEE Transactions on Software Engineering, vol. SE-6, no. 5, pp. 465-479, September 1980.

[KNUT68] Knuth, D.E. "The Art of Computer Programming", Vol. I, Addisson-Wesley, Reading, Mass., 1968.

[LAFE81] Laferriere, C. and Mahmoud, S.A. "Ada and Euclid as Programming Languages for Communications Systems", Intellitech Technical Report, Decembre 1981.

[LAMP77] Lampson, B.W., Horning, J.J. et al "Report on the Programming Language Euclid", SIGPLAN Notices, vol. 12, February 1977.

[LARM77] B.T. Larman, "Spacecraft Computer Resource Margin Management", AIAA Computers in Aerospace III Conference, pp. 97-103, 1981.

[LOND77] London, R.L. "Perspectives on Program Verification", in "Current Trends in Programming Methodology", Vol. II, Yeh, R.T. (editor), Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[LUCK79] Luckham, D. et al "Standard Pascal Verifier User Manual", Stanford University Technical Report, STAN-CS-79-731, 1979.

[MAHJ81] Mahjoub, A. "Some Comments on Ada as a Real-Time Programming Language", SIGPLAN Notices, vol. 16, no. 2, pp. 89-95, February 1981.

[MAHM82] S.A. Mahmoud et. al, "A Survey of Computer-Aided Engineering (CAE) Tools for the Design and Simulation of Multiprocessor Systems", Report #INT-82-15, Intellitech Canada Ltd., Ottawa, 1982.

[MELL80] P.M. Mellian-Smith "Permissable Processor Loadings for Various Scheduling Algorithms", Computer Science Lab, SRI International.

[MORI79] Moriconi, M.S. "A Designer/Verifier's Assistant", IEEE Transactions on Software Engineering, vol. SE-5, no. 4, pp. 387-401, July 1979.

[MORR77] Morris, J.H. and Wegbreit, B. "Program Verification by Sub-Goal Induction", in "Current Trends in Programming Methodology", Vol. II, Yeh, R.T. (editor), Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[OKAD80] Okada, K., Futatsugi, K. and Toru, K. "Reliable Program Derivation in Functional Languages by Applying Jackson's Design Method", IEEE Fault-Tolerant Computing, pp. 91-96, 1980.

[PARN72] Parnas, D.L. "A Technique for Module Specification with Examples", CACM, vol. 15, no. 5, pp. 330-336, 1972.

[POLA79] Polak, W. "An Exercise in Automatic Program Verification", IEEE Transactions on Software Engineering, vol. SE-5, no. 5, pp. 453-457, Septembre 1979.

[PYLE81] Pyle, I.C. "The Ada Programming Language", Prentice-Hall International, London, 1981.

[RAMA81] Ramamoorthy, C.V. et al "Application of a Methodology for the Development and Validation of Reliable Process Control Software", IEEE Transactions on Software Engineering, vol. SE-7, no. 6, pp. 537-555, November 1981.

[REYN76] Reynolds, C. and Yeh, R.T. "Induction as the Basis for Program Verification", IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 244-252, December 1976.

[ROBI77] Robinson, L. and Levitt, K.N. "Proof Techniques for Hierarchically Structured Programs", in "Current Trends in Programming Methodology", Vol. II, Yeh, R.T. (editor), Prentice-Hall, Englewood Cliffs, New Jersey, 1977. (also in CACM, vol. 20, no. 4, pp. 271-283, April 1977).

[ROBI79] Robinson, L., Silverberg, B.A. and Levitt, K.N. "The HDM Handbooks", vol. 1-3, Computer Science Lab, SRI International, Menlo Park, California, June 1979.

[ROSS77a] Ross, D.T. and Schoman, K.E. "Structured Analysis for Requirements Definition", IEEE Transactions on Software Engineering, vol. SE-3, no. 1, pp. 6-15, January 1977.

[ROSS77b] Ross, D.T. "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Transactions on Software Engineering, vol. SE-3, no. 1, pp. 16-34, January 1977.

[SILB81] Silberschatz, A. "On the Synchronization Mechanism of the Ada Language", SIGPLAN Notices, vol. 16, no. 2, pp. 96-103, February 1981.

[SU77] S.Y.H. Su, "Computer Hardware Description Languages and Their Applications: an Introduction and Prognosis", Computer, Vol. 10, No. 6, pp. 10-13, June 1977.

[WEGN80] Wegner, P. "Programming with Ada: An Introduction by Means of Graduated Examples", Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[WIRT73] Wirth, N. "Systematic Programming: An Introduction", Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[YEH77] Yeh, R.T. "Verification of Programs by Predicate Transformation", in "Current Trends in Programming Methodology", Vol. II, Yeh, R.T. (editor), Prentice-Hall, Englewood Clifs, New Jersey, 1977.

[YOUN80] Young, W.D. and Good, D.I. "Steelman and the Verifiability of (Preliminary) Ada", SIGPLAN Notices, vol. 16, no. 12, pp. 113-119, December 1980.

[YOUR75] Yourdon, E. and Constantine, L.L. "Structured Design", Yourdon Press, New York, 1975.

intellitech
Intellitech Canada Ltd.
352 MacLaren Street,
Ottawa, Ontario.
K2P 0M6
(613)235-5126