intellitech

The Intelligent Use of Technology

(2)

SPACECRAFT MULTIPROCESSOR DESIGN METHODOLOGY: SPECIFICATION AND HARDWARE/SOFTWARE PARTITIONING

INT-83-52

SPACECRAFT MULTIPROCESSOR DESIGN METHODOLOGY: SPECIFICATION AND HARDWARE/SOFTWARE PARTITIONING

Computer Aided Engineering Tools for

Spacecraft Multiprocessor Systems

(Contract # OER 82-05067)

AUTHOR: Dr. C. Laferriere

APPROVED BY: Dr. S.A. Mahmoud

INTELLITECH CANADA LIMITED

352 MacLaren Street
Ottawa, Ontario
K2P 0M6

JULY 1983

Government of Canada / Gouvernement du Canada

Department of Communications

DOC CONTRACTOR REPORT DOC-CR-SP -83-060

DEPARTMENT OF COMMUNICATIONS - OTTAWA - CANADA

SPACE PROGRAM

TITLE: SPACECRAFT MULTIPROCESSOR DESIGN METHODOLOGY: SPECIFICATION AND HARDWARE/SOFTWARE PARTITIONING

AUTHOR(S): Dr. C. Laferriere

ISSUED BY CONTRACTOR AS REPORT NO: INT-83-52

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

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: 36001-2-0560
SER NO. OER82-05067

DOC SCIENTIFIC AUTHORITY: R. A. Millar

CLASSIFICATION: Unclassified

DATE: July 1983

FOREWORD

This document is the final report on tasks 3.4, 3.5 and 3.6 of the "Computer Aided Engineering Tools for Spacecraft Multiprocessor Systems" contract. As such, it constitutes deliverables 4.3 and 4.4. This work was done for the Federal Department of Communications, Communications Research Centre, Shirley Bay, Ottawa, Ontario.

TABLE OF CONTENTS

LIST OF FIGURES

1.0 INTRODUCTION

In recent years, there has been a considerable improvement in the performance of computer systems. Faster circuits and devices, miniaturization and better design techniques have all contributed to an increase in computing power and to an overall reduction in size and power consumption. This, in turn, has made possible the use of computers in applications which had hitherto been too complex. Such an application is the control of a spacecraft by an on-board computer system.

A spacecraft computer system normally performs several functions such as : processing of data obtained from various sensors, house keeping functions (e.g. monitoring temperature, power supply output, etc.), telemetry and support of remote re-programming of the computer system. Previous reports [LAFE82], [OUIM82] investigated the use of computers aboard spacecraft, as opposed to using ground based computers. The results of this investigation show that an increasing number of functions have been (or are being) taken over by on-board computers [CARN83], [THEJ83]. Of course, the computing power at the spacecraft designer's disposal is limited, especially in the uniprocessor case. It is now obvious that multiprocessor systems are a solution to the need for more processing power. Unfortunately, however, the design of special purpose, let alone general purpose, multiprocessor systems is still not well understood. Several difficulties are encountered in the design of multiprocessors:

1. Their complexity can be quite high. Consider, for example the various types of processors that are available, the numerous interconnection schemes that can be used, etc. and soon, the complexity of the design makes itself felt.

2. The software that will run on multiprocessor systems has to be designed carefully if full advantage of the multiprocessor hardware is to be taken.

3. Since high reliability is a necessity, both the hardware and the software have to incorporate some reliability mechanisms. For the hardware, this requirement translates into replication techniques and radiation hardening. On the software side, special recovery algorithms and fault detection routines have to be designed and coded.

4. Design contraints are quite stringent especially because of the space qualification requirements imposed on the hardware components. Software quality has not been emphasized as one would have expected although formal verification of certain modules is likely to be mandatory on some future spacecraft.

In view of these difficulties, the design of multiprocessor systems for spacecraft applications was studied to derive some guidelines to help the designers of such systems. The findings of that study and other related work are reported here.

The most important factor determining the overall success of the design operation is whether or not a suitable design methodology, complemented with appropriate computer aided engineering tools, exist. Such a methodology was developed [LAFE82] and is based upon a top down approach. With this methodology a designer would express the design in a set of functional sepcifications capturing the essence of the spacecraft operations. This functionality would be refined and expressed as a network of data flow elements. Physical or implementation constraints would also be listed carefully; timing constraints, such as maximum permissible time to perform a given operation would be applied to the data flow net. All these, the data flow net, timing information and constaints are inputs to the next stage: Hardware/Software Partitioning.

Hardware/Software partitioning requires as a stating point a series of guidelines, preferences or arbitrary choices from the designer. Based upon those choices, the number of processors can be determined as a function of the computational load of the software which has to be executed. Additionally, that stage may determine that certain primitive functions cannot be implementd by any combination of hardware/software and that the only other alternative is the use of dedicated hardware.

The next stage of the methodology supports two concurrent but complementary activities. On one hand, special CAE tools are employed to define the necessary hardware structures. This work is done in software only, thereby relieving the designer of the task of breadboarding and testing. In fact, powerful software packages (e.g. N.mPc, N.2) are used to simulate the functionality of the hardware, completely in software. On the other hand, the software is developed using a multi-tasking language such as Ada and is based to a great extent upon the system functional description obtained previously. In this fashion, the development of the software should flow naturally from the high level downwards.

When both the software and hardware are ready, it is possible to integrate them and to test them fully. This is done using N.mPc/N.2 which support the integration of software (in load modules) with target machines; also included are dedicated hardware devices. Using this flexible testbed, final acceptance tests can be carried out. Further modifications, if needed, can be incorporated into the design.

The use of computer aided engineering (CAE) tools with this methodology is quite important. Already mentioned in this connection were N.mPc and N.2. Other tools will likely have to be developed and

integrated in the methodology. Of special concern are the areas of functional specification and of hardware/software partitioning. These areas will be investigated in more details in subsequent chapters. This will also be supplemented by examples from actual spacecraft applications such as Attitude and Orbit Control Systems (AOSC). the examples themeselves are not extremely involved but serve to illustrate important points. The design of spacecraft on board software is still largely a new field and such application software is proprietory. Developing it for this study would have been impossible due to its size and complexity.

The present document being a report on the design methodology for spacecraft multiprocessor systems, its structure follows closely the introduction in its treatment of various topics. Following the introduction (Chapter 1), Chapter 2 deals with the design methodology in detail. Concepts and techniques related to the design of multiprocessor systems are explained. The methodology encompasses several levels: high level specification, hardware/software partitioning, implemenation (hardware, software and their integration). These are dealt with in Chapters 3,4,5 respectively. Finally conclusions can be found in Chapter 6.

2.0 MULTIPROCESSOR DESIGN METHODOLOGY

As explained briefly in the introduction, the design methodology attempts to translate functions into implementation. In doing so, it assumes that functions can be described in such a way as to allow this high level representation to become, usually, a combination of software and hardware. The software, in this case, would represent the required functionality while the hardware would support the execution of the software.

The system's functionality can be characterized by a structure and a behaviour. A behaviour is the series of actions carried out by a given module. These actions are in turn supported by a structure of lower level elements (or modules). This notion is very important to understand the methodology and the way it works. Figure 2.1 sheds some light on the structure - behaviours concepts. In that figure, a given set of functions (behaviour) are implemented by a module which is itself made out of lower level modules (perhaps primitive operations such as add, substract, in software and/or, or gates in hardware). Those lower level elements have to be connected in a very particular fashion if the behaviour of their combination is to be desired one. This interconnection pattern is the structure component of the module.

It is also important to realize that the structure behaviour concepts apply equally to hardware and software. Referring to Figure 2.1, the Input(s) can be electrical signals or numerical data. The functions inside the box could be, as mentioned before, gates, arithmetic operations, or abstract operations still at high level. This duality of structure-behaviour in both hardware and software is what makes possible the top down design methodology.

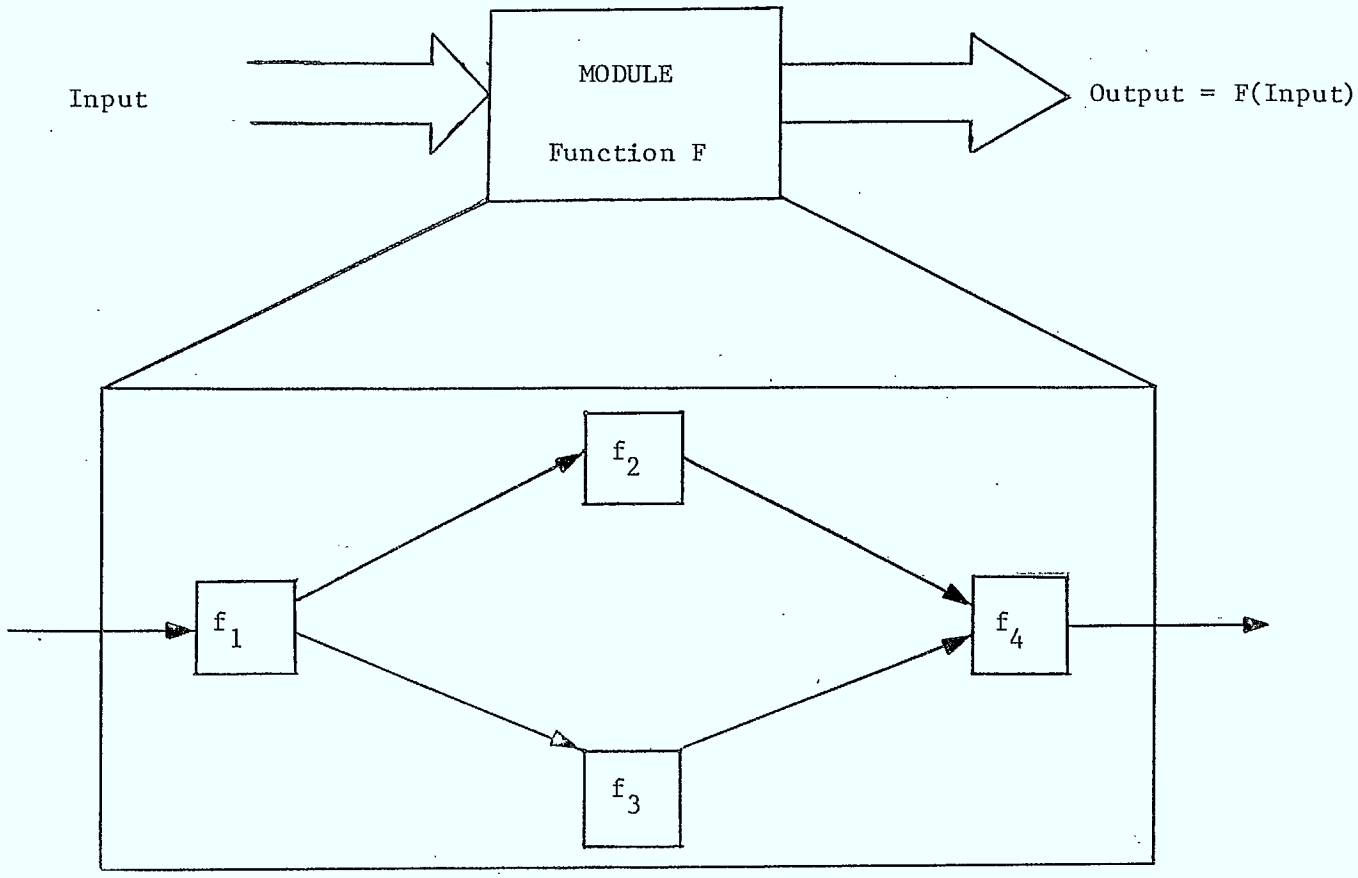


FIGURE 2.1 Structure/Behaviour Example

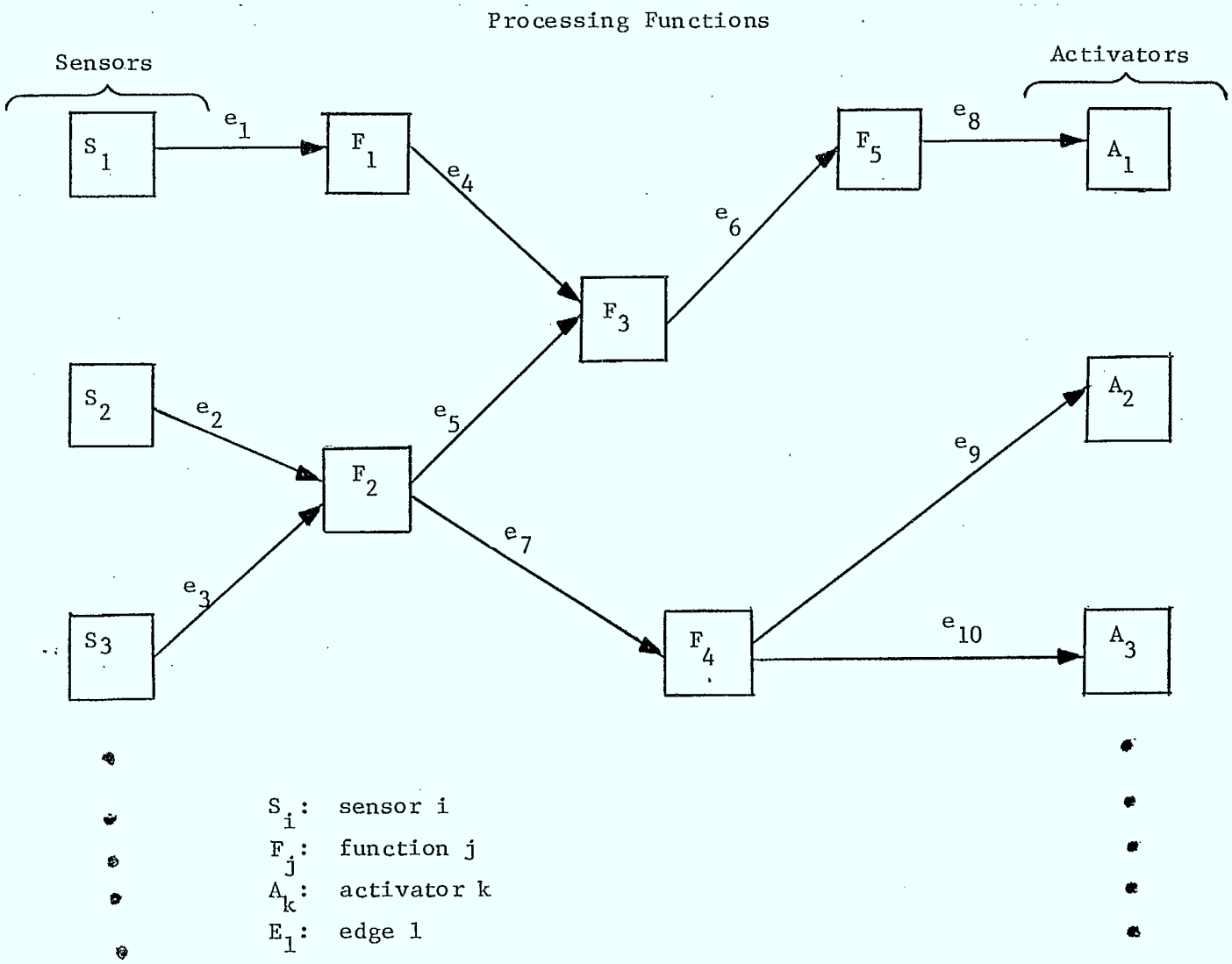
The next item of importance in the elaboration of a design methodology is the nature of the end product. In this case, the methodology aims at producing spacecraft onboard multiprocessor systems for onboard processing of collected data, housekeeping chores, etc. By the very nature of the tasks involved, it becomes apparent that the functions of a spacecraft computer system are modelled by an Input/Output representation. This means that data of various nature flow into the computer system, are processed, and are fed to the output data sinks. More precisely, the input data originate from various sensors on request from a processor or simply at given intervals. Those values can be collected by interrupting the program running at the time the value became available (asynchronous) or by sampling the device's ports at regular intervals (synchronous). Another characteristic of the input data is its flow over time which can be regarded as a lower level concern at this point in the methodology. The next step is the processing of the data items obtained from the sensors functions are called into play to process the input data, and although During that step, the functional level does not convey any idea of time, other levels do. Consequently, the input data flow rate becomes important since it determines, to a great extent, how fast the functions will have to be executed. This information is central to the hardware software partitioning stage. Finally, the output of the processing step is fed to various activators, such as thrusters, etc. It should be realized that those actuators have a maximum permissible input rate; data cannot be entered faster than the device can accept.

Figure 2.2 shows a data flow representation of a simple input/output system. The preceding description applies to the input sensors, output activators and processing functions. Obtaining this

representation will be the first basic step of the methodology.

The data flow model is very well suited to describing the functions of on board computer systems. Figure 2.3 depicts an Attitude Control Systems (ACS) which will be used as an example throughout this report [DUSI83]. The complete loop through the spacecraft dynamics in space is also shown in Figure 2.3. It is a reminder that the spacecraft control system is working in a closed loop system since the changes effected by the computer system are then refelected in the input data from the sensors. Incidentally, that the ACS was shown in Figure 2.3 was chosen arbitrarily; an orbit control system (OCS) or a combined attitude and orbit control system (AOCS) could have equally been chosen. The ACS example was judged sufficient to illustrate the important points.

The top down design methodology for multiprocessor systems is shown in Figure 2.4. The top three boxes cover the functional description of the multiprocessor system. As outlined before, the purpose of this stage (shown here as three separate sub stages), is to obtain a representation of the system like the one shown in Figure 2.3. That representation can be further refined by a more complete data flow decomposition [YOUR75] and/or a hierarchical decomposition [DIJK76]. The basic representation of Figure 2.3, or something more refined, serves as the basic input to a "Data Flow Analysis" stage. Data flow analysis is concerned with the flow of data into and out of the system. In other words, this stage tries to determine how fast should certain functions perform their task so that the computational load associated with them can be assessed.



NOTE: An edge e_l is characterized by a peak flow rate PFR(e_l) and an average flow rate AFR (e_l). Similarly for sensors, there are a Peak data rate PDR (S_i) and average data rate ADR (S_i). Activators have a maximum permissible rate MPR (A_k).

FIGURE 2.2 Data Flow Representation

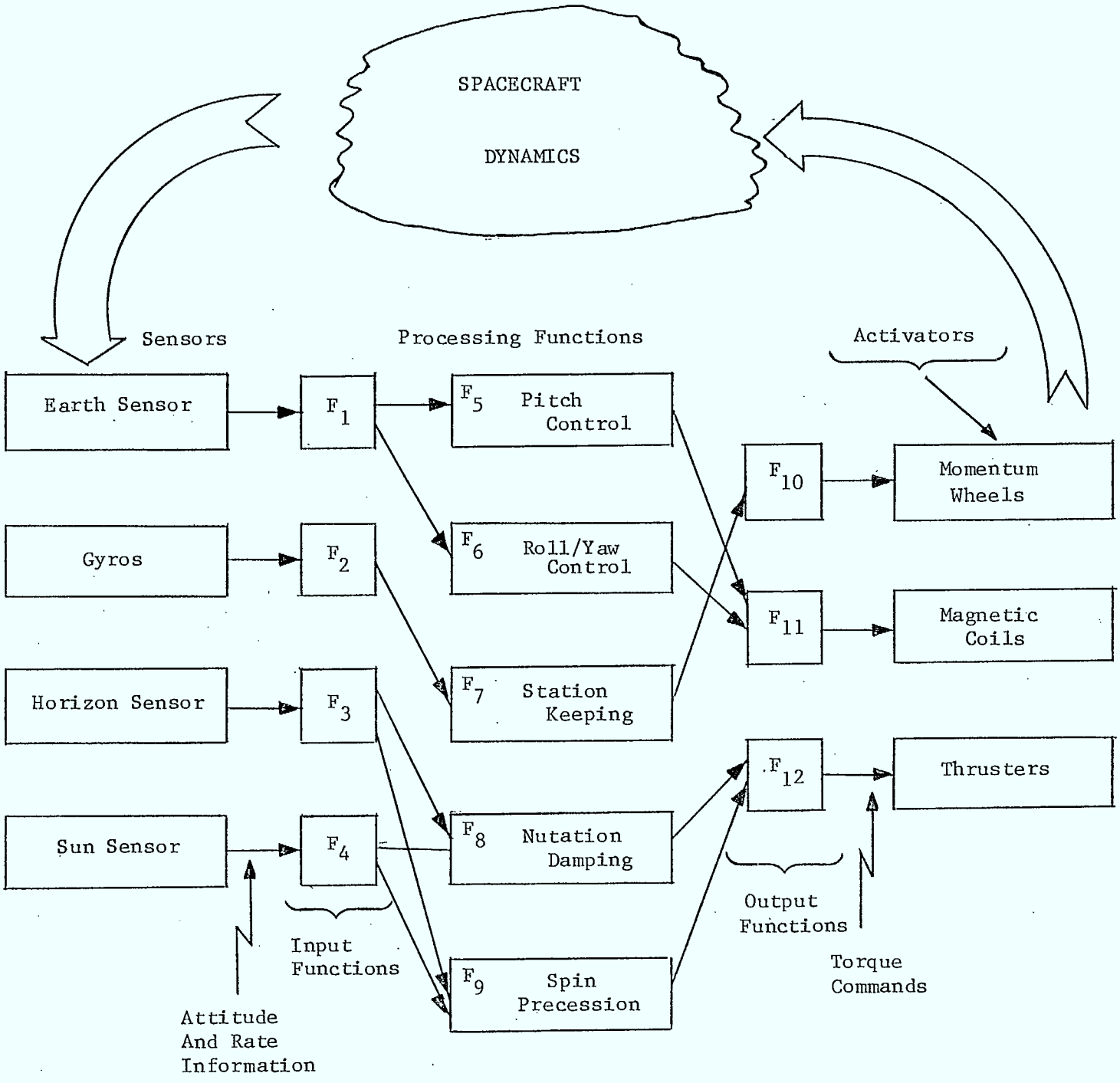


FIGURE 2.3: Attitude Control System of a Spacecraft
(NOTE: For Example purposes only)

Data flow analysis is elaborated on later but it should be emphasized that:

1. Its purpose is to establish the computational load of functions;

2. It is made possible by the input/output nature of a spacecraft control system and the fact that all those functions are periodic.

The computational load estimate tries to be a measure of the computations that have to be done per unit of time. The unit of time can be arbitrarily chosen but should be smaller than the smallest service cycle in the system. The computational load is not an extremely accurate figure, but rather a guideline. The reason for the lack of accuracy lies with the functional description and the difficulty in determining the algorithms to be used. Obviously, there is not question of software implementation at this level but nevertheless, several algorithms can potentially be used for moderately complex functions thus introducing variations in computational load.

There are two methods of obtaining the computational load of the functions of the system. The first one, termed "Data Flow Analysis", relies on an analysis of the data flow paths and the data flow carried along those paths. The data flow is characterized by an average and a peak flow rates from the sensors and by a maximum permissible rate into the actuators. Based on these rates, it is possible to arrive at a set of upper bounds for the time taken by each function to process the data. The time alloted to each function (during a given cycle) together with the complexity of the tasks to be performed constitute the measure of computational load for that function.

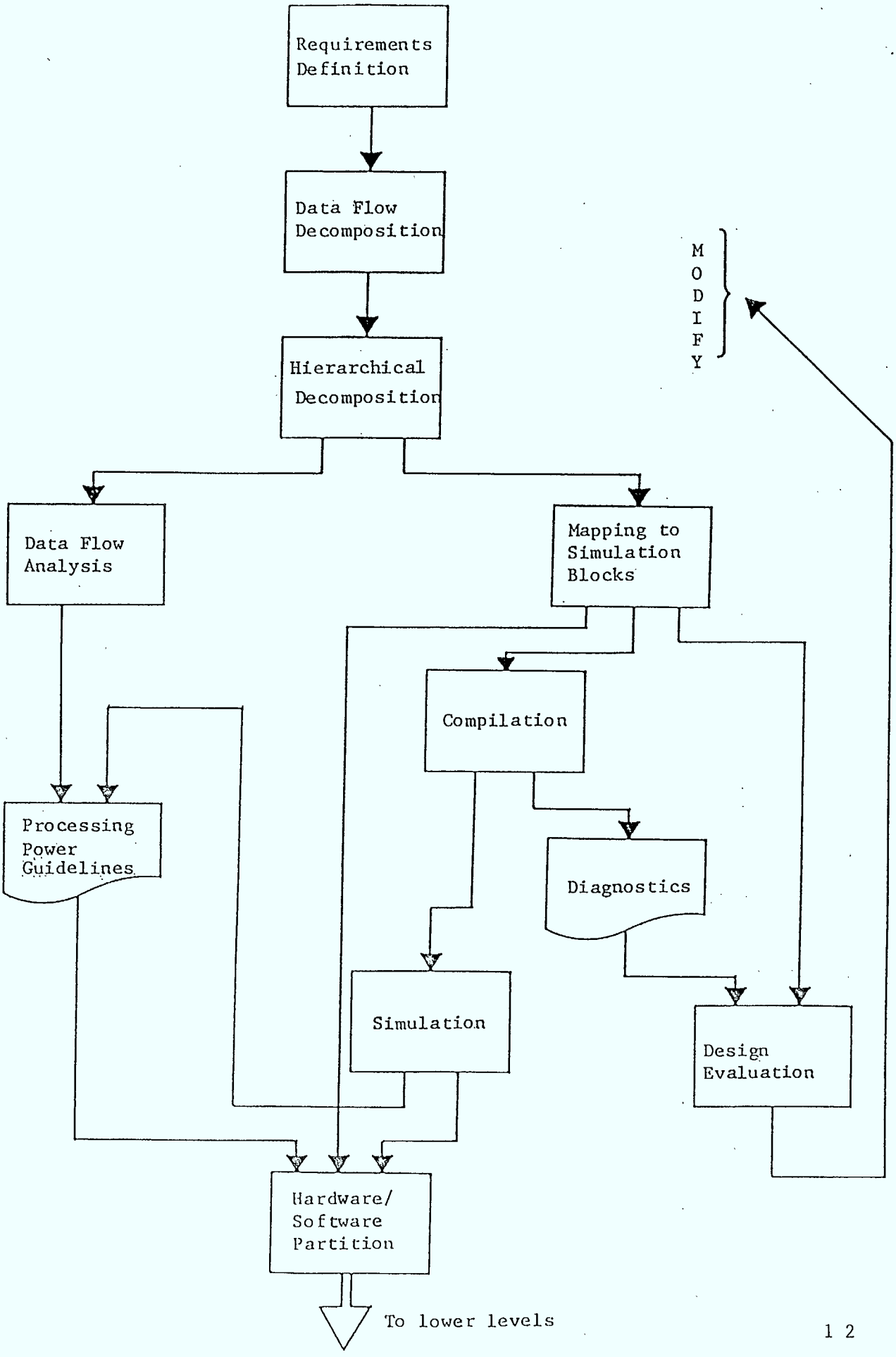


FIGURE 2.4 Design Methodology

The second method of obtaining the computational load of a function is based on a functional simulation. That route is shown on the right hand side of Figure 2.4. This method requires that the functional description of the system be mapped into a set of Ada constructs, hereafter referred to as simulation blocks. The system representation with simulation blocks can be compiled and executed. The benefits are twofold:

1. An inconsistencies can be revealed, thanks to the diagnostics from the compiler.

2. Upper bounds for the time allocated to each function to perform its tasks can be ascertained. This can be done, especially in Ada, by not coding the function's algorithms and by replacing it with a "delay" statement. Various iterations may prove necessary before the right set of delay durations is found.

When computational load indications have been obtained for all the functions of the system description, the next step, hardware-software partition, can be undertaken. The purpose of this step is to take as inputs the computational guidelines and other assorted constraints, and map the system's functional description into possible implementations. There exists a wide range of acceptable implementations at this stage and it is necessary to narrow down the choices further. This is accomplished by tightening the constraints and especially by introducing the concept of reliability. As pointed out before, one of the most important characteristics of a spacecraft onboard processing system is its reliability. The reliability of a system depends on many factors such as architectural configuration, fault detection facilities, software recovery algorithms, etc. Some of those factors have yet to be determined, e.g. software algorithms, while others are in relatively final form, e.g. architecture, components selection (major chips). By critically examining the proposed implementations with those criteria in

mind, it should be possible to eliminate unsuitable alternatives and thus concentrate on a few promising ones. If it turns out that none of the proposed implementations is suitable, a loop back to the higher levels of the methodology is necessary. Usually, what is entailed is a further refining of the functional decomposition allowing the use of more processors or special purpose devices. This will result in more spare processing power for recovery algorithms for example. Additionally, direct choices can be made by the designer, in an interactive fashion, in order to guide the methodology towards a suitable implementation.

Once a suitable configuration is selected, presumably from the set of potentially suitable configurations, three major tasks remain: hardware implementation, software implementation, and integration and testing. These steps are shown in Figure 2.5.

The output of the hardware/software partitioning stage is made out of two parts:

1. A set of hardware general purpose processors and optionally of special purpose processors. Various constraints are also given, such as minimum execution speed, type(s) of bus interconnection, I/O transfer rate(s) etc.

2. A complete functional description of the tasks to be performed by the system. This description differs from the functional description used by the hardware/software partitioning stage in that it does not include functions that will be implemented completely in hardwre on the account of faster execution speed.

The hardware development path uses the information listed above and endevours to create a suitable hardware structure to support the software. In the first stage, a design is arrived at in an iterative fashion, although that is not explicitely shown in Figure 2.5. Hardware implementation follows to create a system at the VLSI building block

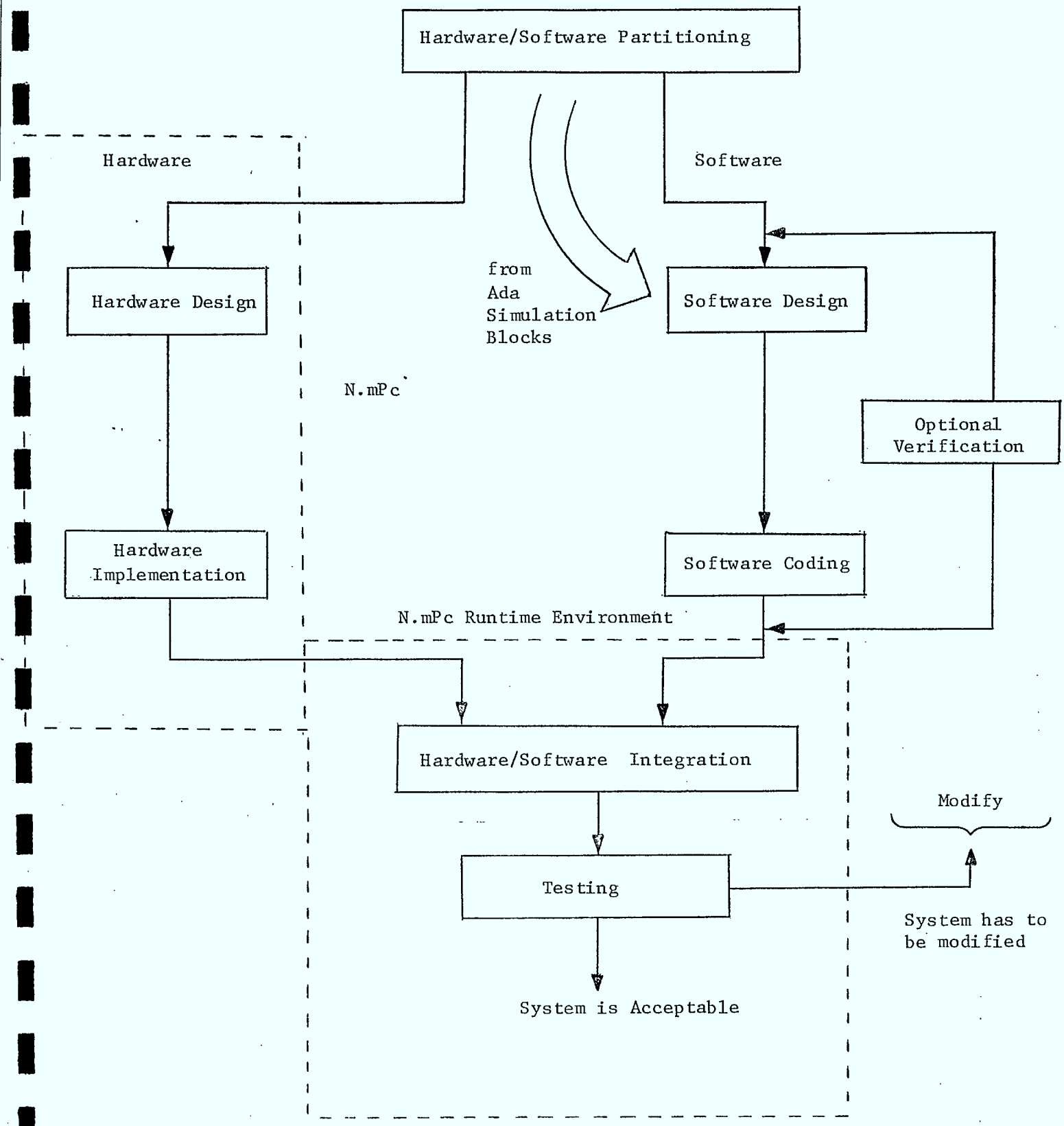


FIGURE 2.5: Lower Levels of the Design Methodology

level. In other words, the system thus created makes use of either Processor/Memory/Bus structures or of Microprocessor/Memory/Bus Tranceivers/Interrupt Controller, etc. chips, all connected as in a real physical system. There is no hardware building as the N.mPc system is used. The use of N.mPc in such an environment has been the subject of an investigation within the framework of the current work. Relevant information can be found in [LAFE83a], [LAFE83b].

On the software side, the functional specification is used extensively as it would usually have been translated into Ada simulation blocks or other suitable constructs. Using the simulation blocks, it is possible to decompose the functions into a hierarchy of subprograms and thus implement the functionality of the system. At this stage, verification assertion can be used if formal software verification is desired.

Having completed the hardware and software design and implementation, the next step in the methodology is the integration of hardware and software and final testing. N.mPc run-time environment is used to accomplish the integration and the testing of the system. it is interesting to note that not a single piece of hardware had to be built, but, if needed, the actual implementation of it would be relatively straightforward.

The final testing stage is concerned with the performance of the target system. This can be ascertained using the N.mPc runtime environment as long as proper allowances are made for the difference between simulation time and physical time; simulation time is a linear expansion of physical time and N.mPc handles various timing cycles in a consistent fashion. The final test will doubtless reveal inadequacies in the design of the target system. Those in adequacies can be

corrected by looping back to an earlier stage, making the correction and re-doing the part of the design/implementation work that is affected by the change. How far back to loop is a difficult question to resolve; going too far back into the design work entails a considerable amount of changes to take care of. However, if the uncovered inadequacies are precisely indentified, it should be possible to locate where that particular design decision was made and effect corrections with a minimum of overhead.

This concludes the description of the design methodology for multiprocessor systems for spacecraft. The next three chapters will in turn take a look at special aspects of the methodology such as high level specification, hardware/software partioning and hardware/software implementation.

3.0 HIGH LEVEL SPECIFICATION TOOLS AND TECHNIQUES

This section addresses the difficulties arising from the derivation of high level specifications of the system. From the previous section on the design methodology, it is possible to identify two requirements which any specification tool must satisfy if it is to be of any use at all in the current context. These two requirements are:

1. The ability to capture the input/output nature of the system. In other words, the ability to describe a system similar to the block diagram characterization of Figure 3.1 (a).

2. The ability to take a description of an input/output system and expand it into a more accurate specification using methods that highlight the hierarchy of actions or the timing sequences of those actions. Such methods usually produce hierarchical decomposition, yielding results similar to those of Figure 3.1 (b).

For the purpose of this work, three such methods will be discussed:

1. Graphical methods;

2. Algebraic description;

3. Programming language description.

Before studying those methods further, it is important to realize the importance of this step as the obtaining of high level specification is of crucial importance to the rest of the methodology work. Furthermore, an important factor in the choice of the method to be used is how well will this method and its results integrate with the subsequent steps of the methodology. It may happen, therefore, that ease of interfacing of one method makes it more suitable than another one with greater power of specification but of difficult handling.

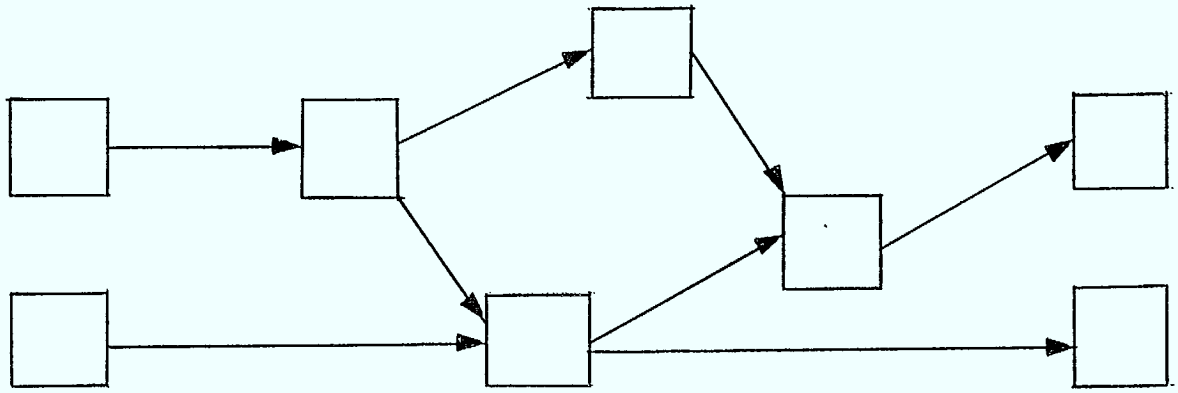


FIGURE 3.1 (a) Input/Output System Characterization

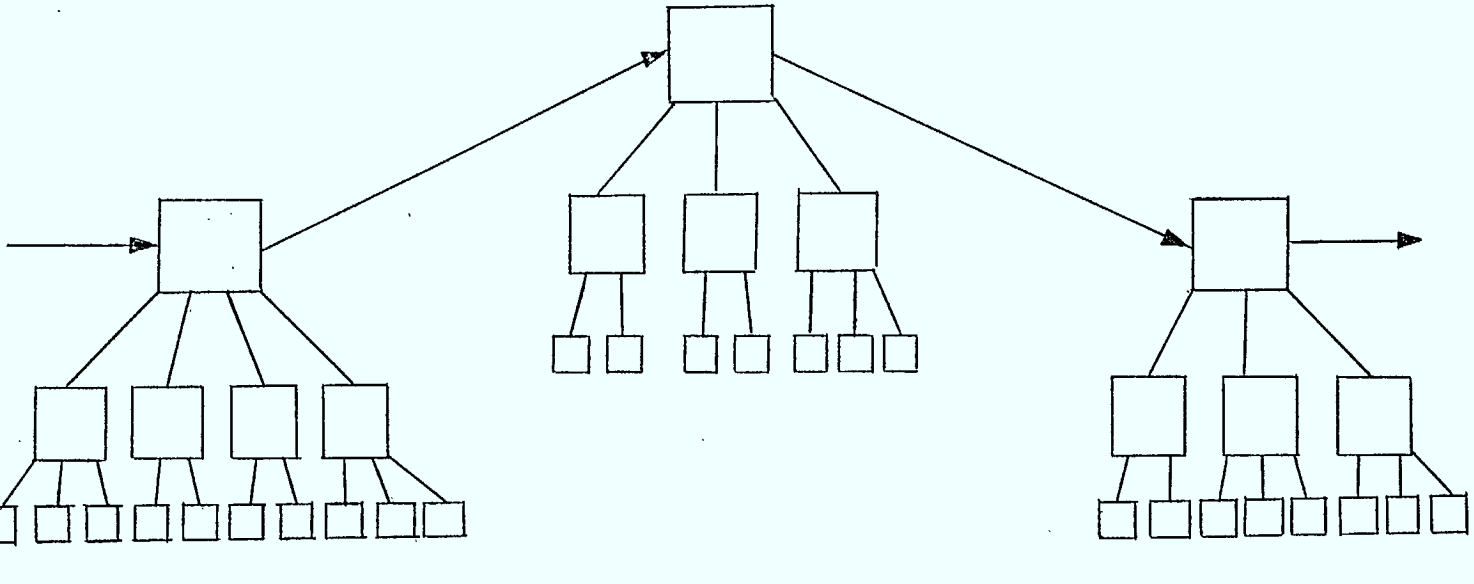


FIGURE 3.1 (b) Hierarchical System Characterization

3.1 GRAPHICAL METHODS

Dataflow diagrams and hierarchical diagrams as shown in Figures 3.1. (a) and 3.1 (b) are good examples of graphical methods. Those methods are the most natural at a high level and are usually quite good as a first attempt at system description. However, they suffer greatly from:

1. Lack of formalism. In fact, it is difficult to spot errors and inconsistencies using such tools.

2. Difficult interfacing. By their visual nature, graphical representations are quite good at enhancing human communications but are not suited to human/machine communications and they do not lend themselves well to automated processing.

Consequently, it is advisable that, in using the methodology, preliminary specifications be obtained by graphical means and transformed into a more powerful form later.

3.2 ALGEBRAIC SPECIFICATION

Algebraic specification methods have been used in a few systems already, albeit in a somewhat restricted fashion. One notable exception is the SIFT fault tolerant flight control system which was described entirely using algebraic methods.

The SIFT system [MELL82] was also partially verified by means of input and output assertions coupled with an automated verification condition generator. The verification aspect which prompted the need for algebraic specification is not of concern here, although future work should no doubt investigate hardware verification.

The algebraic specification method which will be described shortly is dependent upon the notation of levels, or hierarchy as illustrated in Figure 3.2.

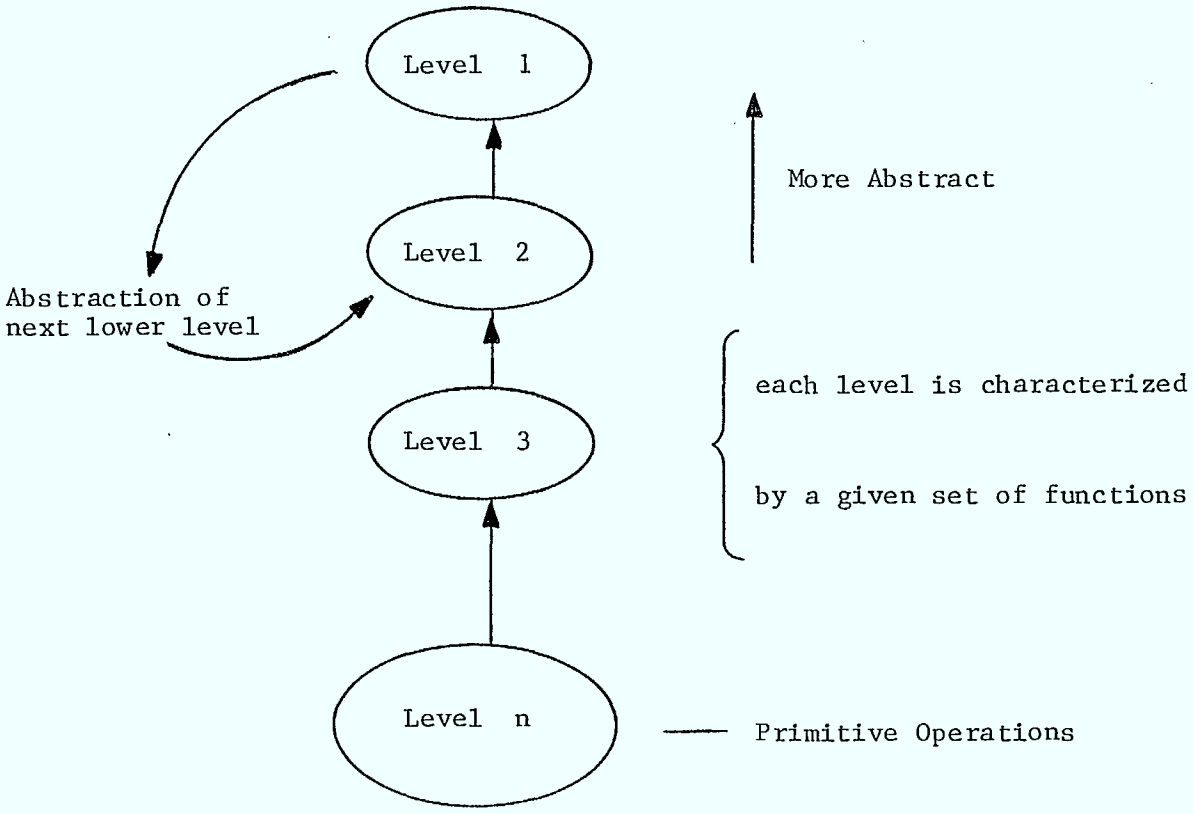


FIGURE 3.2 Multi Level Specification

As indicated in Figure 3.2, each level is composed of functions which are abstractions of more primitive functions existing at lower levels. Thus, a system can be described using high level functions and then, in turn, each of those functions can be decomposed into more primitive ones. It is also possible to mix high level and lower level functions in a same description providing that proper interfaces are included.

Figure 3.3 describes a very simple example to illustrate the concept of algebraic specification. It shows three tasks, A, B, and C with task C processing the output of tasks A and B. Those two tasks had got their input from other tasks (or input devices) although that is not shown on the diagram. In the SIFT system description, periodicity of functions is handled by letting each function correspond to a task and each task being also characterized by a particular iteration. Therefore, a given task A, as in Figure 3.3, could perform the transformation:

function (A).

The result of such a function upon a set of input data, say x_1, x_2, x_3 would be

result (A, i) = apply (A, (x_1, x_2, x_3))

where i denotes the ith iteration and x_1, x_2, x_3 are input data to the ith iteration. Obviously, x_1, x_2, x_3 are values related to the ith iteration since they are of the form:

x_i = result (X_i, k_i), that is resulting from the kth iteration of task X

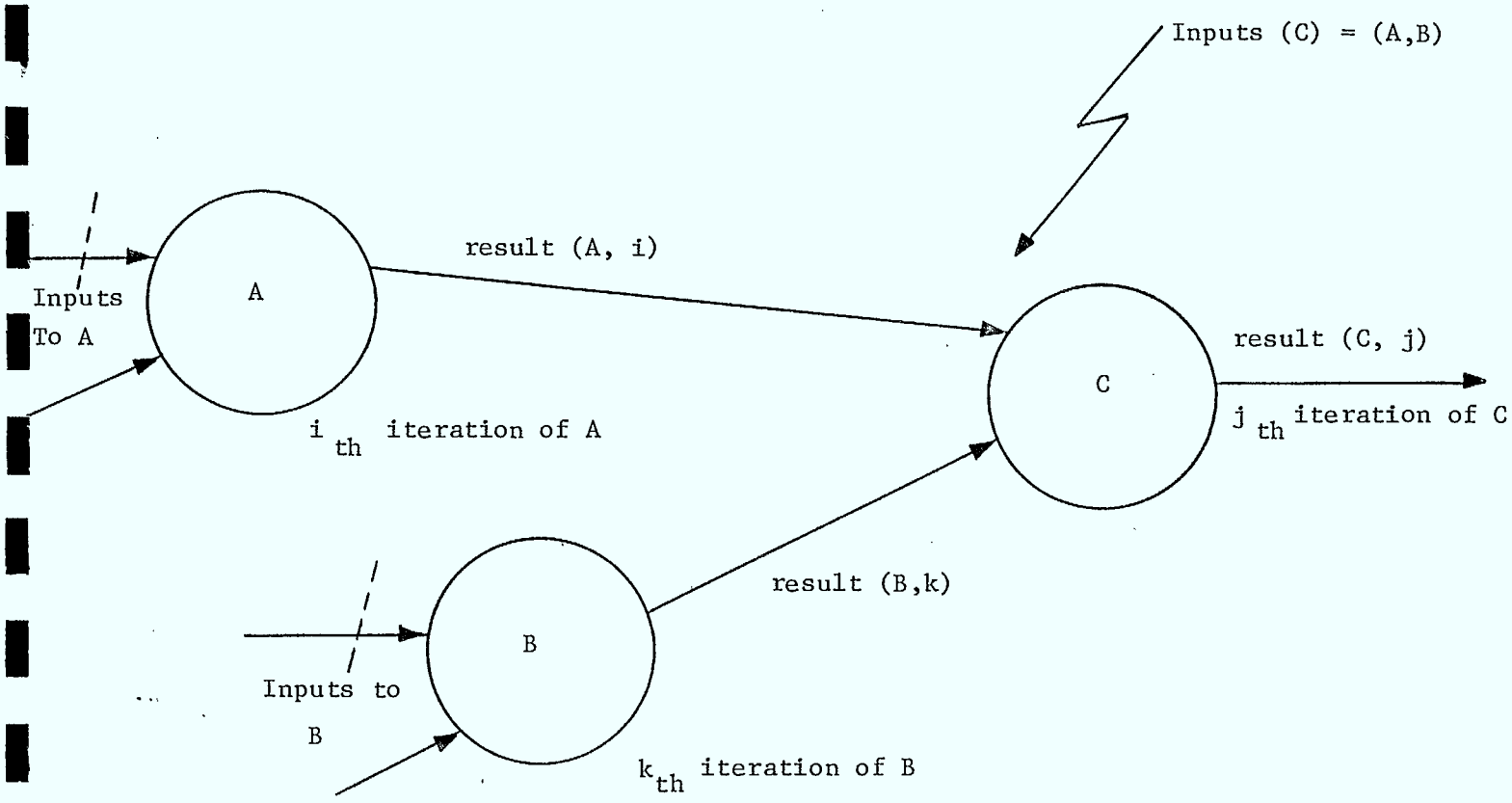


FIGURE 3.3 Example of Algebraic Specification

The resulting output produced by task C during its jth iteration can be expressed as follows:

result (C, j) = apply [C, (result(A,i), result(B,k))].

Here again, the input data is characterized by an origin, i.e. a task, and an iteration during which it was produced. The concept of iteration is therefore very useful to model the flow of data from sensors to activators in a spacecraft environment. It is interesting to note, however, that strictly speaking the notion of physical time is still absent.

In the example above, the function (C) still has to be defined in terms of how it transforms its inputs into suitable outputs. In the SIFT system, an important restriction is that a task can have many inputs but can only produce one output. This is so in order to facilitate the formal verification of the system. The definition of a function can be given in terms of how it transforms the data, or in other words, by specifying:

1) the input data set, (domain)

2) the output data set, (range)

3) the function itself, i.e. how it relates individual input data items to output data items.

It should be pointed out that the sets of input and output values can be decomposed further, as is expected in multi-level specification work. In this fashion, a more precise description of the system can be obtained. In going to lower levels, considerations will have to be given to important factors such as:

1. Timing; at lower levels, the concept of task iteration is reduced to the notion of timing.

2. Scheduling; since timing is considered and that tasks are periodic, proper thought has to be given to the scheduling of those tasks.

Absent from these factors is the actual shape and nature of the target hardware/software system. This will only be specified at much lower levels.

Another aspect of algebraic specification methods of the SIFT category is the way in which they handle reliability. In the SIFT system, high reliability is achieved through replication of tasks and voting on the output of each. Algebraic specification can handle this description task adequately if the concept of physical processor is introduced. Figure 3.4 depicts a part of the simple system of Figure 3.3 with replication of tasks.

In Figure 3.4, the output of each task is sent to voting mechanisms at all sites that should participate on the voting. Consequently, in the example of Figure 3.4, three instances of task A send their output to three voters (voting mechanisms), with a voter on each processor. The output of a voter is obviously the correct (at least perceived to be so) result of that function at that site. The notion of task iteration is of importance in the example of Figure 3.4, and more so than before since the voters have to await results from different processes on different processors.

As mentioned before, going to a lower level often entails the introduction of other functions. In fact, these are necessary to ensure the completeness of the description. Therefore, other task characteristics will then have to be described and in turn their descriptions will bring the system description to a lower level. In this fashion, a complete multilevel specification of the system can be obtained.

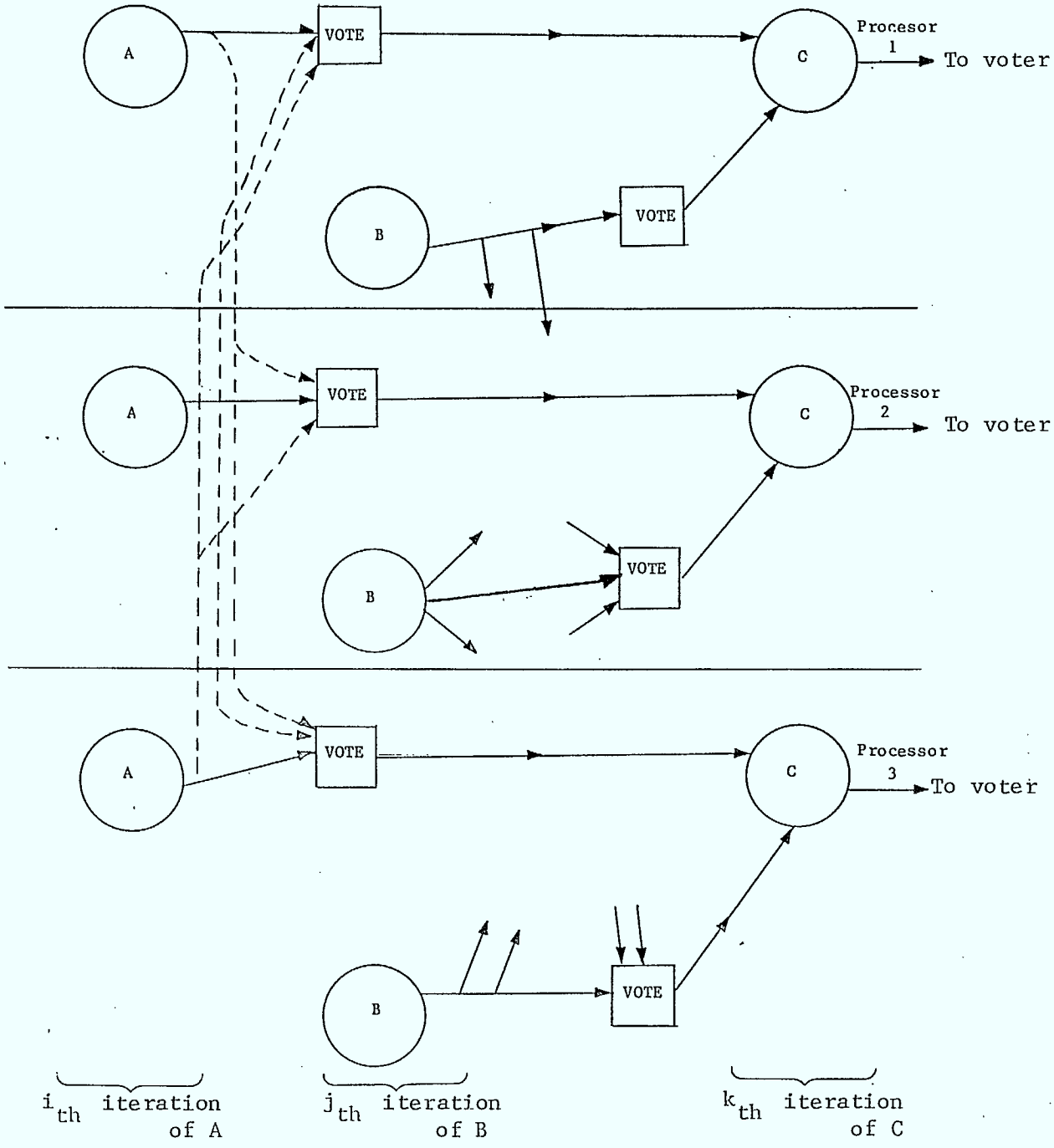


FIGURE 3.4 Reliability and the Algebraic Specification

3.3 PROGRAMMING LANGUAGE SPECIFICATION

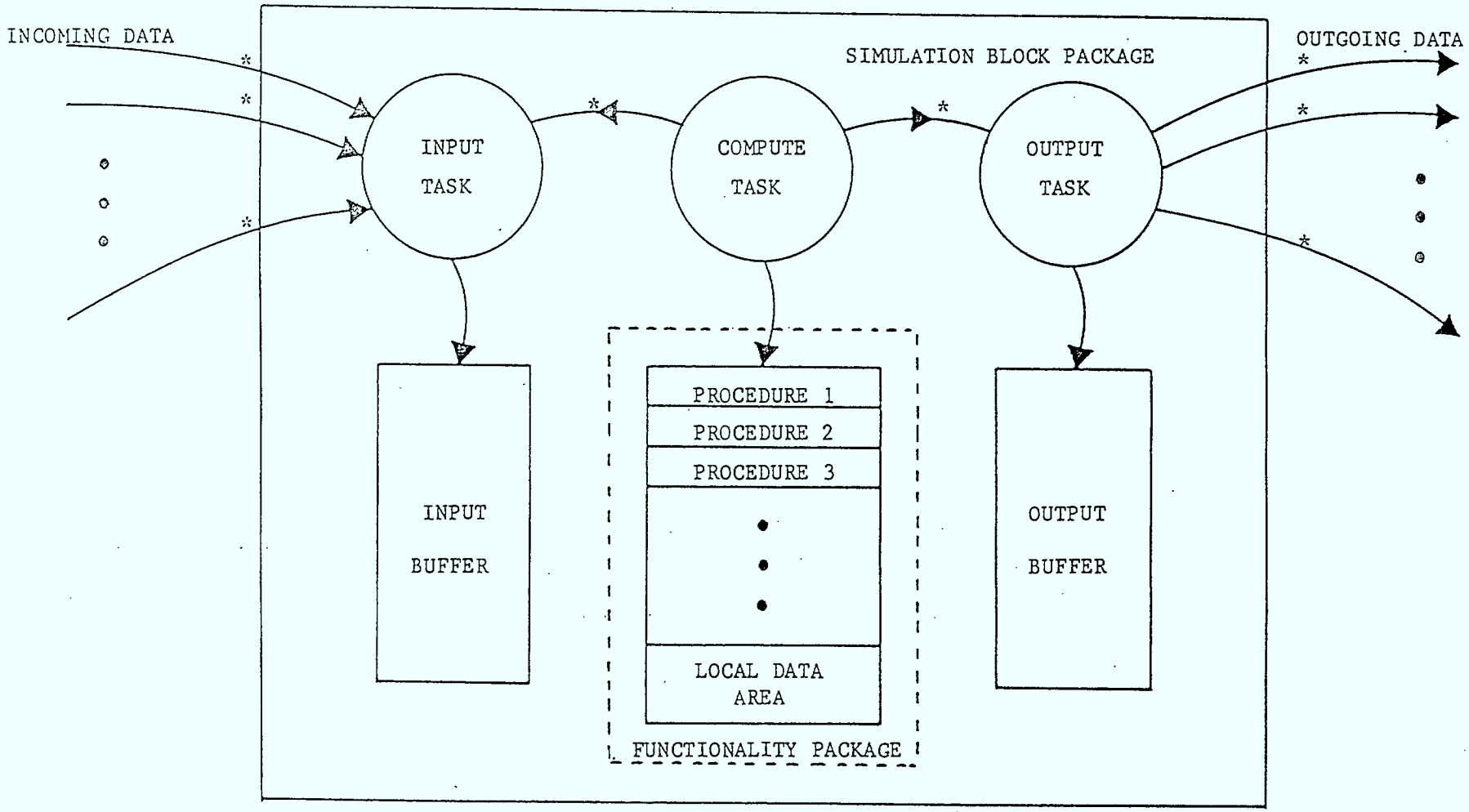
Programming Language specifications, as the name indicates, specify systems by using a high level programming language. It therefore requires a powerful language capable of supporting flexible data structure description and of handling concurrency. The fist requirement, flexible data structuring, is necessary to represent the various data present in the system, at the various levels of decomposition to be encountered. The second requirement, support of concurrency, is required by the need to represent functions processing their data seemingly at the same time, or at least, during the same period. [A period can be defined to be the length of time during two invocations of the slowest recurring task].

For the purpose of this section, and also for the methodology itself, Ada* was chosen as the programming language to be used in the specification task. Ada supports flexible concurrency, has powerful data structuring facility, and provides for separate compilation and static protected variables through packages. With Ada , two types of mechanisms can be used to represent the high level description of a system:

1. Package with "Input"/"Output"/"Compute" tasks.

   This mechanism was outlined in an earlier report [LAFE82] and refined in [LAFE83c]. Figure 3.5 depicts a simulation block which uses three tasks, "Input", "Output" and "Compute", and is contained in a package.

(*) Ada is a trademark of the U.S. Department of Defence



*: ADA RENDEZVOUS

FIGURE 3.5 Simulation Block

The "input" task is an independent Ada task whose function is to accept data from various other modules (sources or output tasks) and store it in an input buffer. The input task can be interrogated by the compute task as to the availability of the data.

The "Compute" task receives data upon request from the "Input" task. With the data in its local data area, the "Compute" task will perform the processing associated with the functionality of the module which is being specified. This functionality is implemented by the program of the compute task and by all the procedures it uses during its execution.

The results of the "Compute" task are passed onto the "Output" task for transmission outside the module. The data is first obtained from the "Compute" task, stored in an output buffer and sent to other modules (sinks or input tasks).

As can be seen from the diagram of Figure 3.5, the simulation block can also accommodate source and sink modules. A sink module consists of a simulation block with just an "Input" task/input buffer arrangement and similarly for a source module. For simulation purposes, these input/output tasks can be modified so that their behaviour emulates that of the real devices.

As far as the compute task is concerned, it should be pointed out that the task draws upon the resources of another package, the "functionality" package. That package contains the procedures that implement the functionality of the module, and each of those procedures can be subjected to further hierarchical decomposition. The purpose of the "functionality" package is to allow easy manipulation of the procedures and data while, at the same time,

presenting a consistent interface to the compute task and other modules. It is interesting to note that simulation blocks can form a network thus representing data flow decomposition. At the same time, hierarchical decomposition can be carried out by breaking down the main taks into a set of procedures, which can, themselves, be decomposed in the same fashion. One of the main advantages of this particular technique is the fact that every component is enclosed in a common package and has a pre-determined role to play.

2. Network of Tasks.

There exists an alternative to simulation blocks as outlined previously. This alternative is to dispense with the package constructs, the input and output tasks and to represent the function to be described as a single task. This task and tasks describing other functions can be linked in the same fashion as the output task of a simulation block was linked to the input task of another block. This arrangement is shown in Figure 3.6. A few words of explanation are in order about that figure. The top half of it depicts an Ada task whose behaviour is that of the function to model. The task is called and calls other tasks, thus effecting data transfers. The diagram cannot show, however, the hierarchy of procedures within the task. For this purpose a hypothetical hierarchy of functions is shown in the lower half of Figure 3.6.

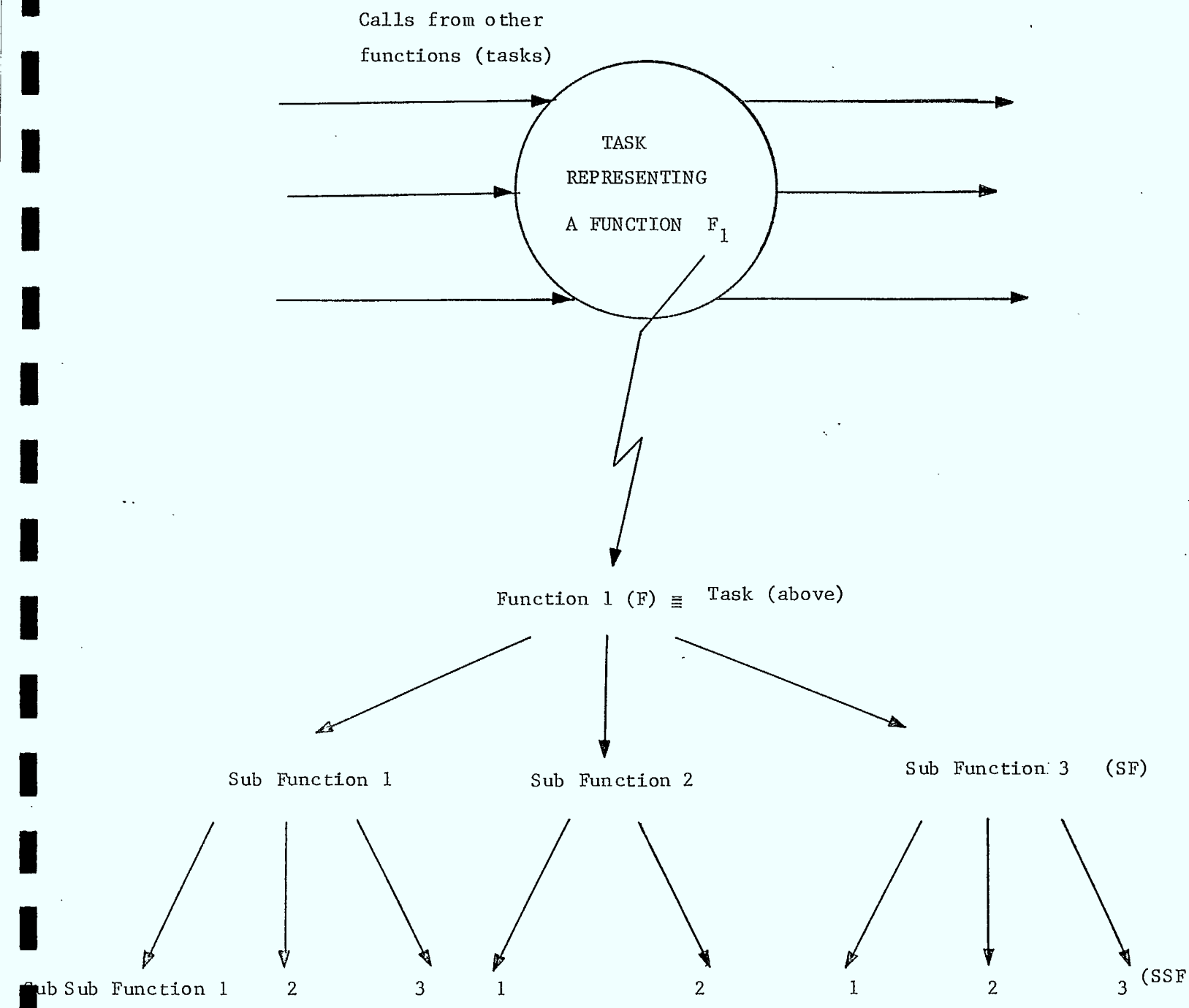


FIGURE 3.6 Example of Task Description

The representation of the lower half of Figure 3.6 in a programming language notation is to be found in Figure 3.8. Figure 3.7 introduces the necessary background material to Figure 3.8. That material is the Ada formal definition of a task. The first part of the task definition is concerned with entries and rendevous which will be used for linking tasks together. This topic will be dealt with next.

```
task declaration    :: =  task specification

task specification  :: =

     task [type]  identifier  [is

          {entry declaration}
          {representation specification}

     end  [identifier]];

task body           :: =

     task body identifier is

               [declarative part]

     begin

               sequence of statements

     [exception

               {exception handler}]

     end   [identifier];
```

FIGURE 3.7 Ada Task Specifications And Body
(From [DOD80], p. 9-1)

```
task Function1 is
            .
            .   -- to be determined later
            .
end  Function1;

task body Function1 is

      -- declarative part of function1

      procedure SubFunction1
           -- declarative part of SF1

              procedure  SubSubFunction1
                   -- declarative part of SSF1
                   -- local variables, etc.
              begin
                   .
                   .  -- statements of procedure SSF1
                   .
              end SubSubFunction1;

              procedure SubSubFunction2
                      -- declarative part of SSF2
                      -- local variables, etc.
              begin
                   .
                   .  -- statements of procedure SSF2
                   .
              end  SubSubFunction2;

              procedure  SubSubFunction3
                      -- declarative part of SSF3
                      -- local variables, etc.
              begin
                   .
                   .  -- statements of procedure SSF3
                   .
              end  SubSubFunction3;

              -- local variables of SubFunction1

              begin  -- this begin corresponds to procedure SubFunction1
                   .
                   .  -- statements making use of local
                   .  -- variables of SubFunction1
                   .  -- calling the sub programs
                   .  -- SubSubFunction 1,2 and 3
              end  SubFunction1;

      procedure  SubFunction2
              -- declarative part
              -- similar to procedure SubFunction1
```

(continued on next page).

```
                procedure  SubSubFunction1
                      -- note that SubSubFunction1 relates
                      -- to SubFunction2
                      -- declarative part of SSF1

                begin
                     .
                     . -- statements of procedure of SSF1
                     .
                end  SubSubFunction1;

                procedure  SubSubFunction2
                       -- declarative part of SSF2
                begin
                     .
                     . -- statements of procedure SSF2
                     .
                 end  SubSubFunction2;

          -- local variables of SubFunction2

          begin
               . -- statements making use of local variables
               . -- of SubFunction2 and calling the sub programs
               . -- SubSubFunction1 and 2
               .
          end  SubFunction2;

          procedure SubFunction3
                 -- declarative part
                 -- similar to procedure SubFunction1

                   procedure SubSubFunction1
                                .
                                .
                   procedure  SubSubFunction2
                                .
                                .
                   procedure  SubSubFunction3
                                .
                                .
          -- local variables for procedure SubFunction3

          begin
               . -- statements making use of local variables of
               . -- SubFunction3 and calling the sub programs
               . -- SubSubFunction1, 2 and 3
          end  SubFunction3;

     -- local variables of task Function1
     -- end of the declarative part of task Function1
```

(continued on next page)

```
begin  -- beginning of task Function1
     . -- statements making use of local variables
     . -- of task Function1 and calling the
     . -- subprograms SubFunction 1, 2 and 3.
     . -- use of entry calls and accepts is also in
     . -- here

end  Function1;
```

FIGURE 3.8 Specification of Function1

Figure 3.8 showed how the hierarchical decomposition of a function can be expressed using a high level language like Ada. In the previous section, high level description tools were described as needing the above mentioned hierarchical description capability and, also , the capability of describing network of functions. When the specification is being done using a high level language (i.e. Ada), the tool used is the Rendezvous. The Rendezvous is a synchronization primitive of the Ada language and is fully described in [DOD80] and [PYLE81]. It is used by independent tasks wishing to co-ordinate their actions or force a pre-determined execution sequence despite their parallel execution. The Rendezvous mechanism is illustrated in Figure 3.9 for the case of a task, "Device", emulating a source of data and of a task, "INPUT", receiving the data and processing it further or simply storing it.

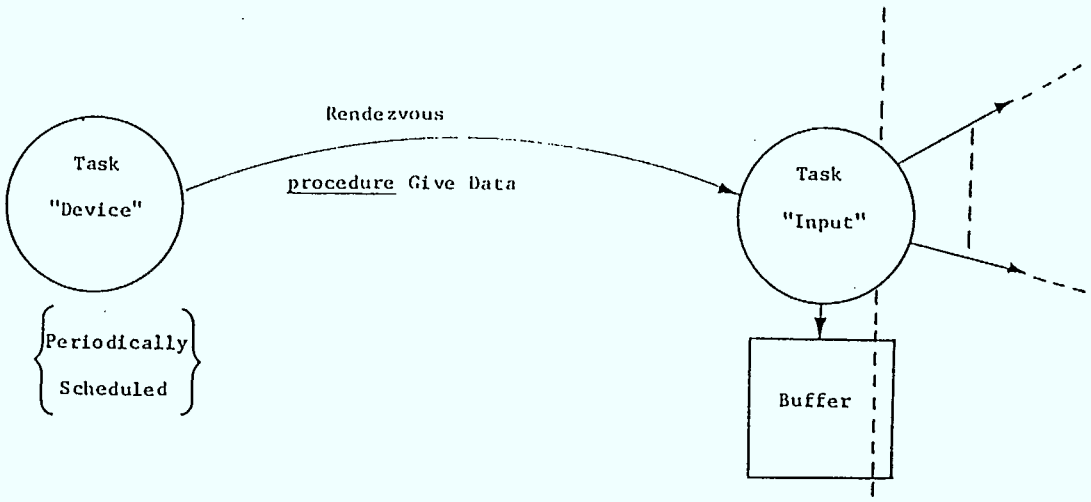
From the examination of Figure 3.9, several points become obvious:

1. The Rendezvous mechanism is undirectional. There is a task which is the active party and does the calling and another task which is passive and accept or delays acceptance of a call.

2. When bi-directional co-ordination is desired, two Rendezvous have to be set up. Care has to be exercised to prevent cycles and the ensuing possibility of deadlock.

3. The body of the procedure which will be executed as a result of a successful Rendezvous is in the called task and is enclosed by the do end delimiters.

4. Data can be exchanged during the Rendezvous, and the exchange is not restricted to only one direction. The parameter list with the Rendezvous procedure will determine the type of parameters and whether they are read-only , read-write, and write only.




```
task  Device is
        -- no incoming entry calls
end;

taskbody Device is
        -- declarative part
        -- e.g. procedure Generate
                          Data
begin -- of task Device
      .
      .
      .
loop
      .
      .
      .
      .
      x: = Generate/Data;
      Give Data (x); --potential wait
        .
        .
        .
end loop;
      .
      .
      .
end device;
```

```
task Input is
entry GiveData (X:in item);
  --any other entries if any.
end;

task body Input is
  -- declarative part
  -- local procedures and variables
begin  -- of task Input
    .
    .
    .
loop
    .
    .
    .
    .
   accept GiveData (X: in item) do
   . -- statements of rendezvous
   . -- procedure
    .
   end GiveData;
    .
    .
    .
   -- operations on
   -- input data
   -- followed by
   -- transmission to
   -- other tasks
    .
    .
    .
SendData (____);
end loop;
endInput;
```

FIGURE 3.9 Example of Rendezvous

5. The language definition foresees the possiblity of a task waiting on another. It could either be the calling task waiting on the called task or vice versa. Facilities are provided in the language for a conditional call, that is the caller is not committed to the call if the called task is not yet ready to accept it, and for selective accept, that is the called task can choose whichever entry is active.

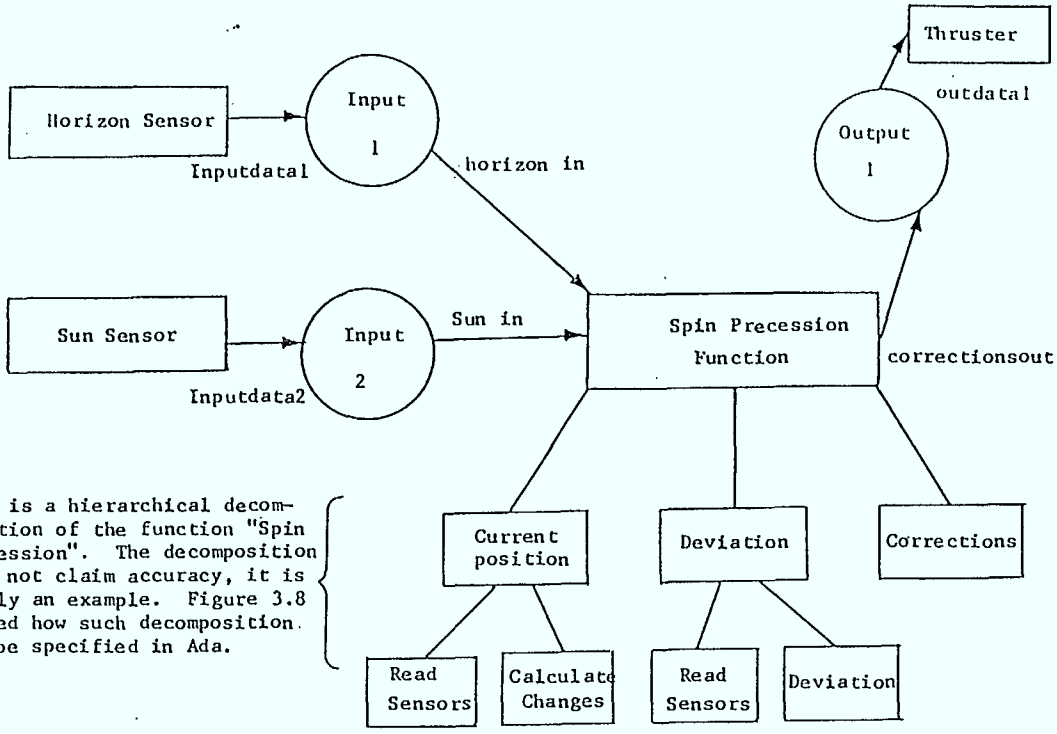
The tasks have an independent execution and, for functional specification, this translates into the benefit of having the system exercised completely at the functional level. Procedures can be left blank, or with simply a delay statement and as long as parameters and Rendezvous match properly, compilation is possible. If delay statements have been used in procedures, execution is also possible. Interestingly enough, by varying those delay durations one can obtain a good idea of execution time, hence computational load guidelines.

As a further example of how a network of tasks (or simulation blocks) can represent a system, a part of the Attitude Control System, shown in Figure 2.3, has been specified. This specification is depicted in Figure 3.10. A careful look at Figure 3.10 will reveal some important differences between specification using simulation block and specifications using a network of tasks. These differences are summarized below:

1. The Simulation Block Construct (SBC) uses an Input and Output tasks besides the "Computational" tasks; the Network of Task Approach (NT) does not. It is interesting to note, however, that the code for the input task of an SBC is found in the Rendezvous related procedure for any given call in the NT method. In other words, SBC's input and output tasks have been absorded into a task of the NT variety and the associated overhead has been dispensed with. The gain in simplicity is offset by a corresponding loss in generality. In the NT approach, a custom made design has to be

obtained; no short cuts are possible such as templates for interconnections.

[It may be argued that some form of short cut is possible in the NT method, as regards templates. Although it is true in a strict sense, the preparation of such templates produces an equally considerable overhead than the one it was to alleviate.]



This is a hierarchical decomposition of the function "Spin Precession". The decomposition does not claim accuracy, it is simply an example. Figure 3.8 showed how such decomposition can be specified in Ada.

```
task HorizonSensor is
end;

task body Horizon Sensor is
begin
     .
     .
     .
     Inputdata1 (     );
     .
     .
     .
end Horizon Sensor;
```

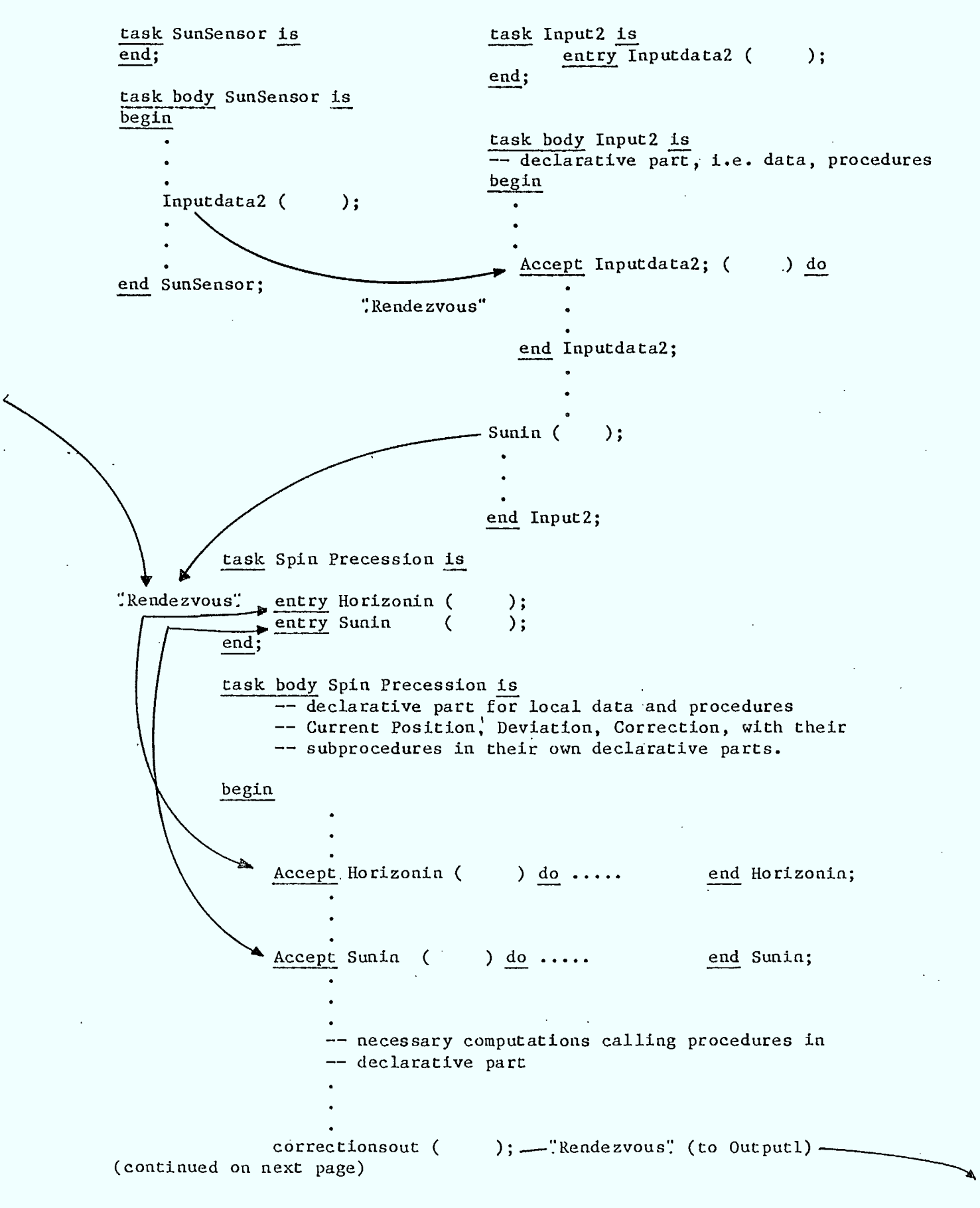
```
task Input1 is
        entry Inputdata1 (      );
end

task body Input1 is
 -- declarative part, i.e. data, procedures
begin
     .
     .
     .
     accept Inputdata1(      )do
          .
          .
          .
     end  Inputdata1;
          .
          .
       Horizon (      );
          .
          .
          .
     end Input1;
```

"Rendezvous"

"Rendezvous"
(to spin Precession)

(continued on next page)



(continued on next page)

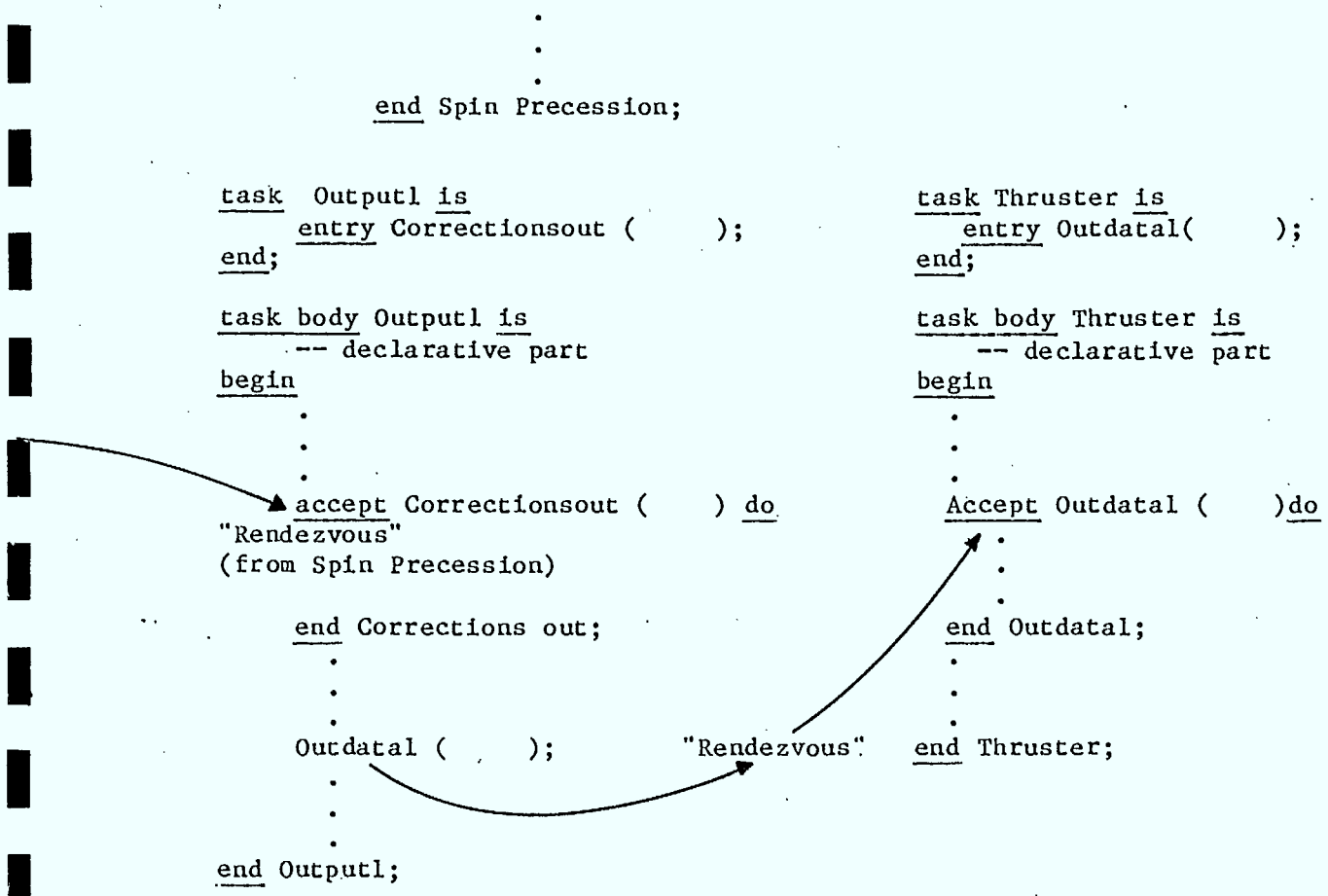


FIGURE 3.10 ACS Example With Rendezvous

2. The functionality that the specification exercise tries to capture is expressed in both cases by procedures or more precisely, by a hierarchy of procedures. The packaging of those procedures differs between SBC and NT. In SBC, procedures, subprocedures, etc., are contained in a special package called the "Functionality" package. In the NT approach, procedures are located in the declarative part of the task representing the functions being currently specified. In the task declarative part, one would also find all the data variables local to that task. Procedures can also call on procedures of their own; those subprocedures are declared and elaborated in the declarative part of their parent procedure.

3. Finally, the most obvious difference is that an entire package surrounds the SBC whereas there is no such thing in the NT case. The reasons for the package construct in the SBC were twofold:

   a) It provides a relative amount of protection and procedure grouping;

   b) It allows certain data structure to be static at execute time, i.e. during a functional simulation.

   Whenever that latter need is not essential, network of tasks can be used advantageously.

3.4 EVALUATION OF THE DESCRIPTION TOOLS

The suitability of any description tool should be assessed with respect to the following criteria:

1. Functionality and Expressive Power.

   The functionality of a given system can be expressed well by all three methods. The graphical method is good at the early stages of the design but soon suffers from being cumbersome as the function/boxes proliferate. It appears that the task model has the extra advantage of lending itself well to describing hierarchical decomposition.

2. Timing and Scheduling

Concerns about timing are usually absent from any high level specification. Timing does play a role however when the transition to a lower level description is done. In fact, timing is vital for data flow analysis and hardware software partitioning. Two types of timing can be described: synchronous and asynchronous.

In a satellite system, most (if not all) housekeeping tasks are of a periodic nature. This translates into a "Demand" or "Event" driven system which can be described by asynchronous timing. These system work well provided that there is enough computing capacity. A network of tasks, each independently scheduled by a suitable Ada run time environment can specify those systems quite well. The scheduling enables the description of a system driven by data sources whose rates are also dependent upon the scheduling of tasks by the run time executive.

Synchronous timing is a lower level technique to remove the randomness or uncertainty associated with event driven systems. The loose scheduling characterizing event driven systems is replaced by a rigid and tight scheduling based on a common clock reference. There are two reasons for migrating from asynchronous to synchronous timing:

a) to facilitate formal verification of the system;

b) to make debugging easier.

In most situation, however, demand driven systems will be specified using the network of tasks method.

3. Correctness

Design correctness is an important concern in spacecraft systems. A design can be verified by means of assertions based on predicate calculus. These assertions are inserted in the description and are used to generate the verification conditions and to prove the resulting theorems. The assertions characterize the system entirely, i.e. the input, output data and the transformations applied to the input to get the output. Assertions can also accommodate multi-level specification and they characterize the interface between levels as well.

The algebraic description tools are ideally suited for this task although programming language based descriptions can handle assertions equally well.

From the preceeding discussion, it turns out that a Network of task is an adequate description tool and is the best suited for the task at hand. Simulation block contructs could be used if special applications warrant the extra overhead introduced by the simulation blocks.

4.0 HARDWARE/SOFTWARE PARTITIONING

The purpose of this stage is to transform a high level functional description into hardware-software modules. This stage is, at best, a very difficult one to perform; the difficulties will be evident when the stage is defined.

A block diagram of the hardware/software partition stage is to be found in Figure 4.1. In that figure, there are three main parts: the input blocks, the partitioning step and the output blocks. There are three input blocks, and each will be examined separately:

1. High Level Specification.

This input is from the upper levels and is in the form of functions, linked by data flow arcs. The functions may be decomposed into a hierarchy. The function itself will become the basis for implementation, that is, will correspond to a processor. It is possible of course to combine more than one function per processor just as it is possible to break down a function into more sub functions when the target processor is not fast enough. Figure 4.2 depicts these two situations. In the top half of Figure 4.2, a system is shown to be composed of seven functions. A partitioning is done and it turns out that several functions can be amalgamated to run under one processor, (e.g. F_1, F_3, F_4). It may happen, however, that the computational load imposed by a function will not be satisfied by the target processor and in that case the function will have to be broken down. This is illustrated in the bottom half of Figure 4.2 where the function F is decomposed into three sub functions A', B', C'. Those sub functions were previously procedures in a hierarchical decomposition of Function F. The

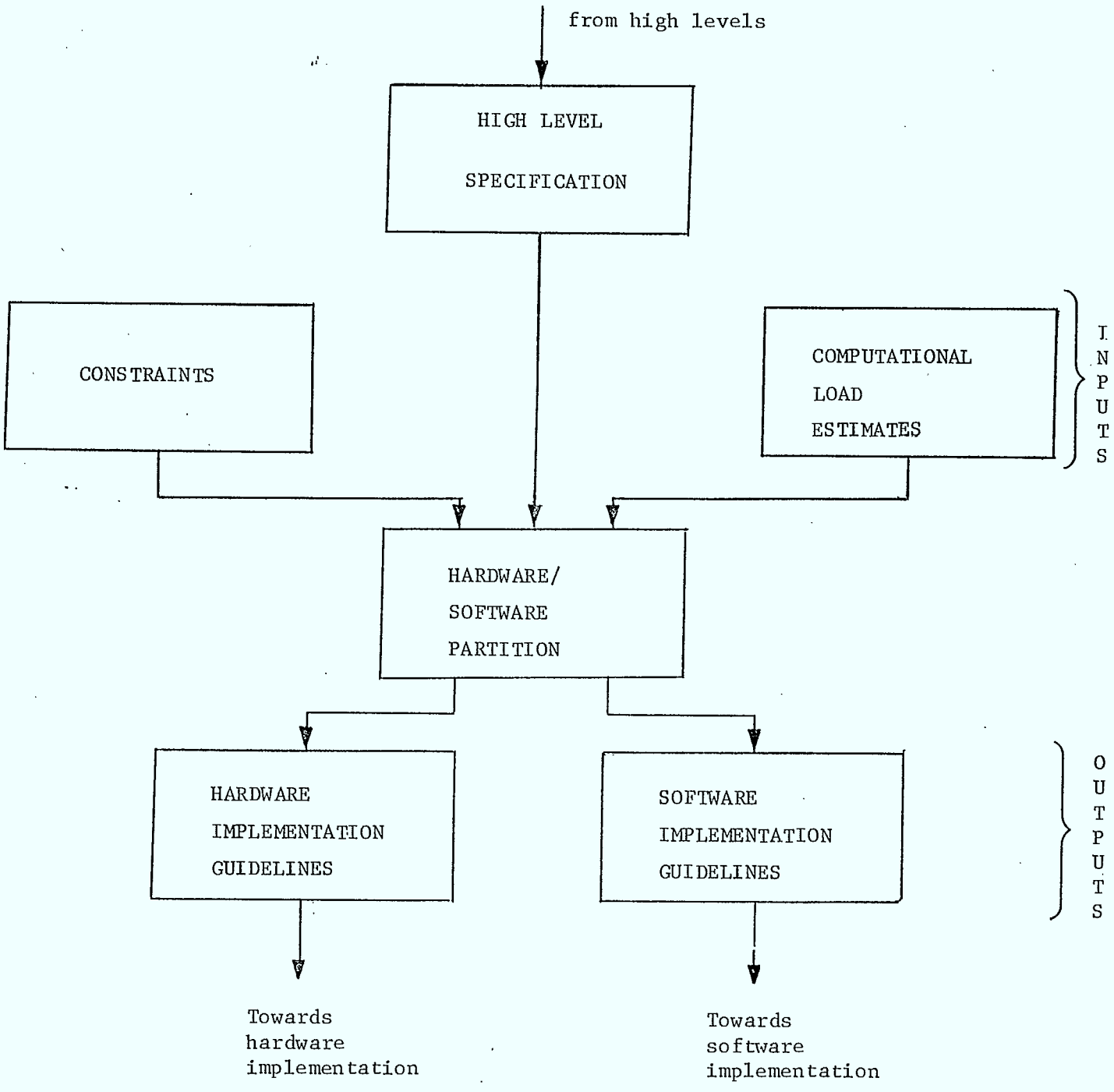
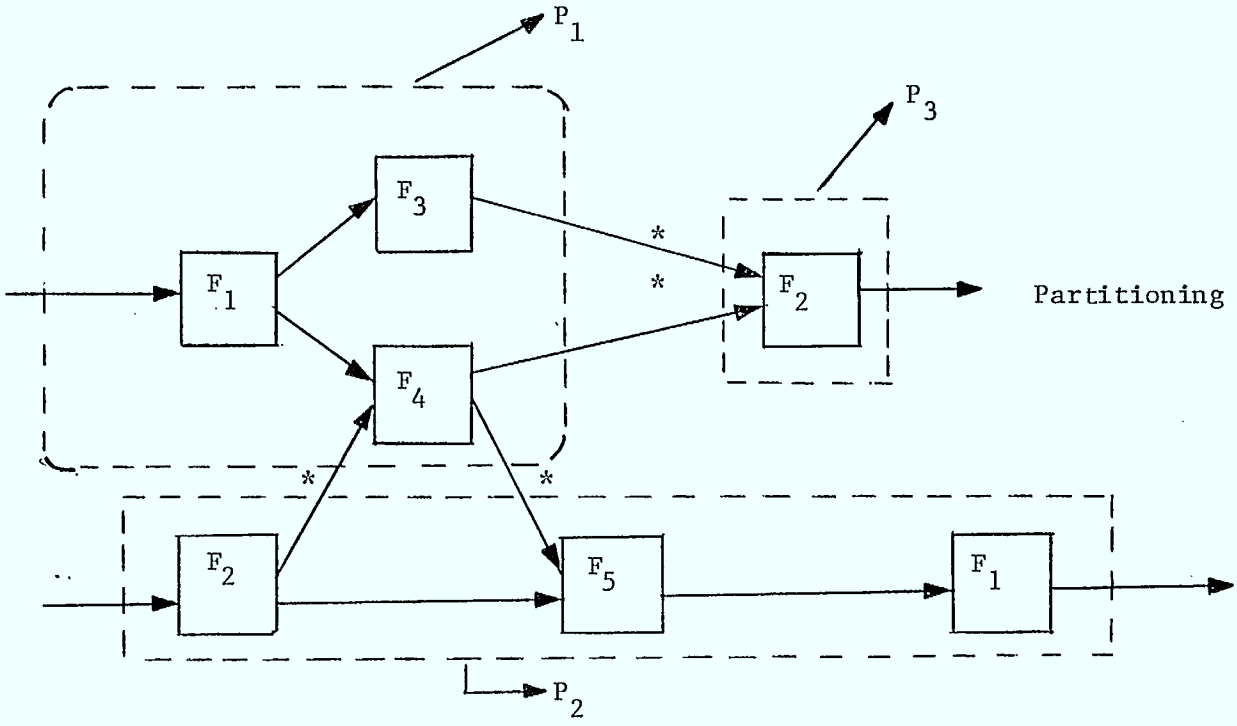
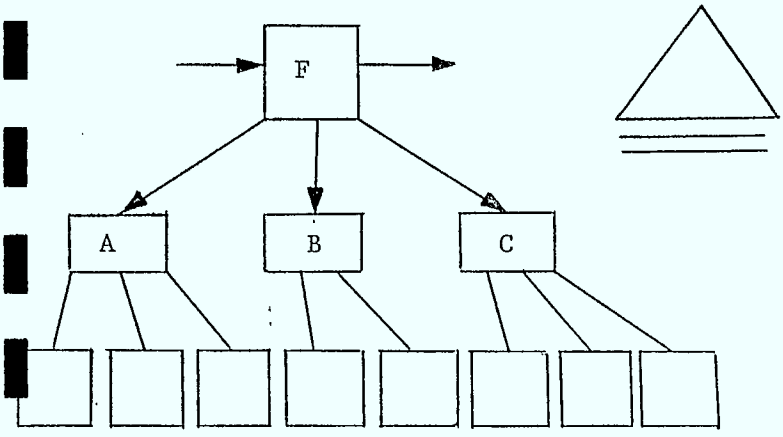
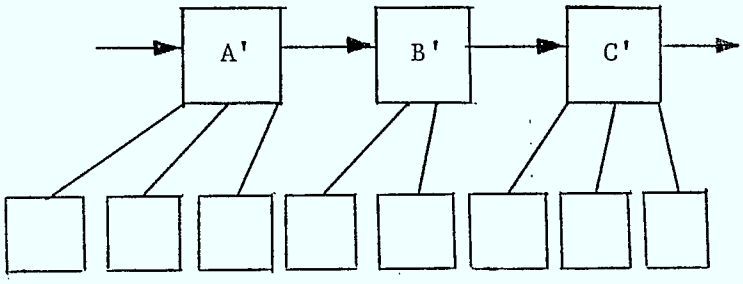


FIGURE 4.1 Hardware/Software Partition







A' = A + extra intelligence
B' = B + needed to implement
C' = C + F. In other words, F was distributed over A, B, C.

FIGURE 4.2 Function vs. Processor

diagram also shows the extra intelligence that had to be added to the original A, B, C to enable them to emulate correctly the Function F.

Incidentally, the top half of Figure 4.2 also shows interprocessor boundaries resulting from the partitioning operation. These boundaries are analogous to a cut into a graph and can give an indication of message traffic or data flow between processors. The type of interprocessor communication support is greatly influenced by the amount of traffic to be handled.

2. Constraints.

Under this heading are grouped all factors contributing to reduce the size of the solution space. Constraints should be evaluated and precisely tabulated to facilitate and enhance the design exercise. Examples of constraints would be: upper limit on power consumption, requirement for radiation hardened parts, mandatory use of a particular component on a particular architecture, etc.. The hardware/software partitioning stage makes use of the constraints to restrict its attention to feasible configurations only.

3. Computational Load Estimates

Besides constraints, computational load estimates are the other parameters used in the partitioning process. A computational estimate for a given function is an indication of how fast the function should be performed. This estimate can be established in two ways. One possibility is to look at the aggregate rate of data flow into the function and to determine the length of time at the function's disposal to perform its work. This processing time

is an upper bound and serves as an estimate of the computational load. Another way of calculating this load is by compiling the Ada high level specification of the system, replacing procedures/functions by delays and adjusting the delay parameters to have a consistent system. In both cases errors are unavoidable and can only be corrected by iterating several times through this part of the methodology.

4.1 PARTITIONING

Looking back at Figure 4.1, the next part to describe is the partitioning activity itself. At the onset of partitioning, there exists a great deal of fluidity since neither processors nor algorithms have been determined. The process of partitioning will be described using two diagrams; the first of those diagrams, Figure 4.3, depicts the whole process graphically for a Function F. In a first step, the computational load of F is estimated. Then, according to constraints, a basic processor is chosen and, according to the complexity of F, several hardware/software ratios are possible. For example, it is possible to perform a Fast Fourier Transform using software or using dedicated hardware. Having determined the hardware/software ratio, it is possible to obtain an indication of the performance level of the tentative configuration. Coupling this performance level indication with the software algorithms likely to be used, it is possible to obtain an estimate of the execution speed of the function which can be compared with the computational load obtained previously. If the execution speed of the function is such that the permissible execution time would not be sufficient, a different partitioning has to be used.

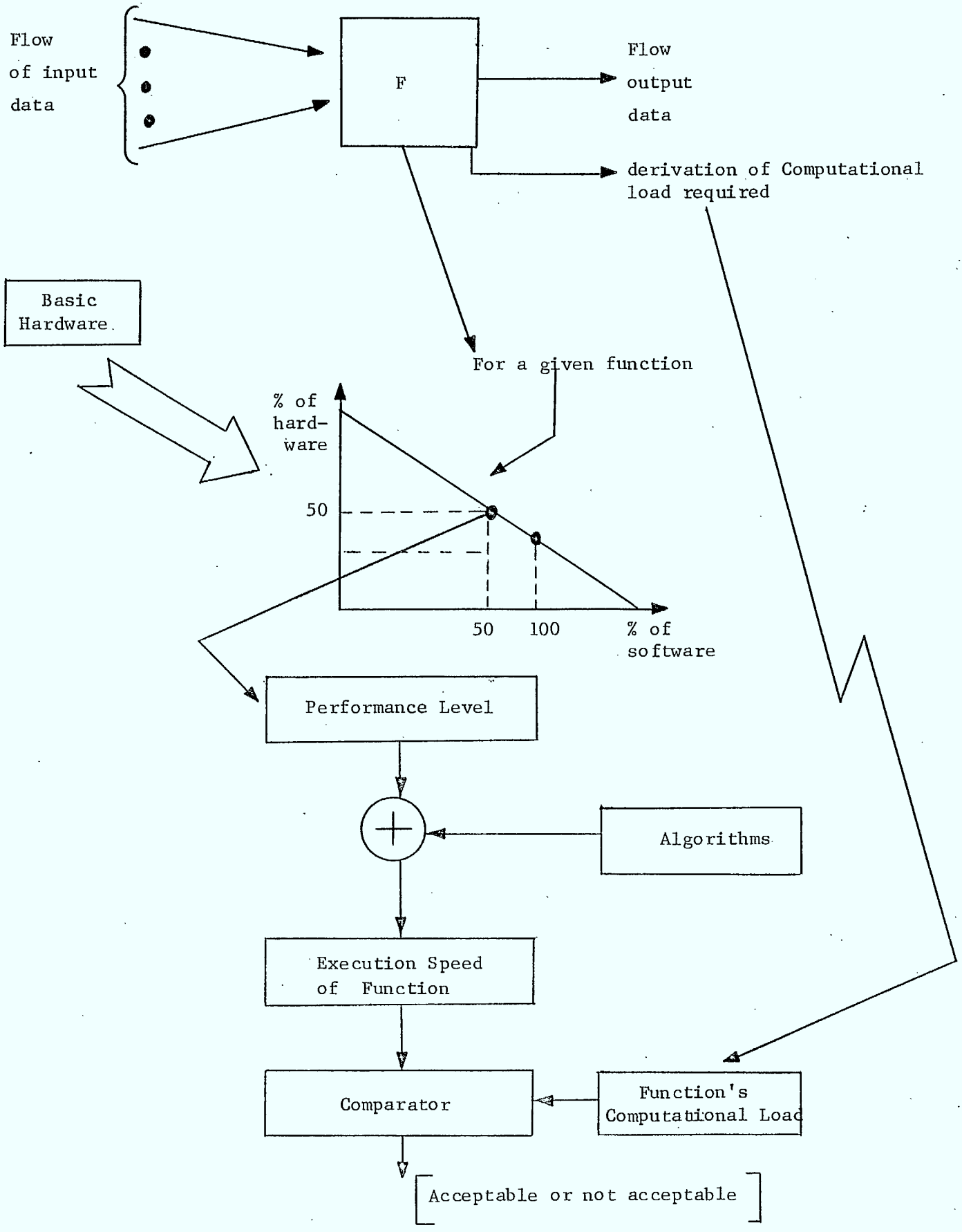


FIGURE 4.3 Partitioning Process

A different view of the partitioning process is presented in Figure 4.4. In that Figure, there is a diagram plotting computational load versus Hardware/Software ratio. Several curves, C_is, are plotted and each represents the domain of feasible implementation. For example, if curve C_i is considered, and the computational load is X then the rectangle delimited by the dashed lines represents the allowable ratios. As it stands, a ratio of about 42/58 (42% hardware, 58% software) is the ratio with the most software. If a higher ratio of software is required, then one should move to different curves.

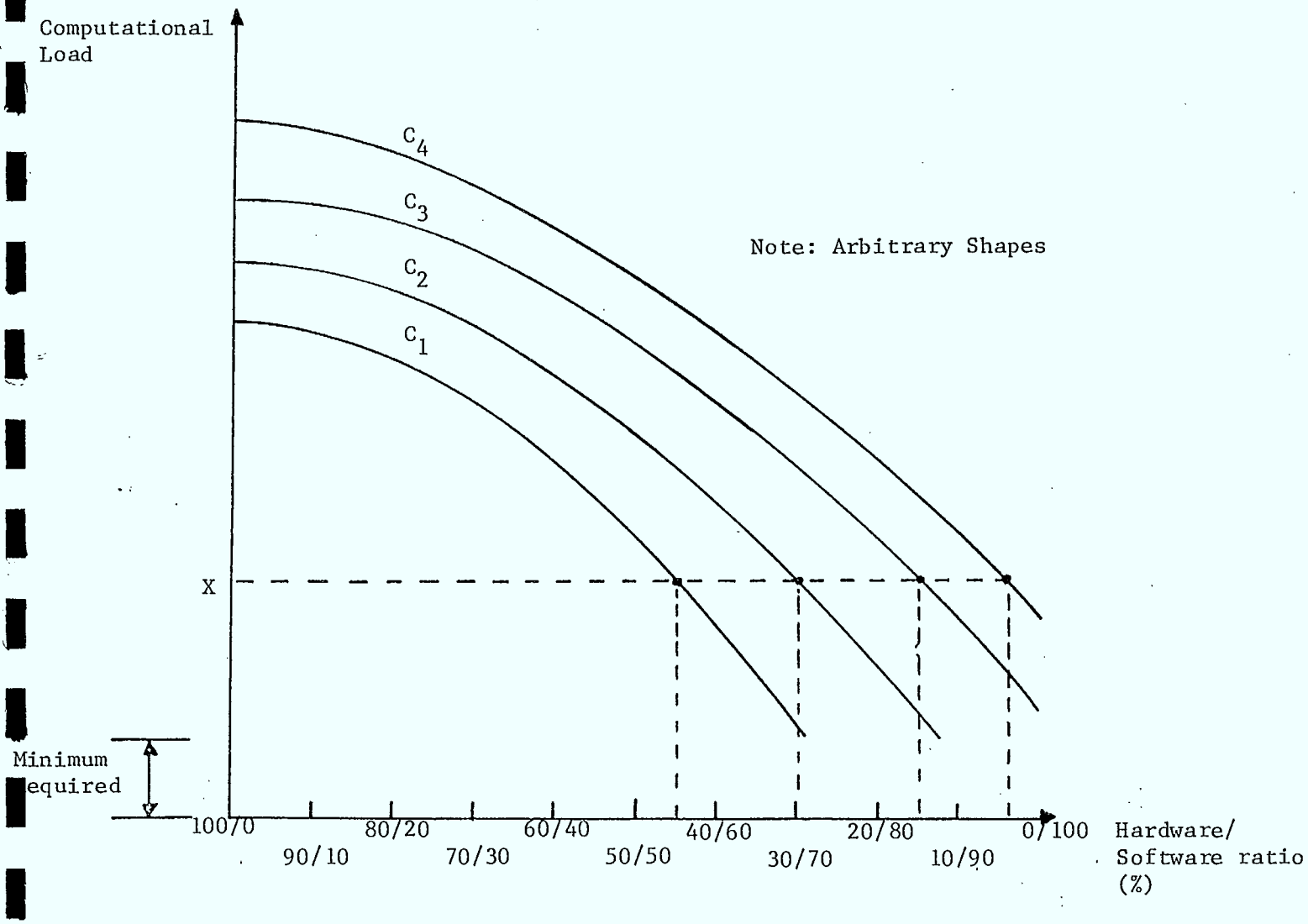
The curves differ because each represent a different set of characteristics. Those characteristics are as follows:

1. Basic hardware used, e.g. technology, speed, architecture, instruction set, memory/bus cycle time, I/O transfer rate, etc.

2. Software algorithms used, e.g. type of algorithm, implementation language, compatibility between high level languages and the hardware, effieciency of the compiler, efficiency of the implementation, etc.

In other words, each curve represents different basic choices and the graph is used to illustrate the basic choices and resulting trade-offs the designer has to reckon with.

It is also interesting to note that, in the diagram of Figure 4.4, a minimum computational load is indicated. This corresponds to basic operating services which are always necessary, irrespective of the functions being partitioned. Besides this overhead, there also exist two other types, namely the overhead imposed by distributed O/S and that originating in attempts to satisfy the reliability requirements.



Note: Each C_i has a corresponding set of parameters,

$$C_i(p_{i1}, p_{i2}, p_{i3}, \ldots),$$

which uniquely characterize the implementation curve.
For further explanation, see text.

Figure 4.4: Implementation Curves

4.2 OVERHEAD

There are two areas which are going to introduce overhead which will have to be taken into account when choosing hardware/software ratio and power. Those two areas are: reliability and distributed operating services.

Reliability is an important concern in satellite design. The equipment has to operate reliably for long periods of time and no repair work can be undertaken after launch. Satellite building and launching being such an expensive business, it makes sense to endeavour to increase overall reliability. This can be achived as follows:

1. By the use of better construction techniques for board level assembly and, especially, high quality radiation hardened components which have been thoroughly tested. This approach is common to all important hardware development projects.
2. By providing hardware redundancy (double, triple, etc.) for all major components such as processors, memory, bus drivers, busses, etc. This results in greater hardware complexity, inclusion of special purpose circuits (e.g. for reconfiguration, failure detection, etc.), greater power consumption and, sometimes, reduced processing power. Furthermore, more sophisticated hardware design techniques are usually required.
3. In addition to hardware reliability enhancements, software recovery algorithms are also necessary to provide for fault-isolation, reconfiguration, and recovery. These algorithms will introduce an extra computational load which may be considerable, especially when the system is experiencing failures.

Operating system support also introduces overhead, as mentioned before in connection with the diagram of Figure 4.4. In that diagram, the overhead was associated with local O/S support. This includes support for interprocess communications (when multitasking is used), scheduling of processes, Input/Output services for sensors, activators and communication with ground control. Besides these local O/S services, distributed O/S services will likely have to be provided because of the multiprocessor nature of the target architectures. Those distributed services include: interprocessor communication, monitoring of other processors and of their performance, taking part in majority voting (if necessary) and distributed co-ordination.

The overhead introduced by reliability concerns and O/S support has to be designed into the system. In practical terms, for a given curve C_i, this translates into a shift (See Figure 4.5) upwards, to a point where the proportion of hardware is greater. Alternatively, one may see the effect of overhead as jumping to another curve, but keeping the hardware software ratio the same. Going to another curve (higher curve C_j) has the effect of altering some basic implemenation parameters. In this case, this could mean adopting a faster hardware processor, a new architecture, faster memory chips, better software alorithms, etc.

4.3 OUTPUT OF PARTITIONING STAGE

The outputs of the hardware/software partitioning stage are a set of implementation guidelines for both the hardware and the software. This is shown in Figure 4.1. It should be pointed out, that this does not constitute a unique solution but, rather, a multitude of possible, feasible and acceptable solutions.

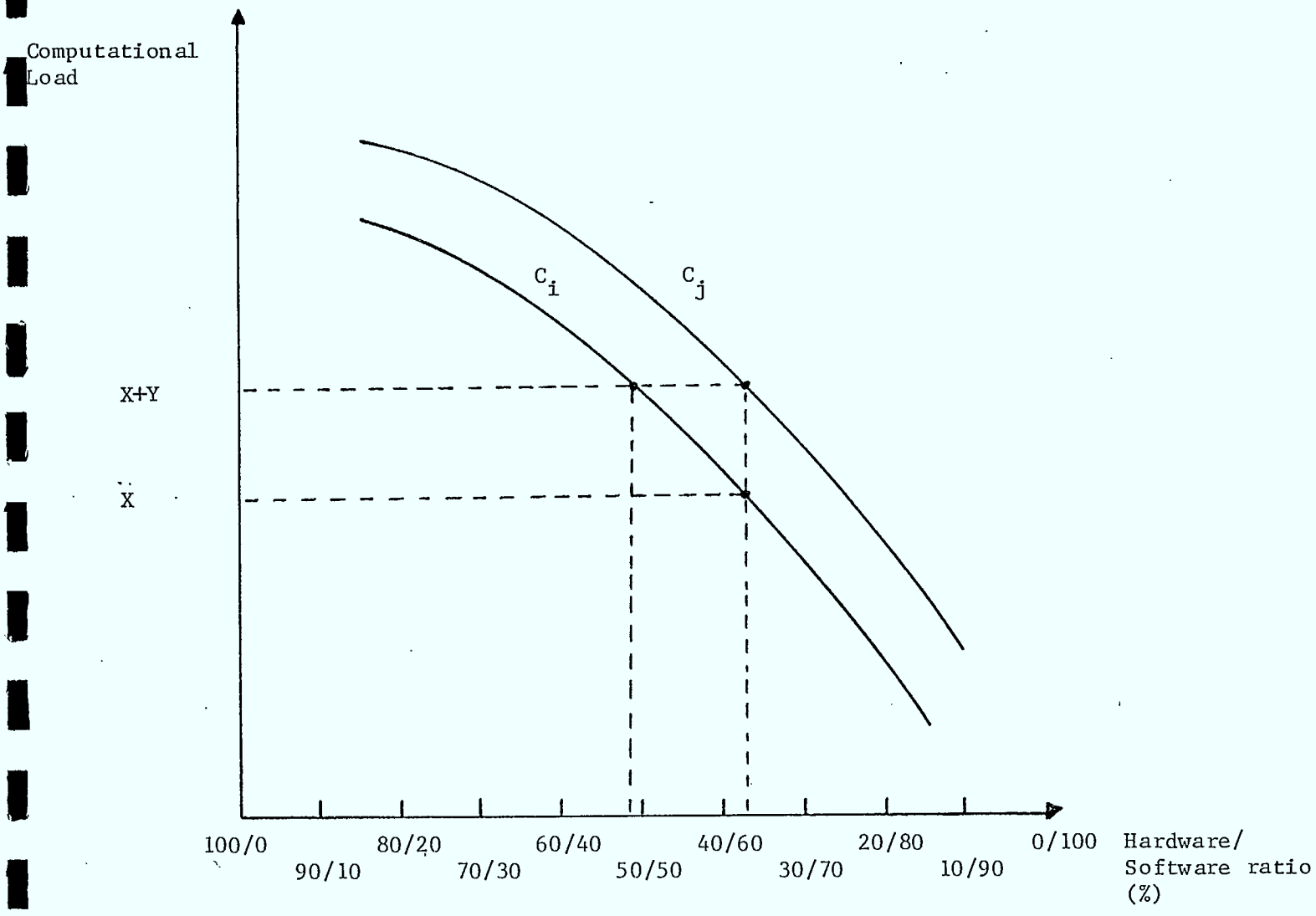


Figure 4.5: Overhead and Implementation Curves

The software implementation rules are derived, in most part, from the functional specifications written in Ada. In those specifications, there are a number of functions which were subjected to partitioning and which can be represented as independent tasks. The hierarchical decomposition of those tasks into procedure is of further help. Not all the functions will have to be processed that way; some functions will be implemented exclusively in hardware.

The hardware implementation rules will obviously include those functions that are to be realized in hardware and also guidelines as to how to implement the general purpose hardware responsible for the execution of the software. This relates to the parameters associated with each of the curves in the implemenation diagram of Figure 4.4.

5.0 IMPLEMENTATION

Following hardware/software partitioning stage, the methodology calls for the independent development of the hardware and the software. The main idea behind this stage is to avoid building physical processors and instead concentrate on a "soft" implementation of the system which can be easitly tested and modified.

5.1 HARDWARE IMPLEMENTATION

The inputs to the hardware implementation stage will be:

1. A list of processing elements and other architectural choices related to the given implementation curve(s) worked with.

2. Some extra constraints pertaining only to the hardware implemenation stage.

3. Some operator/designer directives to guide in choosing starting configuration. This capability is quite important in a Computer Aided design exercise since it allows the designer to direct the computer along the way which he/she deems desirable and promising.

It is proposed in the methodology to implement the processing elements and support circuitry using a hardware development tool such as N.mPc. N.mPc [PARK79a], [PARK79b], [ORDY79], [ROSE79], is a facility that permits the development of new hardware structures and of their software totally in software. A simple block diagram of N.mPc is shown in Figure 5.1

N.mPc can be divided into three logical sections, the hardware compiler, the software assembler/linker and the runtime environment.

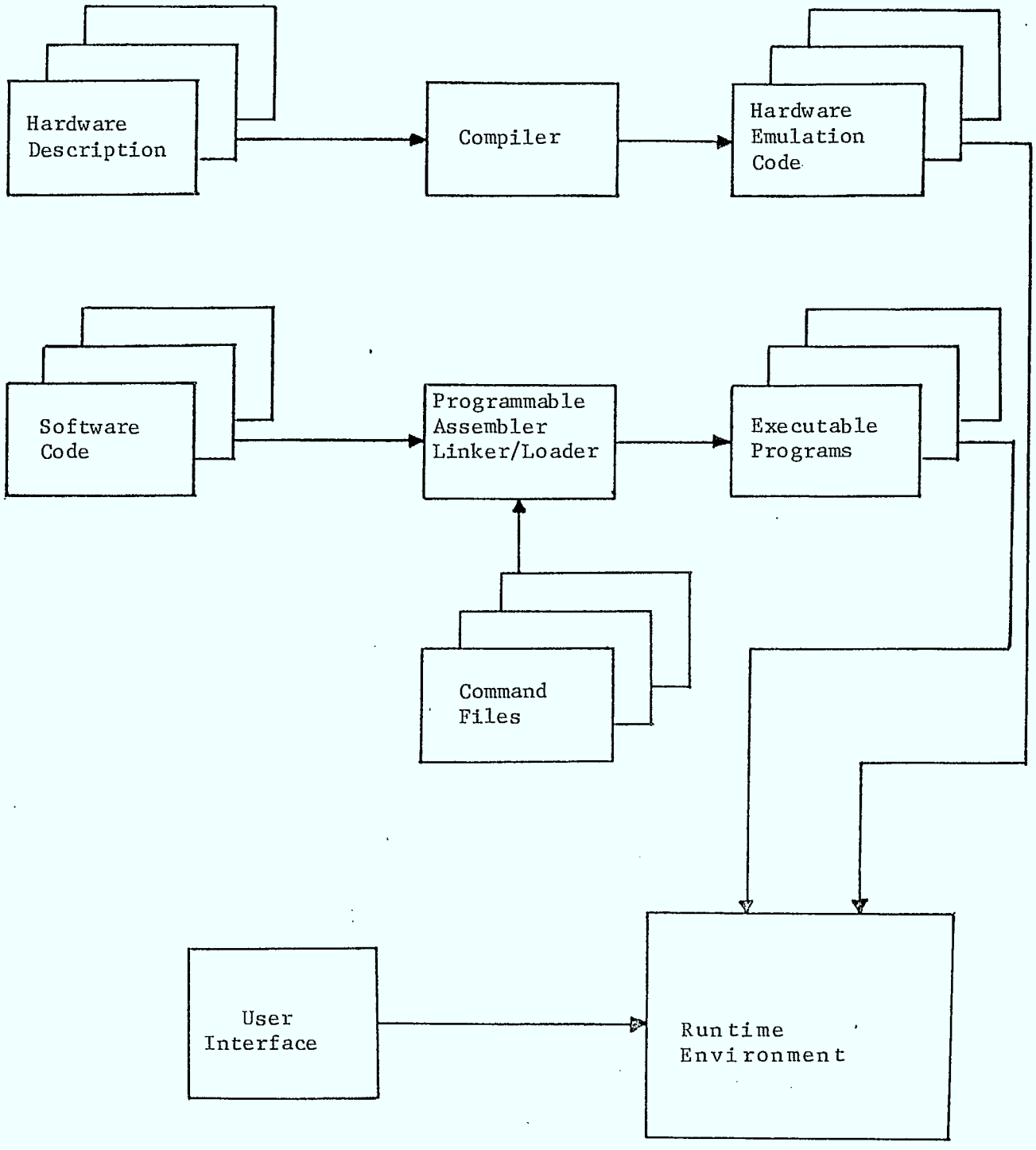


Figure 5.1: N.mPc System

1. The Hardware Compiler

In N.mPc, a hardware structure is defined by an ISP´ program. As mentioned before, hardware exhibits both a structure and a behaviour. The hardware behaviour is written into the program, the execution of which will faithfully emulate the real hardware. The structure is defined by means of "port" constructs which determine how the hardware component interact with the outside. In addition, the connections of various devices is done by the ecologist as instructed by a topology file.

2. The Software Assembler/Linker

The hardware description is compiled by the ISP´ compiler and an executable module is produced. The software on the other hand is written in assembler and is assembled and linked by the meta micro assembler and the linking loader respectively. Both those facilities are programmable, that is to say, can be made to accommodate any programmable hardware device defined in ISP´. Programming of the meta micro assembler and Linking Loader is by means of special command files, as indicated in the diagram of Figure 5.1.

3. The Run Time Environment

The run-time environment provides for integration of the various hardware modules through a topology description file. It also links the programs written for the programmable devices with the hardware emulation of those devices. Finally, it provides a run time environment for those programs as well as a user interface. The user controls the simulation entirely and the user monitor the execution of the software and of the hardware executing the software.

The above description of N.mPc and of its approach to system design is unfortunately sketchy. It does however give the flavour of the intended course of hardware design.

5.2 SOFTWARE IMPLEMENTATION

The software implemenation stage receives as inputs the high level functional specifications in which functions roughly correspond to tasks. Of course this correspondence is not binding as it may very well happen that a function assigned to a task which is in turn assigned to a processor will find that the task in question has to be broken down into several subtasks. What really matters though, is that the combined behaviour of those sub tasks is equal to that of the original task.

These functions/tasks are going to be refined and decomposed further, as indicated in the diagram of Figure 5.2. Coding is the next step and should follow naturally from the high level Ada description. The Ada code thus produced is compiled by an Ada compiler and a multi tasking application is created. This application is ready to execute on the target hardware, provided that a suitable Ada run time executive exists.

The Ada run time executive (or Ada run time environment) is a collection of the necessary support services which are of a distributed as well as a local nature. It is interesting here to note that the processing elements indicated in Figure 5.2 may be of a different nature and be totally incompatible at the machine level. The software development stage assumes the existence of an Ada run time environment on each of the processing element (denoted as "Local O/S" in Figure 5.2). The nature of the Ada language requires the existence of certain standard features such as concurrency. Since each local O/S supports

concurrency and all other Ada prerequisites, it is reasonable to assume that a physically distributed but logically centralized Ada run time environment can be created on top of the local O/S. It remains nevertheless that those [potentially different] local O/S have to be produced and that job is likely to be done in assembly language using the features of the meta micro assembler. Should a bare bone Ada comiler exist for the target machine, the job would be made easier as part of the executive could be written in Ada.

In most cases, the choice of processing elements will boil down to a few space qualified processors for which, Ada compilers and Ada run time environments will be available. The main difficulty would then be to construct a distributed executive out of the local O/S available. With the availability of a complete Ada run time environment for the target system, it will be possible to execute the application program.

5.3 HARDWARE/SOFTWARE INTEGRATION

The integration stage will take place when the hardware and software development is completed. For the integration, N.mPc will be used to mix the software emulation of the target hardware with the software code produced by the compilation of the source code together with the run time executive.

In the above fashion, the whole system can be exercised: the input sensors can be substituted by files containing test values; the actual code is executed by the simulated hardware. The values would be processed in simulated time but, otherwise, the functionality of the code would not be altered.

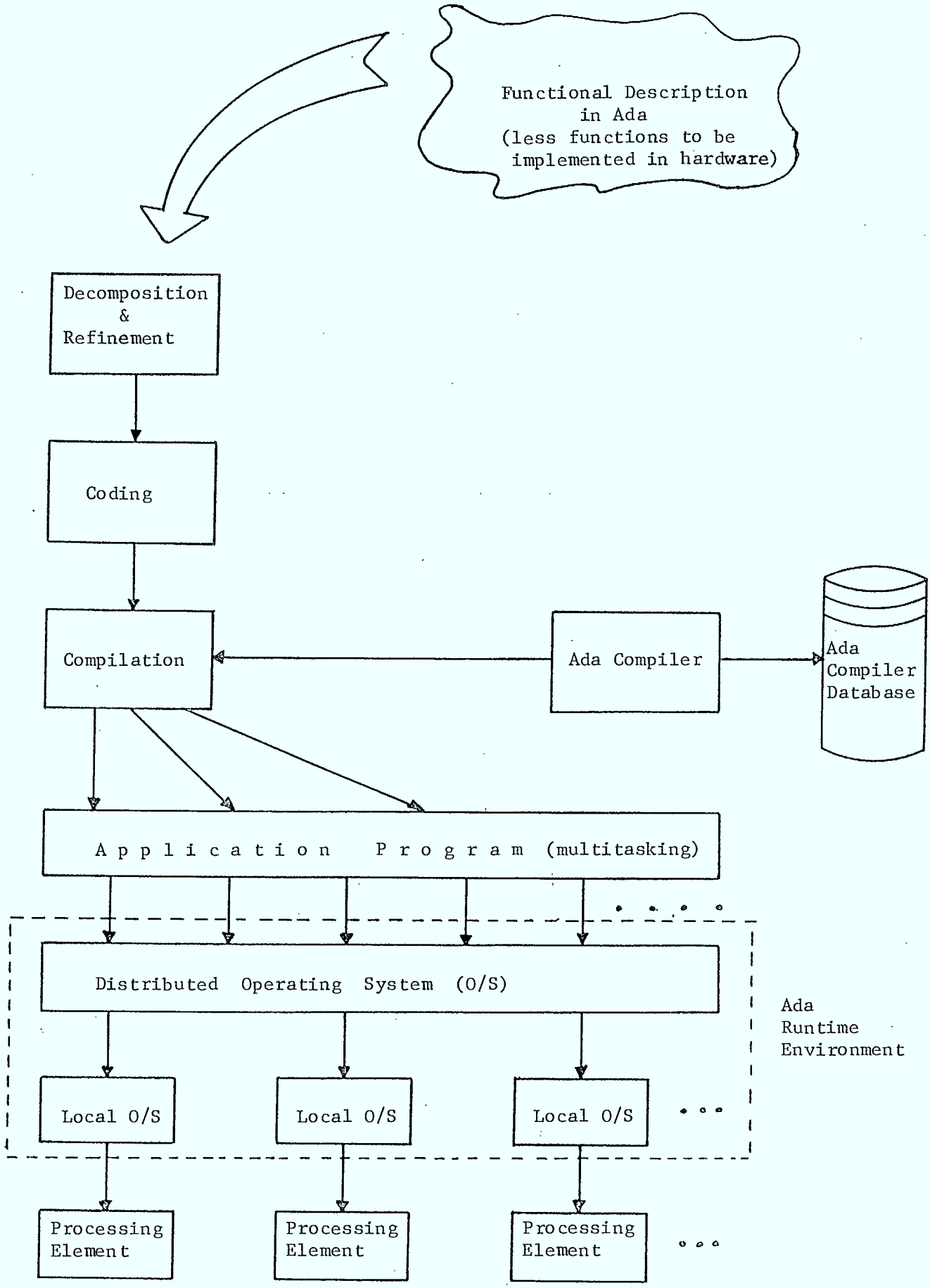


Figure 5.2: Software Implementation

The output could be redirected to files and thus an input/output comparison with the results of actual tests or analytical model could help assess the correctness of the hardware/software system.

Furthermore, if one makes allowances for the scaling factor between simulated time and real time, it could be possible to determine the performance of the system. Reliability could also be thoroughly tested by introducing various failure conditions into the system.

It is obvious that the integration stage is also a testing stage and that final acceptance of the system depends on its performing properly during those tests. If the system does not perform well it should be easy to isolate the problem and correct it. The correction can be achieved by going back to an earlier step of the methodology. The solution to a given defieciency will not be unique and it is expected that the designer will come up with a set of modifications.

6.0 CONCLUSIONS AND DIRECTIONS FOR FURTHER WORK

This report has presented a comprehensive design methodology for multi-processor systems destined to be used as on-board processors for spacecraft. The motivating factor behind the effort to develop the methodology was the need for more intelligence and information processing aboard modern spacecraft. This need can only be fulfilled by the use of advanced computers such as multiprocessor systems. Fortunately, it is now possible to implement multiprocessor systems with cheap, microprocessor based processing elements.

The use of multiple processor systems is indicated because of their greater processing power and of their inherent potential for reliability. The greater processing power of multiprocessors is based on the assumption that the tasks can be given to individual processors thus achieving greater execution speed and it also requires the necessary distributed executive services. Multi-processors can be made more reliable than their uniprocessor counterparts by adding more processors and support subsystems and by executing tasks redundantly.

To bring all the advantages of multiprocessors to bear, advanced techniques and tools are necessary for the design, implementation, building and testing phases. Furthermore, it would seem appropriate to re-use as much as possible any high level work that has continutity in lower levels. The design methodology presented herein and all its associated computer aided tools is an attempt at creating those tools and techniques. The questions that immediately comes to mind is: how feasible and practical is this methodology? The answer is in two parts, each addressing a specific set of issues.

1. In terms of tools, used in conjunction with the methodology, N.mPc's primary function is in the hardware design phase and the Ada language serves both as a functional specification language and as a software development language. N.mPc is an established technology although very recent and still undergoing changes and improvements. Ada is fast emerging as the standard language for embedded systems. No Ada compilers are available at the present time although it is expected that several will be released in the short term future.

2. The methodology itself is made out of several parts and there are still some undefined areas. The functional decomposition stage is relatively easy to visualize and similarly for the software development and implementation which is based on the functional decomposition. The most difficult stage is the hardware/software partitioning. The difficult areas in that stage are the data flow analysis, the obtaining of the implemenation curves and the estimation of the overhead introduced by the necessary O/S support services and by the reliability techniques. The last stage, integration and testing is straight forward in that it uses N.mPc almost entirely.

In view of the above remarks, it would appear that further work should concentrate on the hardware/software partitioning in general and on overhead estimation in particular. As mentioned before, the partitioning stage includes data flow analysis, derivation of implemenation curves and estimation of overhead. It is suggested to concentrate on the last item, estimation of overhead, as it is necessary for proper use of the former two components of the partitioning.

There are two main concerns within the estimation of overhead activity: O/S support overhead and reliability overhead. The former, O/S support, will be determined by the Ada run time environment either distributed or local and will be available when the executive is implemented. The latter, reliability, is of great importance and it is desirable to investigate futher the following issues:

1. Design techniques for reliable systems;

2. Estimation of overhead introduced by several reliability techniques and how the overhead relates to the resulting system robustness;

3. Selection of an adequate configuration to satisfy the reliability requirements (and other related requirements);

4. Determination of the impact of reliability on software especially as software recovery algorithms are to be used in reliable systems.

REFERENCES

[CARN83] P.C. Carney "Selecting On-Board Computer Systems", IEEE Computer, pp. 35-42, April 1983.

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

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

[DUSI83] E.W. Dusio, T.P. Murphy and W.F. Cashman, "Communications Satellite Software: A Tutorial", IEEE Computer, pp. 21-34, April 1983.

[LAFE82] C. Laferriere, W.T. Brown, J.G. Ouimet and S.A. Mahmoud, "The Definition and Specification of an Integrated Set of CAE Tools for Spacecraft Multiprocessor System Design", Technical Report No. INT-82-16, Intellitech Canada Limited, Ottawa, Canada, March 1982.

[LAFE83a] C. Laferriere and A. Lam, "N.mPc and its Utility for Spacecraft Applications", Technical Report, Intellitech Canada Limited, Ottawa, Canada, 1983.

[LAFE83b] C. Laferriere and A. Lam, "N.mPc and its Utility for Spacecraft Applications: N.mPc Simulation Listings", Technical Report, Intellitech Canada Limited, Ottawa, Canada, 1983.

[LAFE83c] C. Laferriere, "Computer Aided Engineering Tools for Spacecraft Multiprocessor System: Status Report for January 1983", Intellitech Canada Limited, Ottawa, Canada, January 1983.

[ORDY79] G.M. Ordy and F.I. Parke, "An Evaluation of the N.mPc Design Environment", Proceedings of the 16th Design Automation Conference, pp. 537-541, June 1979.

[OUIM82] J.G. Ouimet, et al, "Review of Multiprocessor Systems and their Spacecraft Applications", Technical Report, Intellitech Canada Limited, Ottawa, Canada, March 1982.

[PARK79a] F.J. Parke, "An Introduction to the N.mPc Design Environment", Proceedings of the 16th Design Automation Conference, pp. 513-519, June 1979.

[PARK79b] F.J. Parke, et al, "The N.mPC Run Time Environment", Proceedings of the 16th Design Automation Conference, pp. 529-536, June 1979.

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

[ROSE79] C.W. Rose, et al, "The N.mPc System Description Facility", Proceedings of the 16th Design Automation Conference, pp. 520-528, June 1979.

[THEI83] D.J. Theis, "Spacecraft Computers: State of the Art Survey", IEEE Computer, pp. 85-97, April 1983.

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

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