NASA CR-
14163

P-019

A DISCUSSION OF HIGHER ORDER SOFTWARE CONCEPTS AS THEY APPLY TO FUNCTIONAL REQUIREMENTS AND SPECIFICATIONS

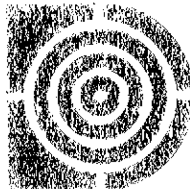
by

Margaret Hamilton

December 5, 1973

(NASA-CR-141653) A DISCUSSION OF HIGHER ORDER SOFTWARE CONCEPTS AS THEY APPLY TO FUNCTIONAL REQUIREMENTS AND SPECIFICATIONS (Draper (Charles Stark) Lab., Inc.) 23 p HC CSCL 09B G3/61
N75-18920
Unclas 12495

Reproduced by
NATIONAL TECHNICAL INFORMATION SERVICE
US Department of Commerce
Springfield, VA. 22151



The Charles Stark Draper Laboratory, Inc.
Cambridge, Massachusetts 02139

PRICES SUBJECT TO CHANGE

P-019

A DISCUSSION OF HIGHER ORDER SOFTWARE CONCEPTS AS THEY APPLY TO FUNCTIONAL REQUIREMENTS AND SPECIFICATIONS

by

Margaret Hamilton

December 5, 1973

THE CHARLES STARK DRAPER LABORATORY, INC.
CAMBRIDGE, MASSACHUSETTS 02139

Approved:
D. G. Hoag

ACKNOWLEDGEMENT

This report was prepared by the Charles Stark Draper Laboratory under Contract NAS9-4065 with the Lyndon B. Johnson Space Center of the National Aeronautics and Space Administration.

The publication of this report does not constitute approval by the National Aeronautics and Space Administration of the findings or the conclusions contained herein. It is published only for the exchange and stimulation of ideas.

TABLE OF CONTENTS

Page

REQUIREMENTS. 2

SOFTWARE 2

DESIGN. 3

SPECIFICATIONS 3

ALGORITHM 4

STRUCTURED PROGRAMMING. 4

PROGRAM STRUCTURING 5

STRUCTURED DESIGN CONSIDERATIONS FOR REQUIREMENTS. 5

PROGRAM STRUCTURING CONSIDERATIONS FOR REQUIREMENTS 7

DESIGNS THAT SHOULD NOT BE INCLUDED IN THE REQUIREMENTS 13

SPECIFICATIONS DESIGN CONSIDERATIONS. 15

SUMMARY 17

REFERENCES 19

A DISCUSSION OF HIGHER ORDER SOFTWARE CONCEPTS AS THEY APPLY TO FUNCTIONAL REQUIREMENTS AND SPECIFICATIONS

In order to demonstrate Higher Order Software (HOS) concepts as they apply to the Shuttle Software System, a definition of terms is given. The concepts discussed and the terms defined here are intended to be consistent with Reference 1 and Reference 2. What is important, of course, is not that these terms, per se, be used; but that the concepts discussed below be more clearly understood.

A forthcoming CSDL memo will discuss, in detail, the application of HOS techniques to an actual prototype MERCURY Module.

Consider the top-down problem solving process of a Shuttle Applications example, Entry Guidance. The major steps might be as follows:

Step 1 - Shuttle Program Mission Requirements

Step 2 - Entry Guidance Mission Requirements

Step 3 - Entry Guidance Avionics System Requirements

Step 4 - Entry Guidance Software System Requirements

Step 5 - Entry Guidance Software Functional Requirements - (Requirements Design Phase)

Step 6 - Entry Guidance Software Architectural Requirements - (Specifications Design Phase)

Step 7 - Entry Guidance Software Verified Code.

(It is important to note here that Entry Guidance is an example of a Section A Applications Module as defined in Reference 1, pages 6-11. The relationship of the Applications Modules to the Systems Modules* are defined in more detail in the Reference 1 report.)

REQUIREMENTS - are "something wanted or needed."[5]
In the above example, Step 1 defines the Mission Requirements of the Shuttle Mission. Entry Guidance is defined as one of the Mission Requirements from Step 1. Step 2 defines the Mission Requirements of Entry Guidance. The Avionics System is one of these Requirements. Step 3 further defines requirements needed by Entry Guidance from the Avionics System. Software is one of these Requirements. Step 4 defines requirements to Step 5 of the flight computer program for Entry Guidance. Step 5 takes these software Requirements and produces Functional Requirements for Step 6. The Functional Requirements in Step 5 define how the software will solve the problem. It is in this Step that the software functional algorithms are designed. (This Step, the Functional Requirements, has been called different things by different people. Examples of other recent labels are: Design Equations[6], Level 1, Book 1, etc.). HOS refers to this phase of software as the Requirements Design Phase. Step 6 defines the Architectural Requirements for Step 7 (i.e., the detailed Functional Requirements of the software). HOS refers to this as the Specifications Design Phase. The final Step, Step 7, produces the verified flight computer code.

SOFTWARE - In each problem to be solved, HOS begins with the formulation of the problem to be solved by a computer and ends with the final verified code. In the example, Steps 5-7 are pure software steps. That is,

*System Modules include FCOS and the Applications Support Software as defined in Reference 3. The HOS concepts discussed here apply both to Applications Modules and Systems Modules and are intended to be consistent with the aims of Reference 3 and Reference 4.

the software process begins once a requirement for that process has been defined.

Shuttle flight 'software', therefore, begins with Step 5, the design of an 'algorithm' for the flight computer, and ends with Step 7, the completion of the verified code.

Discussion here will concentrate on HOS techniques as they apply to Steps 5-7, i.e., the pure software development steps. (It should be noted that HOS techniques are recommended for the higher levels, Steps 1-4 as well, for the process of software design is not unlike the process of design in general.)

DESIGN - a process whereby one "conceives and plans."[5]
If we look at Steps 1-7, there is a continuing design process in each Step. The design for each Step is involved in producing Requirements for the next Step.

The design process in Step 1 produces requirements for Step 2. The design process in Step 2 produces requirements for Step 3, etc. Each Step evolves from the previous Step. In this discussion we will concentrate on the design processes of Steps 5 and 6 where Step 5 produces the Functional Requirements and Step 6 produces the detailed Functional Requirements (i.e., specifications design) of the flight computer software product. Unless specified, 'design' is used here to refer to the 'design' process of both Steps 5 and 6 (i.e., Requirements and Specifications).

SPECIFICATIONS - "a detailed precise presentation of something or of a plan or proposal for something."[5] From a global point-of-view, for a given Step, one could say that the specifications evolving from the previous Step (Step -1) are the same as the requirements for the next Step (Step +1). Following from Step 5, Step 6 is the Software Specifications. Step 5, the Requirements Step, is the first design of how the software does the job (including functional and performance considerations). Step 6, the

Specifications Step, takes the requirements from Step 5 and evolves them into further detail by taking into consideration the architectual aspects such as the software tools (e.g., computer and language) and software restrictions (memory and timing).

<u>ALGORITHM</u> – is a rule or process for solving a problem. There are 1) pure mathematical algorithms (such as a software square root algorithm), 2) pure logical algorithms (such as a data management routine), or, 3) algorithms representing both mathematical and logical considerations.

In the software design process, algorithms are not developed until the beginning of Step 5, i.e., the Requirements Step. At this time, algorithms are designed from a functional point-of-view (that is, the performance of the problem in question must be considered as well as the reliability of the software that will result). A further evolvement of the algorithm design is carried out in Step 6, the Specifications Step. At this time, architectual considerations are brought into play.

Many of us, unfortunately, have been confusing the terms of 'structured programming' and 'program structuring.' One reason is obvious: the terms are too much alike. Until we can come up with a better name for program structuring, however, these terms will be used for the sake of consistency. (Some of the terms that have been suggested for program structuring are: program organization, design organization, heirarchical organization, decomposition of modules, and HOS organization.)

For every module (e.g., the GN&C module at one extreme and the lowest sub-module within GN&C at the other extreme), concepts of structured programming and program structuring are applied.

<u>STRUCTURED PROGRAMMING</u> - the exercise of organization and discipline in the Requirements and Specifications design process. Design segments are arranged sequentially; equations and data are organized so that the flow

of the design is visible. Each logical action in the design is expressed by means of structured constructs (DO WHILE, IF THEN ELSE, DO CASE, DO FOR). We recommend expressing structured designs using the structured design diagrams (pages 12-15, Reference 2 and page 19, Reference 1). A well-known example of proper use of constructs is to eliminate GOTO's.[7]

PROGRAM STRUCTURING - whereas structured programming applies more to a design/programming style, program structuring applies to the process of defining modules and their interfaces (recommended program structuring techniques are described in Reference 1, pages 16-20, 29-31, etc.) There are two major phases of program structuring. The first phase defines the functions and their interfaces from a strictly performance and functional point-of-view. The second phase is concerned with software architecture and is dependent on the language, computer(s), etc. A natural division is to have Requirements worry about the first phase and Specifications worry about the second phase (Reference 1, pages 50-51).

STRUCTURED DESIGN CONSIDERATIONS FOR REQUIREMENTS

Requirements are being produced by several organizations and by many engineers; they are using various tools and techniques. The most important and most practical standards needed at the Requirements level are that the Requirements designs:

- be structured designs. Structured designs can be produced no matter what language or verification tool is used in the design verification process. For example, if a Requirements design is verified using a language without constructs, such as DO CASE, the design in the Requirements book can still be represented by a DO CASE.

- represent the "start" of the Specifications, i.e., they should not go into a detail inconsistent with or beyond the Specifications level. An example of a Requirements design that is both inconsistent and goes beyond the Specifications level is one which shows an indexing scheme which is beneath the level of a DO construct. Another example is when the accuracy (precision) of a Requirement depends on the set of operations in the algorithm[8].

- be presented using structured design diagrams to make the designs easier to communicate to the Specifications designer*.

- use terminology that is uniform wherever possible. It is entirely possible that the engineer will use μ at the Requirements level and leave it up to the Specifications engineer to define MU at the language level. This would certainly make sense if the design verification were done by hand or with a hand calculator. However, if the Requirements engineer provided structured design diagrams using MU and/or wrote a HAL program, using MU, for μ the Requirements level has been carried closer to the Specifications level in the process of completing design verification. It would thus seem more practical for this next step of the design also to be included with the Requirements. The Specifications engineer would have the option of changing the terminology if there were a valid reason for it.

Those Requirements which have been verified using HAL as a language will be closer to the eventual recommended Specifications than those Requirements which have been verified using some other language (e.g., FORTRAN or MAC). Those Requirements using structured techniques with HAL notation will be much closer to the eventual recommended Specifications than those Requirements using structured techniques without HAL notation. As to notation and design style, then, some Requirements will be further than others.

*See pages 39-45 of Reference 1 for example of structured design diagrams.

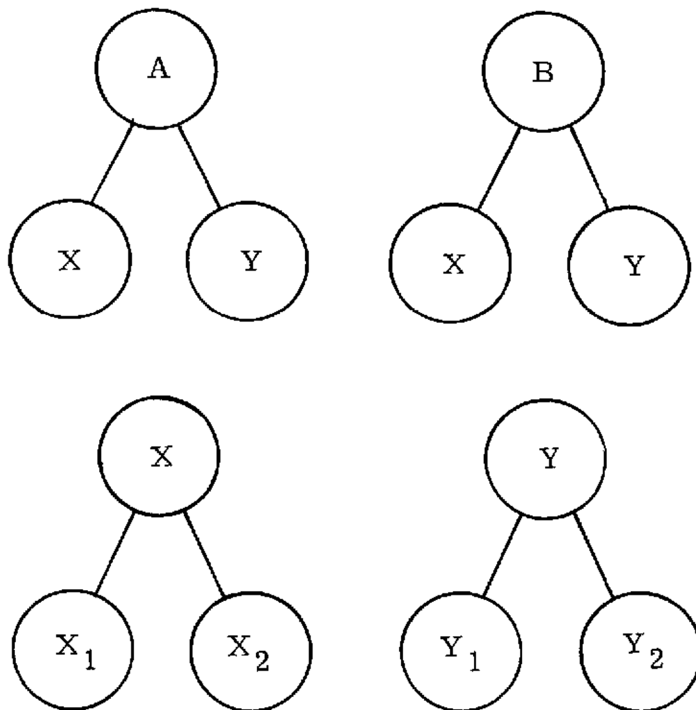
PROGRAM STRUCTURING CONSIDERATIONS FOR REQUIREMENTS

It is especially important that the first major phase of program structuring occur at the Requirements level; for this is the level that should determine what the functions (and thus resulting modules) are and how they should interface with each other (at least from a strictly functional and performance point-of-view). A major concern of the first phase is to structure the functions* so that they will work no matter what the architectural constraints are. For example, if architectural changes have to be made later on in the Specification process due to memory restrictions, an architectural change should be able to be made safely without changing the functional design or performance characteristics. A function at the Requirements level should not depend on its working, for example, as a PROCEDURE, or on how many or how few multiplies there are. This is what we mean when we say that the Requirements Step should produce flexible functional designs. Without the structuring process during the Requirements phase, a new functional and performance design phase would be necessary later in order to both produce the eventual reliable software, as well as to confirm the reliability of the stand-alone Requirements, i.e., there would be two design efforts, rather than one. Without the structuring process at the Requirements level, the original designs are to be trusted less (for after all, they are software too, since they are verified using software). In addition, in order to allow for reliable and efficient comparisons of the design at both the Requirements and Specifications Steps, the functional relationships should be maintained throughout both the Requirements and the Specifications Steps. With functional consistency, new designs (those applying program structuring) would have valid baselines for comparisons. Test results could then be compared at meaningful checkpoints which in many cases might no longer exist if the functional relationships were no longer the same between the Requirements and Specifications Steps.

*See page 50 of Reference 1 for an example of a functional structure.

HOS Techniques[1] assume program structuring to include top-down, modular* design. Since the terms "non-unified" and "unified" have been used in making design trade-off decisions, some definitions are in order. For it is possible that what some people think of as "unified" could very well be "non-unified" and vice-versa.

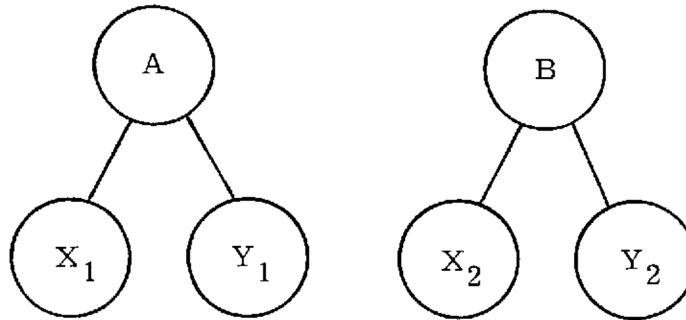
By HOS definition,[1] a typical unified module is one where only that module in a software system assigns a given data set; and where that module must ask about control information already known on a higher level.



Where modules for X and Y assign a data set for A using the set of operations of X_1 and Y_1; where modules for X and Y assign a data set for B using the set of operations of X_2 and Y_2.

FIGURE 1: Module Members of a Unified System

*We distinguish what we mean by modular since it means different things to different people. Modularity is defined in Reference 1, pages 20-27



Where modules for X_1 and X_2 assign the same data set but the set of operations for X_1 is different from the set of operations for X_2; where modules for Y_1 and Y_2 assign the same data set but the set of operations for Y is different from the set of operations for Y_2.

FIGURE 2: Module Members of a non-unified system*

As design modifications are made to a unified module, the resultant logic can easily become more complex and thus harder to understand, more difficult to verify and more difficult to change; for often the decision to go "unified" is made solely for reasons such as memory saving or common use of common data without regard for software reliability. On a long-term project, such as the Shuttle, the flexibility to make reliable changes is a major factor. Forcing a module to become unified ultimately results in misuse of control decisions. An example of this is redundant control decisions being made at lower levels that are or should have been made at higher levels. If a decision is made to go "unified" for some modules, the levels above those modules should be clearly understood in order to prevent the necessity of pulling apart an obsolete integration of sub-modules within a unified module.

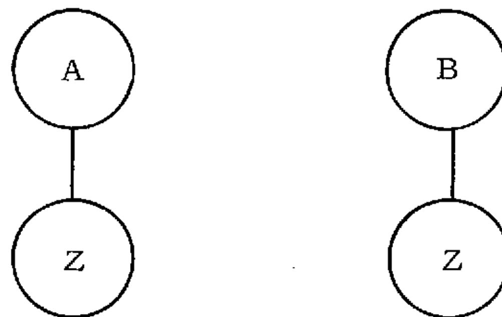
It appears, then, that the first step in the Requirements process would be a non-unified step. If unified modules are to be designed, they would be integrated from the previous step.

*If more than one function requires an existing non-unified module (e.g., if C requires the function performed by X_1 for A_1, then C can invoke X_1.

At present, trade-off studies of non-unified vs. unified designs are being made in order that more definitive guidelines can be defined for practical application to the Requirements integration process.

The use of non-unified designs does not mean that there cannot be common use of common functions (a further refinement of common use of common functions is an important role for the software Specifications design phase).

Here A and B invoke the common function Z.



The Z module performs the same set of operations for A as it does for B. Z operates on a data set specified from A. Z operates on another data set specified from B.

FIGURE 3: Common Functions

The use of common functions not only saves storage, but it also saves on the design, implementation and verification time of both the Requirements designs and the flight software. Higher Order Languages encourage common functions (e.g., Matrix and Vector instructions, SIN routines, etc.). Common functions at a much higher level (some day these could become part of a higher-higher order language) should be likewise encouraged, since they enhance considerably the reliability of the software (for the same reason that an algorithm coded in HOL is more reliable, i.e., easier to use, read, verify, etc. than one coded in assembly language). The important thing that must be adhered to in use of common functions is to guarantee that the function in question is a valid one (see rules for defining valid functions on pages 12-31, Reference 1, and the summary of rules on pages 12 and 13 of this memo). Determining the validity of a function is a major part of the process of program structuring. It is this phase of program structuring that should not only come first, but become an integral part of the Requirements design phase.

Specific reasons why program structuring from a functional design point-of-view is so important are applicable not only to the final resulting flight software, but to the reliability of the Requirements designs themselves. The proper application of program structuring therefore affects both phases of software development. The end result of applying the first phase of program structuring to the Requirements level of software is that the Requirements are:

- safer
- more flexible to work with at both the Requirements and the Specifications levels
- easier to modify
- cleaner and simpler
- more modular-easy to plug in modules or take out modules
- able to be frozen - each module is a unique function that can exist on a stand-alone basis as well as be replaced by another frozen module

- easier to understand
- able to be changed without affecting other areas of software
- not compromised by modules being forced to satisfy all needs to all users (if a higher level changes, a lower level module is not dependent on that change)
- compatible for system integration - the same rules apply for all modules
- easier to verify - fewer paths for exhaustive testing
- easier to develop at different times or in parallel - no function is held up by another one
- integration is less costly and more reliable. (If, for example, all main modules were waiting for a unified common module to be developed, all modules would have to wait for each other's needs to be incorporated into the unified module. Likewise, the unified module has to wait for all modules for itself to be completed. This could result in a "deadly embrace" of a design effort!)
- not required to know about other functions to be completed
- made up of fewer interfaces
- defined to prevent an improper function from being distributed among many different modules, thus preventing more complex logic in each different module as well as in a function itself.

The following is a summary of HOS program structuring considerations at the Requirements level of software (see Reference 1, pages 12-31 for more detail):

- every decision made within a module is directly related to the module function (e.g., an IMU moding module would not depend on crew response procedures)
- data produced per function is related only to module function (e.g., a navigation module should not produce TGO where TGO should be produced by a guidance module)

- a module function should not specify how it is to be used (e.g., a module should not SCHEDULE itself)
- a module may only invoke lower level modules
- a module may only invoke other modules to perform its particular function (e.g., navigation does not invoke the DFCS)
- a function (and resulting modules) should be designed with sub-functions, top-down
- known information on a higher level should not be brought up again at a lower level. For example, if a mode is already known, it should not be necessary to interrogate the mode.
- a function which is required to deal with asynchronous events should be designed to be asynchronous, i.e., artificial time constraints should not be forced
- artificial control constraints should not be imposed on a module (e.g., if ascent guidance can call a common routine, LAMBERT, before breaking down into n ascent guidance modes, it is better than having each guidance mode call LAMBERT)
- flexibility (developmental and real-time) is a major concern. It should not be compromised for efficiency (time and memory) unless it becomes an absolute necessity.

DESIGNS THAT SHOULD NOT BE INCLUDED IN THE REQUIREMENTS LEVEL

There are designs that should not go into the Requirements document, if it can be helped. They usually fall in the areas where an engineer can go too far in a direction that is incompatible with what the eventual specifications level designs should be. Examples of things to be avoided in the Requirements design stage are:

- incorporating factors such as constants which may later need to be changed, e.g., incorporating a fixed cycle time as part of the design rather than a parameter

- presenting the designs in a lower language level than HAL (e.g., indexing for a DO LOOP, or design for a matrix inverse routine when HAL has it in the language)

- incorporating efficiency considerations that will become obsolete as more is know about such things as the compiler, flight computer, etc. For example, a Requirements design should not concern itself with the number of multiplies or divides vs. an alternate table read-in method for the sake of efficiency, since the eventual compiler might take longer to read in a value than to multiply it (or vice-versa)

- requirements which are dependent on architectural* considerations. For example, from a Requirements point-of-view, a design should be valid:

  (1) whether or not it is a COMSUB or an internal PROCEDURE

  (2) whether or not data is explicitly carried from one function to another (parameter passing) or implicitly used (by means of common data within common scope)

  (3) whether or not a function is in-line, CALLed, SCHEDULEd, etc.

  This is not to say that the Requirements engineer should not include the next step of a design, i.e., architectural designs in addition to the Requirements, if they have been completed as a natural step in verifying designs, and

*Architecture design is the second major phase of program structuring. The second major phase worries about the modules from a language and computer(s) mapping point-of-view. This includes partitioning. The major concern of the first phase is to structure the functions so that they will work no matter what the architectural constraints are, i.e., the functional design is flexible. There could be exceptions, but they should be avoided whenever possible.

have followed Specifications rules. Architectural considerations are a major job at the Specifications level. However, many of the designs (e.g., a subset of GN&C design modules) will be verified in HAL and they will be using an architectural mapping that in some cases will resemble the final product. It would seem beneficial for those designs that have been architectually "mapped" out for design verification purposes to be submitted with the Requirements. The Specifications level still has the option, of course, of changing the architecture. This type of change might be labelled as a Specifications design change rather than a Requirements change unless it was crucial that performance relied on a unique architectural requirement.

- representing decision logic based on n design candidates (where the final outcome will be less than n) and where the decision logic is only there to show that the design is an either/or type of design. In this case, the Requirements should show A in a functional design where A is equal to A_1, or A_2, or A_3...
- If two or more point designs are intended to perform the same function for different users of a same function, only one of these designs should be chosen*. There should be no reason why one user should invoke a completely different design of a function than another user, if the function needed by both users is truly the same function. Here, efficiency is of concern to the Requirements integration effort; the Specifications level would not have the means of preventing this type of unnecessary redundancy, since the fact that more than one design performs the same function would not be obvious.

SPECIFICATIONS DESIGN CONSIDERATIONS

Specifications designs are the responsibility of one organization for Shuttle software. Since Requirements are submitted from several organizations, the Specification designers have a key role of integrating and standardizing these designs at the Specifications level. Major tasks to be performed by Specifications engineers include:

*See Reference 8 for methods of algorithm comparisons.

- learning the Requirements

- indicating faulty requirements designs (see page 13). In addition, if structured programming and program structuring concepts have not been applied to the Requirements, this should be done before the 'Specifications' Step is begun (see page 5).

- evolving Requirements into Specifications without altering functional relationships (this includes performance considerations) unless, of course, the Requirements were wrong.

- partitioning functional Requirements for different computers

- providing architectural mapping* within each computer. This includes: 1) determination of those functions which are to be COMSUBS, PROCEDURES, PROGRAMS, FUNCTIONS, etc.), 2) determination of data scope which includes that data which goes into the COMPOOL level, PROGRAM level, PROCEDURE levels, etc., 3) determination of real-time interfaces (relative priorities, cycle times, etc.), and 4) determination of asynchronous and synchronous logic.
- memory sizing considerations (this could include further refinement of common use of common code**, breaking up the program into modules dependent on memory read-in from tape), use of 'Macros', etc.

- determining compilable units of the program (specifically for development purposes)

- refinement of code dependent on timing efficiency considerations (only if necessary)

- standardizing modules and their components (includes naming conventions, methods for verification, etc.)

- enforcing programming conventions (e.g., preventing specifications from using in memory two integration routines, where only one is necessary

*See page 51 of Reference 1 for an example of architectural mapping.

**From a software reliability point-of-view, this is a tough problem at the specifications level. This subject will be covered in greater detail in a later memo.

- taking Requirements designs that have gone to the functional level (or pre-functional level) and evolving them into the detail of Specifications from the point-of-view of using HAL effectively, the computer, etc.

- determining new language/compiler features that bring about more reliable code (e.g., automatic checking features in the compilation or verification tools)

- integration of those "frozen" modules from the Systems and Applications libraries (Reference 1, page 6). For example, the Requirements level might show a use of A_1, or A_2 for function A. In this case the Requirements level would include two functional designs for A, A_1 and A_2. The final Specifications would include only one.

- design and integration of many system functions (specifically those close to the compiler) which might not be included at the Requirements level (e.g., self-test, additional error checking logic, uplink, downlink, etc.). These would include designs which were existent solely for the reason that unique features of the AP101 required them. If any code needed to be written in assembly language, designs of this code could fall into this category.

- verification of Specifications results against Requirements

- final verification of the software program with its interfaces (hardware, man-machine, etc.)

SUMMARY

If you are going to have a Requirements design phase, it makes more sense to do it right the first time, rather than saving it for a later phase. If a design is bad, then the problems encountered in implementing and testing are increased and the chances of obtaining a reliable flight software system are minimized.

In general, the responsibility of the Requirements is to concern itself with the first phase of program structuring (that is, the determination of valid functions). The responsibility of the Specifications level is to concern itself with phase 2 of program structuring (that is, the architectural mapping of the design). In some cases, the Requirements will not

complete the first phase. In some cases, it will go further. An important interface function between Requirements and Specification is knowing to what extent the Requirements have been defined and therefore relaying this information to the Specifications people.

Some designs (e.g., GN&C) may be more crucial than others (e.g., PMS) to apply proper integration design techniques to, for the performance of some functions are highly dependent on mission-critical events. Thus the proper integration of these designs at both the Requirements and Specifications levels are of high priority consideration.

If some Requirements designers used HAL effectively, and structured techniques using HAL notation as tools for design verification, these should be submitted with the Requirements. Why not stamp a higher level of approval on these designs as being further along in approaching the Specifications level than, say, some others. On the other hand, if designs do not meet the recommendations that are set forth for Requirements, these should be stamped accordingly.

The use of structured constructs with design diagrams is much easier to read, understand, etc. Thus the communication between Requirements and Specifications is much more reliable, as well as the designs themselves.

The program structuring determines the functions, and thus the resultant modules. These should be consistent from the Requirements phase throughout to the final flight software product.

REFERENCES

1. Hamilton, M., Zeldin, S., "Higher Order Software Requirements," CSDL - E-2793, August, 1973.

2. Hamilton, M., Zeldin, S., "Top-down, Bottow-up Structured Programming and Program Structuring," (Revision 1), CSDL - E-2728, December, 1972.

3. "Software System Design Guidelines, " IBM Electronics Systems Center, Owego, New York, Contract NAS-9-13548, September, 17, 1973.

4. JSC Internal Note No. 73-FS-1, "Shuttle Avionics Software System Assumptions and Goals," June 25, 1973.

5. Webster's Dictionary

6. "Space Shuttle Guidance, Navigation and Control Design Equations," JSC-04217.

7. Dijkstra, E.W., "Go To Statement Considered Harmful," Communications of the ACM, 11, 3, March, 1968, 147-148.

8. Zeldin, S., "An Example of Algorithm Comparison Using Higher Order Software Criteria," CSDL - P-017, December 6, 1973.