MCR-70-327

DEVELOPMENT OF A KSC TEST AND FLIGHT ENGINEERING ORIENTED COMPUTER LANGUAGE - PHASE I REPORT

C. W. Case
E. L. Kinney
J. Gyure

Martin Marietta Corporation
Denver Division
Box 179, Denver, Colorado 80201

August 1970

Interim Report for Period July-August 1970

Prepared for
National Aeronautics and Space Administration
John F. Kennedy Space Center

PREFACE

This report contains the results of the Phase I effort of the Development of a KSC Test and Flight Engineer Oriented Computer Language Study. This Phase I effort was directed at the examination of existing related languages and their applications.

Three languages, ATOLL, ATLAS and CLASP were examined in detail and their characteristics are documented in a language characteristics table. A general review of several other test oriented languages was undertaken. Information on ATOLL II, MOLTOL, CTL, VTL, TOOL, ADAP, and ASEP is included in this report. Some general comments on other higher order languages not related to testing is also included.

A description of the ATLAS compiler developed at the Martin Company is included in the appendix.

CONTENTS

I. INTRODUCTION

This report covers the Phase I study task of the Development of a KSC Test and Flight Engineer Oriented Computer Language.

This task included the study of numerous languages that might provide useful guidance and/or features for the development of the new language. In addition, the study of existing language applications have provided background and understanding of the role that a test-oriented language plays in the acceptance and implementation of automation. Other results of the Phase I study include a greater appreciation of the degree to which test system characteristics and limitations have resulted in the specialization of "test-oriented" languages, the degree to which test system characteristics have dictated test philosophy, and the evolution of test system oriented programmers from both test engineers and professional programmers.

The comparison matrix, Table 1, lists characteristics of three of the languages studied, and some particularly pertinent features of other languages. The CLASP language, while not test oriented, will most probably be a companion language used in the origination of flight computer programs for space vehicles, particularly in the areas of guidance and navigation. It would, therefore, seem desirable to promote commonality between CLASP and the new language wherever practical. More space program related test programs (in terms of implemented test statements) have been written in the ATOLL language than any other "test-oriented" language. It might, therefore, be expected to contribute to the new language. The ATLAS language is rapidly receiving world-wide acceptance for avionics and other unit testing, and many compilers are in various stages of design. ATLAS is particularly English-like, strictly test-procedure oriented, and test system independent. It might well contribute significantly to the new language.

Several of the other languages studied have commonalities with the above three languages and with each other.

The next two study Phases will select characteristics desired for the new language and prepare a language specification.

II. LANGUAGE DISCUSSIONS

This section contains brief descriptions and discussions of each of the languages studied. An attempt has been made to provide background information leading to the language definition, as well as some general conclusions relating to the impact that the language has had on the program(s) using it. Unfortunately, there was not time to study all test languages in use throughout the country. It is believed, however, that many features of the languages studied were based on studies of still other languages; as a result, these languages actually represent a much larger basis for study than might be immediately apparent.

II. A. ATOLL

Acceptance, Test, Or Launch Language (ATOLL) was developed early in the Saturn development of automatic checkout. Initial RCA-110 computer program development highlighted the necessity for a language which would provide a more direct path from the test engineer, writing test procedures, and the computer object program which would provide for the conducting of Saturn tests. A specification for such a language was released in 1963.

The objectives of the language were to:

1) provide a language to be used in testing the Saturn stages completely independent of the checkout equipment or location;

2) provide a language which would enable a test oriented engineer to write the test programs necessary to test his subsystem without a large training effort;

3) provide a language whose resulting test programs would satisfy other documentation needs such as (a) detailed test procedure, (b) serve as a test program for input to the automatic checkout and launch test system, (c) provide for test review and evaluation by project and quality personnel, (d) provide a checklist for verifying contractor test performance;

4) provide a language format which is easy to learn and close enough to test nomenclature to be easily taught and used;

5) provide flexibility of expressing simple test functions and grouping of these into more complex routines describing a test procedure;

6) provide for ease of making changes.

ATOLL falls short of meeting all of the objectives originally proposed. The use of mnemonics and abbreviations is common. The majority of abbreviations are logical and easily remembered. However, many of these have variations in the statement format which cause them to lose their identify. Some abbreviations are programmer oriented and not test engineer vernacular. Others are strictly checkout equipment oriented and lose their ease of remembering on the part of the systems test engineer. Test article functions have terms applied which must be referred to in a table for understanding. Comment cards are used to provide understanding but are not interpreted by the ATOLL compiler and thus do not insure that the action requested is identical to the comment message. Had the ATOLL met all objectives proposed, the test statement would have included the comment information and would have been interpreted by the compiler, resulting in a one-for-one relationship.

II. A. ATOLL (Continued)

All programmer oriented terms and checkout specific terms would have been replaced with terms familiar to the test engineer. Examples of programmer oriented operators are SETX, INCX, SFLG, TFLG, MSFG, MTSG, SFIT, CODE, BLOK, EXEC, BEGN, RETN, MLSR, CALL DEOL, EXEM, EWD, GOTO, PROB, PROC, PREM.

Examples of test equipment oriented terms are DISA, DISO, SSEL, MDSO, MNTR, TERM, DMON, DPLY, DFYM, SEMI, DFLG, RGMT and RCDC.

Examples of test oriented terms which imply functions which a test engineer may accept in lieu of more familiar tems are RECD, READ, DELY, TEST, and SCAN.

The increased number of ATOLL programs over the last five years indicates a willingness to attempt automatic testing. This can partially be credited to ATOLL by providing a tool which more adequately provides an interface between the test engineer and the checkout equipment. The number of switch positions on the subsystem consoles implies that manual capability and desire preceded the present use of automatic checkout. ATOLL has provided a capability for test engineers to communicate with the program. Training of test engineers to the level of competence necessary has been quoted in the neighborhood of 40 hours instruction and a few months of on-the-job applying the language to be fully competent.

As pointed out in other reports, a shortcoming of ATOLL is lack of provision for arithmetic operations. This has resulted in the need for many machine language programs and subroutines. All readability is lost whenever the machine language programs are executed. This factor violates the objectives originally established for ATOLL toward providing ease of verifying contractor test procedures, test review, and quality acceptance in addition to providing a capability for the test engineer to write his own test procedures. A programmer must communicate with the test engineer to insure his procedures are being implemented correctly into machine language test programs.

From the initiation of the early ATOLL programs until the present time (5 years?) a gradual cross pollination has taken place between test engineers and professional programmers. The test engineers have, through training and use, come to accept ATOLL, and the increasing number of programs utilized today, attests to its acceptance. On the other hand, programmers assigned to the Saturn program have become familiar with the checkout equipment and the Saturn vehicle requirements. The programmer side of the house feels that indeed the ATOLL is an English-readable test oriented language. The test engineers feel that while it is useful, the terms employed and the formats are not test oriented; that to fully understand what is going on requires more study than they are willing or able to commit in addition to looking up too many checkout equipment/vehicle terms to verify that the correct vehicle function has been addressed. In addition, more continuous monitoring capability and an arithmetic capability is desired.

II. B. ATLAS

The Abbreviated Test Language for Avionics Systems (ATLAS) was originated and is maintained by a fluctuating group of knowledgeable personnel from many organizations, including airlines engineering and maintenance organizations, software design organizations, avionics manufacturers, aircraft manufacturers, and automatic test equipment manufacturers. Both domestic and foreign firms are represented. The military services have also participated. The sponsor of the group is the ATE Subcommittee of the Airlines Electronic Engineering Committee. AEEC performs the general function of generating and promoting standardization of avionics and related systems for use by commercial airlines. The ATE Subcommittee's charter is to promote standardization in the area of Automatic Test Equipment.

An early conclusion of the subcommittee was that the major source of automation difficulty was the lack of precise, test equipment independent test procedures that could be used as a basis for generating automatic test equipment programs. Consequently, the subcommittee directed its full attention to the specification of ATLAS, with the intent that all future overhaul and test manuals for avionics procured by the airlines must contain test procedures written in the ATLAS language. The ATLAS specification was released June 1, 1969, [9, 12] and the implementation of the language is well underway. Since current avionics units used by airlines do not have vendor supplied ATLAS test procedures, the ATE manufacturers and airlines using ATE are at present the major writers of ATLAS Test Programs. Eventually, they will become the readers and users of the vendor prepared test programs.

ATLAS is by far the most readable, English-like, test-oriented language of all the languages studied. It purposely attempts to avoid any test system or test equipment orientation so that the airlines might exercise greater lattitude in selecting and procuring their ATE. The language concentrates on the specification, checking and timing of interfaces with the Unit Under Test (UUT). Some general characteristics of the ATE are assumed, however, as illustrated by PRINT, DISPLAY, RECORD and other statements. Test system capability to provide all specified interfaces is also assumed.

In view of the relatively short time since the ATLAS specification was released, it has received remarkable acceptance, most probably due to its readability. Jacobi Systems Corp. profitably conducts training seminars several times a year. The British Ministry of Technology has endorsed ATLAS for technical school courses. The Navy has contracted for a compiler for an ATLAS-derivitive for use with the VAST system. Other non-airline applications are increasing. The ATE Subcommittee and many of its members are swamped with requests for changes, additions, and clarifications from diverse sources.

Response to ATLAS is not all positive, particularly for non-airline applications. Some necessary specification changes approved by the subcommittee over a year ago have not yet been released. ATLAS compilers are expensive and require considerable computer memory. Interfaces between an ATE operator or test conductor and the ATE cannot easily be defined because the language is independent of the ATE (most ATLAS-based variations are a result of this characteristic). The general philosophy of avoiding abbreviations is disagreeable

to test writers, particularly with primitives such as MEASUREMENT, IMPEDANCE, and TRIANGULAR WAVE SIGNAL. Due to its UUT interface orientation, it is not readily adaptable when the test system or UUT interfaces are not totally under ATLAS program control (as might be the case with an intervening data acquisition system). Provisions for non-UUT interfaces are non-existant.

In general, ATLAS may indeed provide a basis for a new aerospace systems test-oriented language, but not without major modification.

II. C. CLASP

The Computer Language for Aeronautics and Space Programming (CLASP) developed as a result of a study undertaken for the NASA Electronic Research Center by Logicon, Inc.[2] This study compared the Space Programming Language (SPL) developed as a result of a study carried out by System Development Corporation [3,4] and PL/1 for use in the development of real-time aerospace programs for NASA.

As a result of this study, it was determined that a number of deficiencies existed in both SPL and PL/1 with regard to the application NASA specified. A new language, CLASP, was proposed which was based on SPL. Further development of SPL has resulted in making CLASP a subset of the more general SPL. CLASP is now identical with the SPL/MARK I subset.

SPL and its subsets, including CLASP, have been designed specifically to assist in the writing of programs in the aerospace field. The software requirements of this application area are at present somewhat restrictive due to the nature of present day aerospace computers. However, future software requirements are tending toward increased complexity and sophistication as new hardware with more capabilities becomes available. The requirement of a higher order language for this application area arises because of these developments.

CLASP was designed using ground rules which require the language to be of immediate and practical use with aerosapce computers of the present and near future. This requirement resulted in a language oriented to arithmetic and logical manipulations with primary emphasis on the solution of guidance and navigation problems.

CLASP is a procedure-oriented language for use by a professional programmer. Due to the many computer dependent features included in the language, a detailed knowledge of the target computer is required.

CLASP has several of what are considered by its designers to be unique features with respect to other higher order languages. Extensive fixed point capabilities with respect to scaling control are provided. A capability exists for scaling control over intermediate results of arithmetic operations. Temporary variables can be declared which take on changing attributes depending on usage. Optimization compiler directives exist which provide modifiable degrees of optimization with respect to both space usage and execution time.

CLASP compilers are currently under development and the language is still evolving. The CASLP language, as presently defined, is not suitable for direct use in a test-oriented environment.

II. D. ATOLL - II

ATOLL-II was developed under contract by General Electric Company as an alternative or replacement for ATOLL for use in Saturn checkout. It was never implemented, but a reference manual 13 and compiler were prepared.

The language is based on Fortran with the addition of real-time test-oriented and Saturn ESE oriented statements. It is more English-like than ATOLL with terms such as TURN ON, TURN OFF, WAIT and a few others. Some other statements and terms are more specifically programmer-oriented (DO, PROC, INCORP, DECL). It is not particularly test-engineer oriented in much of its vocabulary. It does include capability to initiate, synchronize and terminate parallel programs, including continuous monitoring, either in a single or dual computer configuration. It also includes arithmetic capabilities that were not implemented in ATOLL.

Declaration and assignment capabilities are extensive and would require a significant amount of training for most test engineers.

The format of statements includes fixed card column fields for Control, Time, Label, and Statement; however, these may be reassigned at any place in the program by a Card Format declarative.

A general observation is that ATOLL-II includes many capabilities which would be seldom used, but would require significant user training and retraining for continued proficiency by engineers whose principle function is not programming.

II. E. MOLTOL

MOLTOL, Manned Orbiting Laboratory Test Oriented Language, was designed under contract to the Air Force for use in the checkout of the Manned Orbiting Laboratory. It was never implemented due to MOL program cancellation. A MOLTOL Test Writer's Reference Manual, [11] was prepared, and has some areas of incompleteness. MOLTOL almost totally includes ATOLL-II and the discussion of ATOLL-II is applicable. A few significant additions are discussed herein.

All test statements are assumed to be executed with a fixed time interval spacing. Language statements are included to control these execution intervals.

Communications with an on-board computer are accommodated with a DISPATCH statement.

Definition of alternate terms which can be substituted for language primitives is allowed with a statement such as:

DEFINE END AS FINISH

or

DEFINE MONITOR AS NONSENSE

(This capability has obvious ramifications!)

II. F. CTL

The Computer Aerospace Ground Equipment Test Language (CTL) was developed to provide a near English test oriented language for use with the Titan IIIC/MOL checkout equipment. The development of this language emphasized the role of the test engineer in expressing commands and criteria to the test article completely, simply and unambiguously. The role of the customer, reviewer and user was considered and made it imperative that the format be readily understood by all.

Terms were selected which supposedly were familiar to the test engineer and which would be readily translated to machine instructions. Diagnostics played an important part in providing safeguards against illegal or hazardous operations while not restricting the test engineer's flexibility.

A second objective of the CTL was to reduce the writing burden of the test engineer by providing sequence preparation aids. These aids enabled the writer to abbreviate his tests in such a form that the translator could expand the statements into fully defined lists that could be easily understood and reviewed by all interested parties. The CTL/Translator ratio amounted to a 20/1 expansion for a typical test sequence. Each CTL statement required about four computer words after translation which provided a reduction in writing by the test engineer of one CTL statement to 80 machine language statements.

One set of resident programs are used for all test sequences, variations being accomplished by the order of the test language statements and differences in the statement modifiers. With this complete set of programs new or modified sequences requires no additional programming. The sequence which is described by a string of data, is a subset of instructions plus data that would be required for an assembly language or compiler description of the test. The translator provides for efficient packing and conversion of the language statements which permits a reduction of 1/3 to 1/10 in memory space requirements.

The importance of an interpretive language is summarized below:

1) test techniques and requirements are not generally understood by assembly language programmers;

2) test engineers familiar with vehicle test requirements can prepare sequences in near-English format;

3) customer and contractor quality control personnel can easily understand and validate the sequences;

4) changes to any sequence can be accomplished by an engineer by stating the change in test language;

5) field modifications are easily accomplished;

6) the writing burden has been reduced almost one hundred fold;

7) on-line use of CTL enables generation and execution of tests on-site with near zero delay.

CTL consists of 20 basic elements or verbs and their associated modifiers. Elements can be roughly divided into four classes:

1) Discrete signal observation and control - APPLY, RESET, and CHECK/DISCRETE.

2) Analog signal observation and control - CONNECT, STIMULATE, MEASURE, and CHECK/ANALOG.

3) Test flow control - BEGIN, END, TIME, SET, DEFER, and CONTINUE.

4) Special - DISPLAY, SAVE, and RESTORE.

Several elements are provided as tools for the test engineer. These tools provide a capability which eliminates repetition of recurrent test sequences and detailed descriptions of frequently used groups of data.

These elements, designed to facilitate test sequence preparation, include:

1) SEQUENCE;
2) REPEAT;
3) REP/TEST;
4) SYSTEM/TEST

SEQUENCE is used in on-line control. It makes use of established library test sequences within any other sequence. REPEAT, REP/TEST, and SYSTEM/TEST are facilities of the translator whereby repetitious data or elements are inserted into the control file using these elements as a mode.

The basic building block of CTL is the test element. Each element has a set of modifiers associated with it that amplify the element for a specific operation.

Single Test groups elements together to satisfy some unique control response, or display function to make up a single test. One to several dozen test elements can be included in a single test which is assigned a unique number.

Repetitive Tests are tests which are repeated frequently with slight parameter variations. The repetitive test allows the test writer to establish the order and number of test elements required and by simply varying the numeric value assigned to the parameter list and calling out the unique repetitive test name, repeat the test without restating each test element. The repetitive test name is five alphanumeric characters, and the parameter list can contain up to 25 variables.

II. F. CTL (Continued)

The next level is called a sequence which consists of a number of Single Tests and/or Rep Tests. Two types of sequences exist: Library Test Sequence and Test Sequence. A Library Test Sequence is a special version of a Test Sequence. A Library Test Sequence is used whenever a function (such as apply vehicle power) is used in many other test sequences. A Library Test Sequence can be called singularly from the test console or by any currently executed sequence. The Library Test Sequence will have a number of the same form as a Test Sequence but will be uniquely identified as a Library Test Sequence.

A Test Sequence is a collection of Single Tests, Repetitive Test, and Library Test Sequences arranged in a logical order to accomplish a specific testing or control function. Each Test Sequence can be called from the test engineers console. Any Test Sequence may call other Test Sequences.

CTL fails to meet the ideal objectives which were established at the onset of CTL development from the standpoint of readability and the use of terms familiar to the test engineer. As in the case of ATOLL, the majority of terms are programmer oriented and test equipment oriented rather than test engineer oriented. The use of comment cards helps to fill in the missing links, but this information is not interpreted. Therefore, there is no assurance that the comment card and the CTL statements coincide. The elements may be spelled out or abbreviated which supports readability providing the meaning of all elements and modifiers are understood. In the application and measurement of discrete and analog signals, readability is sacrificed by attaching an alphanumeric label to each stimulus and measurement signal. Rather than stating: "TURN FCC POWER ON", the statement would read "APPLY 1D747". In ATOLL this statement would read "DISØ1 MDO, 1823". The readability is less than desired in either case. Of all terms used in the implementation of program generation, the majority fall into test equipment signal interface names which are identified by an alphanumeric number. While it is true, many years of familiarity will enable the test engineer to identify a signal name and function with its assigned number, he must still resort to tables to determine the name and function of many. The individual reviewing, verifying or approving must resort to a table of terms almost entirely, which increases the time span of his task and affects the desirability of the job.

As the Titan III-C/MOL Program was canceled before the full utilization of CTL could be realized, it is difficult to assess the degree of accepting automatic testing directly related to having a TOL. However, the philosophy was such that automatic testing was going to be a way of life and with this ground rule at the onset of the program, everyone related to the task accepted it. However, experience with a previous program was accepted to the point that the need and use of a TOL on the CAGE program was a foregone conclusion prior to initiating any design effort. The time and effort expanded upon the CTL was so generally accepted that the need and requirements were included in the VIKING SYSTEMS TEST EQUIPMENT as soon as trade studies resulted in the decision to include a computer in the ground test system. The availability and experience with a TOL has certainly increased the general acceptance of automatic testing at the Martin Denver Division.

II. F. CTL (Continued)

As is true for ATOLL, a shortcoming of CTL is the lack of provisions for arithmetic operations. There is also no provision in CTL for calling machine language programs. Both of these capabilities might have been required for guidance checkout. The guidance system for all Titan vehicles has been the responsibility of the guidance associate contractor as well as the provisions for its acceptance and testing. This has resulted in the guidance contractor supplying the necessary equipment for guidance checkout, and no provisions are made in CTL. All other systems were accommodated by CTL.

The software group for the CAGE test program consisted of design engineers, professional batch operations programmers, and real-time programmers. During the course of the design, cross pollination of engineers and programmers did occur. For the most part, engineers became more programmer oriented, a few of them slipping over to the programming side of the house. Several of the programmers professed a desire to remain in real-time programming effort and were assigned to the Engineering Department. The selection of specific elements resulted from decisions by design engineers rather than test engineers. For this reason, and as a result of cross pollination, it is felt by those involved in CTL development that it is indeed a near-English test engineer oriented test language.

II. G. VTL

The Viking Test Language (VTL) was designed to serve as the communication medium between the test engineer and the Viking System Test Equipment (STE). Since many organizations and individuals are included in the writing, reviewing, and implementing of test procedures, it is necessary that the VTL be designed as a conversational english language requiring a minimum of training for use. The majority of these people are not programmer oriented or professional programmers.

A prime objective of the VTL was to simplify the test engineer's burden by including several shortcut features into the language and Viking data files. Symbolic data symbols, group numbers, and system repetitive tests are available to reduce the data input requirements.

The test sequences, written in test language, are translated by the off-line software system.

The test language translator processes input written in the Viking test language. The test sequences, library tests, and referenced data is read from the appropriate files and processed. The output of this process is a machine oriented sequence for the on-line system and an English language tabulation of the composite sequence. The machine oriented sequence may be placed in core memory, magnetic tape or on the rapid access disk file.

As the language evolved, the specific test equipment configuration was unknown. It was assumed that this equipment would consist of a digital computer with core memory, magnetic tape, and disk memory. The language was designed specifically to meet the requirements of the Unit Under Test, in this instance, the Viking Lander Capsule.

The language structure is based upon building test sequences consisting of test blocks, system repetitive tests and library tests arranged in a logical order to accomplish a specific test or control function. Any test sequence may be called by the operation from the test console. Any test sequence may call other sequences.

The lowest level is the test element. Each element performs a specific purpose such as commanding a relay closure, establishing a stimulus, delay, etc. Modifiers are associated with test elements which amplify the element to a specific operation.

The second level, referred to as Test blocks or System Repetitive Tests consists of logically assembled test elements which satisfy a unique control, response, or display function. The number of test elements in a test block may vary from 1 to several dozen. System Repeat Tests (SRT) are also second level in the language organization. These are used to define often repeated operations. They provide a shorthand which allows the test writer to use a predefined set of elements by only specifying the SRT name and listing the parameter table. SRT's are defined on the system level and are entered into the data files so that they may be used by all sequence writers.

II. G. VTL (Continued)

The next higher level of the language organization is the sequence level which is constructed from test blocks and/or SRT's. Two types of sequences are provided: Library Test Sequences and Test Sequences.

Library Test Sequences consist of frequently used functions such as "Apply Lander Power." Within such a sequence, all required prerequisites and control/response necessary statements to satisfy the function are contained. A library test sequence can be called from the test console or by any currently executed sequence. The library test sequence has an identifier of the same form as a test sequence.

A Test Sequence is a collection of test blocks, system repetitive tests, and library test sequences arranged in a logical order to accomplish a specific testing or control function. Each test sequence can be called from the test console or by currently executed sequences.

The objectives of VTL and much of the conceptual design incorporated ideas from its predecessor, CAGE Test Language. Fewer elements are included with several of them being unique to the Viking test article. The shortcomings of the ATOLL and CTL are also found in this language with the exception of a capability for communicating with the onboard digital computers. A test element named "LINK" provides a capability for communicating with the Guidance Control Computer GCC and the Command Control and Sequencer (CCS) memory. Run and halt commands, as well as load and verify single data words or blocks of data words or instructions. Data words can be read out of the GCC or CCS memory or registers for verification, analysis and display.

Example: LINK WC017 LOAD M0134

would cause data block WC017 to be loaded into the GCC memory starting at location M0134. Data blocks WC017 would have been previously defined in the data files.

LINK WC017 VERIFY FLAG M0134 would cause the block of data words defined by WC017 to be read out of the GCC memory starting at location M0134, transmitted downlink to the Viking Systems Test Equipment (VSTE), compared against the WC017 data block that was loaded in the previous example, and a flag to set for each word that does not compare properly. If all words compare, no flags will be set.

LINK RUN will cause a command to be sent to the onboard computer or sequencer to initiate onboard operation of their function.

The format and selection of operators or elements are definitely more computer and STE oriented than they are test engineer oriented. It is true that the test engineers' writing effort is greatly reduced but at the sacrifice of readability and useability by organizations other than the originating organization. It would be difficult to get widespread test procedure acceptance on the basis of VTL programs.

II. G. VTL (Continued)

VTL as ATOLL and CTL fails to provide an arithmetic capability which appears to be the result of incorporating requirements of the specific test article and system test equipment--in this case, the Viking Lander Capsule.

In the case of Viking, time, schedule, and manpower have affected the test philosophy and the test language. Having developed two previous test languages for two different programs, sophistication based on prior experience was certainly possible. However, the constraints previously mentioned in conjunction with program requirements provided guidelines which dictated to some extent the nature of the test language to be provided.

II. H. TOOL

The Onboard Checkout System's (OCS) Test Oriented Onboard Language (TOOL) is the culmination of a development cycle beginning in 1965 and extending into 1970. This effort was undertaken by the Denver Division of Martin Marietta for the NASA Manned Spacecraft Center. The purpose of this effort was to develop an independent, real-time, computerized system for verification and monitoring of experimental and developmental subsystems for various space vehicles.

In 1966 an OCS breadboard, demonstrating the feasibility of the OCS concepts, was delivered to NASA.[5] An initial version of TOOL was provided at that time. Development was continued with the construction of a prototype unit called the digital test set, delivered in 1967.[6] Another version of TOOL was provided at that time. Development continued with the resulting delivery of a flight packaged OCS with the present version of TOOL in 1970.[7]

TOOL[8] is part of an on-line interactive multiprogrammed system which enables a test engineer to create tests which utilize the OCS hardware. The system is self teaching through the use of extensive cuing techniques. A test engineer can write new tests, review tests previously written, and modify in considerable detail previously written tests.

Translation of tests written in TOOL results in data list entries which are then stored in computer memory for later execution. An interpreter passes entries from the data list to the appropriate routines for execution of the desired test. This differs from the compilation approach in that a compiler translates input source statements into the machine code of the target computer. Translation of input source code into data list entries results in a considerable core savings for the storage of tests.

The TOOL system provides for the concurrent execution of a multiple number of tests along with the asynchronous processing of hardware interrupts. Tests can also be executed while monitoring operations are active. Tests can be written, reviewed, and modified while other tests and monitors are operating. Priorities can be assigned to the execution of tests. Allocation of the limited hardware and software resources available is a function of the TOOL system.

A password can be attached to a test to allow only authorized personnel to have access to that test. Protection keys can be set which prevent improper operation or alteration of a test.

TOOL is an English-like language making extensive use of readable abbreviations as modifiers to language primitives. It is designed as a special purpose, application oriented language. Input is in a fixed format. Simple data variables are provided along with a limited expression solving capability. Language primitives are OCS hardware and system oriented. FORTRAN like unconditional branching, conditional branching, and looping are provided.

II. I. ADAP

The Adaptive Intercommunication Routine (ADAP) also referred to as the Block II ACE system was developed primarily to enable the automation of many test sequences. In pre-Block II testing the basic framework of the software was the same as is currently used with ADAP. The test requirements using ADAP for implementation require the same formalized input decks that has always characterized programming requirements submittal by ACE users.

ADAP provides for storing programs on magnetic tape and the calling of automated sequences from the test console. Portions of the Test File Tapes (TFT) are loaded from tape and stored in core: Systems Monitor, U/L and D/L Control, and ADAP control (subprogram). Other programs are loaded by the Monitor and executed as requested by the test engineer from his console.

The automatic operations desired are stored on tape in groups. Each group consists of one or more Intercommunications. Such Intercommunication consists of one or more sequences for the computer to execute. The sequences simulate data entry normally performed by a test engineer's execution of an R-START or C-START switch at his console. A sequence may include command generation, transmission, and monitoring of downlink responses before a subsequent operation can take place.

Requirements for test programs are originated by contractor systems engineers. The contractor then translates these requirements into formal subprogram specifications fully describing all parametric requirements and operations of each subprogram. These specifications are then submitted by the contractor to the computer programming group after NASA approval. Professional programmers write the subroutines and perform the mechanics necessary to insert the subroutine into the Test File Tape.

The parameter cards submitted to the programming group by the contractor are unique to ACE and are not intended to provide readability. The program was not designed to aid the test writer or to improve readability by reviewers. A programming input scheme had been in operation which utilized the ACE-S/C in a manual mode. The addition of ADAP enabled a more closed-loop automatic mode to be implemented.

Program Requirements Processing Specification (PRPS) cards are prepared for both Uplink and Downlink equipment utilization and interface requirements. In addition, on Operational Checkout Procedure (OCP) must be prepared. The OCP is a sequential description of the order of START executions, START switch settings, and the expected response on the various control consoles. A Test Engineer Test Oriented Language would be expected to accomplish both of these tasks within a much shorter time span and improved acceptance routines.

ADAP has been responsible for implementation of Automatic test programs. In so far as the degree of automation currently experienced with ACE-S/C and the acceptance of automation attributed to ADAP, a need for more automation was pushed by one of the contractors and ADAP was proposed as a means to accomplish this end. The need involved time: performing one manual ACE Test consumed 2 hours; when implemented with ADAP, the test time was reduced to fifteen minutes. In this regard, ADAP has attributed to more general acceptance of automation by ACE users.

II. I. ADAP (Continued)

The test system characteristics definitely affected the implementation of ADAP as similar procedures for implementing tests are carried on with ADAP test program inputs. The formerly manual mode of test initiation was incorporated into the automatic mode programs by simulation of START switch execution.

Test engineer and programmer interfaces are rigidly controlled by the documentation requirements initially implemented. As a result of this formal organization, test engineers and programmers for the most part are isolated from the cross pollination process.

II. J. ASEP

Automatic Sequence Execution and Processor (ASEP) has been developed for use in ATM checkout utilizing existing ACE-S/C equipment and programming. ASEP provides the capability for parametric controlled sequences of commands and displays and also assures maximum core utilization through packing or relocatable, relative addressed sequences. Whereas the ADAP test sequences were located on magnetic tape and called by test engineer or executed sequence, ACEP programs are resident in core.

Test engineer interaction with the equipment and the program execution is almost identical to ADAP. The C-START addresses are implemented in the same manner and with the same codes indicating the ACE-S/C Operating System and Monitor will probably be used for SKYLAB A programs.

Automatic routines will simulate R-START and C-START execution.

Programming requirements will be input to the programmer group in the same manner that ACE-S/K programs have been implemented in the past with the use of Uplink and Downlink PRPS cards. One significant difference will be in the specification of limits--these will not appear on the Downlink PRPS cards.

ASEP is intended as a test language. As such, the test language input may override the necessity for Operational Checkout Procedures and enable the PRPS and test language inputs to provide the machine language object program. Sixteen elements or operators are utilized. Many of these elements such as ADD, GOTO, LEGAL, SET are computer/programmer oriented. ADD provides for incrementing internal computer counters. SET can affect a counter, flag bit, or event to a specifiedstate. LEGAL establishes the C-STARTS that can be used to activate a particular routine. While GOTO is programmer oriented, in areas where cross-pollination between programmers and test engineers can take place, test engineers accept this term as well as such terms as BEGIN, END, EXECUTE, INTERRUPT, and IF. Additional elements provided by ASEP include AUTO ONLY, DISPLAY STATUS, DELAY, DISPLAY (40 characters, or only some characters--leave rest unchanged, or display contents of address counter) MEASURE, REMARK, and STOP. ASEP provides a capability for the test engineer to change limits via card reader input. An ASEP routine must be inactive while changes are being made.

As ASEP has not been used on a program, its affect on acceptance of automatic test sequences is an unknown. However, the availability of ASEP for the SKYLAB program will undoubtedly insure that automation of test sequences will be utilized to a considerable extent. The SKYLAB program will be Huntsville's initial use of the ACE-S/C and the preparation of test input requirements caused some alarm early in the program. The development of ASEP was initiated no doubt to simplify the test requirements for the test writer and to encourage the utilization of the digital computer in its natural role.

II. J. ASEP (Continued)

The test system characteristics (in this case, ACE-S/C) very definitely have resulted in specialized test language. The simulation of R & C START switches, the method of handling parameter changes, and the display characteristics are the most obvious characteristics which are accommodated in the language.

Due to the ACE-S/C Programming Requirements Organization, it is felt that test engineers and programmers will continue to interface through rather formal documents which will inhibit any cross-pollination. Each function is rigidly bounded and buffered. Each will continue to perform his tasks through clearly defined channels and process forms.

The objectives of this test language appear to have been satisfied. At the outset, a method for closed loop automatic checkout maximizing core usage was desired. These appear to be provided. The test article required no more and the checkout equipment is capable of meeting these requirements. Comparison of this language with those previously discussed indicates the same shortcomings of the majority in the form of readability, completeness, arithmetic operators, and in the provision of a document which serves the requirements of test engineer, quality control; project verification, and management review.

II. K. OTHER LANGUAGES

The languages previously discussed are, with the exception of CLASP, oriented to the programming of test and checkout applications. They are special purpose languages defined for use in a particular problem area. Many other higher-level programming languages exist, some for special purpose applications and others designed for more general usage. It is relevant to this study to give some consideration to these other languages, at least to be aware that the problems encountered in creating a test and flight engineer oriented computer language do not end with the specification of that language.

Significant questions remain to be answered with respect to the actual development of a compiler and operating system for that language. One of the most basic of these questions is in regard to the choice of a language for use in developing the system to support the problem oriented language itself. Considerations relevant to this choice are of a completely different nature than those governing the design of a problem oriented language. Questions arise with respect to the best utilization of professional programming talent to create the necessary supporting systems in a timely and efficient manner. After these systems are developed, the further question of proper long-term maintenance arises.

The utility of higher-level languages for application to specific problem areas is recognized. The question then becomes one of whether such languages are available to write compilers and executive systems. At least two such languages, both reasonably widely known, are available. A small number of lesser known languages are available specifically aimed at compiler writing. One language, currently under development, shows promise of being a very good choice for both compiler writing and executive system development.

The discussion to follow will give an overview of two multipurpose languages, JOVIAL and PL/1. Both are languages that have proven themselves in the applications under consideration as well as providing capabilities in many other areas.

JOVIAL - Jule's Own Version of the International Algebraic Language was developed by the System Development Corporation and the first compiler was operational in 1960. The language has seen extensive military use and a version of JOVIAL is a standard programming language for Air Force Command and Control Applications. The Command and Control Application requirements necessitated a powerful language combining a balanced set of numerical scientific calculation capabilities and data- handling capabilities.

The basic objective of JOVIAL is to provide a procedure-oriented language for the use of professional programmers in solving large complex information processing problems. The language is considered to have fulfilled its objectives but has not been used extensively outside of military command and control applications.

JOVIAL has been successful in techniques whereby JOVIAL compilers have been written using the JOVIAL language itself.

PL/1 - This designation is the name of the language and is not an acronym. PL/1 was developed by IBM and the first compiler went into operation in 1966. The language was designed to be applicable to all those areas previously covered by FORTRAN (a scientifically oriented language), COBOL (a business-oriented language), and JOVIAL.

PL/1 is a procedure-oriented language for which almost all concepts relating to that type of language have been implemented. It is designed for the use of professional programmers.

The language has been successfully used for writing compilers and operating systems.

Another language which merits consideration for the development of compilers and executive systems is the Space Programming Language (SPL). SPL is an outgrowth of studies undertaken by Systems Development Corporation regarding spaceborne software. JOVIAL and PL/1 were considered the best available languages for the entire aerospace application area. Neither of these languages met all requirements, however. As a result, a new language, SPL, was defined. This language is an extension of JOVIAL.

The CLASP language, one object of this study, is a subset of SPL. Study of the capabilities of CLASP will perhaps be indicative of the power of SPL. The complete language, designated SPL/MARK IV, is expected to be specified by the end of 1970 and will contain features usable on aerospace problems and real-time programming tasks. Due to its nature as an extension of JOVIAL, compiler writing capability should be fully implemented in the language.

The language is being developed as a standard language for space applications and, as such, will probably be involved in NASA future programming efforts. SPL merits attention as development progresses to determine if stated goals can be met.

III. LANGUAGE CHARACTERISTICS COMPARISON

The form and content of the comparison table on the following pages is fashioned somewhat after the characteristics identified by Sammet.[1] An attempt has been made to cover the general and non-test-oriented characteristics at the beginning and the more technical and test-oriented characteristics toward the end.

Many semantic and conceptual definition problems exist between programmers and engineers and between engineers with different backgrounds. A glossary of definitions has been included in the hope of alleviating some of these difficulties.

TABLE 1. LANGUAGE COMPARISON TABLE

Characteristic	Purpose 1
ATOLL Acceptance Test Or Launch Language	Provides a test-oriented language for use in automatic checkout and launch of Saturn stages and vehicles.
ATLAS Abbreviated Test Language Avionics Systems	Test oriented, test equipment independent, English-like language for use by test engineers and technicians to document test procedures for shop tests of commercial airlines avionics units and to program general prupose Automatic Test Equipment. Eventually to be written by avionics suppliers and used by airlines.
CLASP Computer Language for Aeronautics and Space Programming	Procedure-oriented language for use by professional programmers in the development of real-time aerospace programs for aerospace computers: emphasis on guidance and navigation.
Other Languages	

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Language Responsibility 2	Who has Implemented the Language 3
ATOLL This language was developed by IBM for the NASA-MSFC to be used specifically with the RCA-110A computer checkout facilities.	SATURN contractors, (Boeing, NAA, DAC, & IBM) for Stages S-IC, SII, SIVB and IU utilize the language.
ATLAS ATE Subcommittee of Airlines Electronics Engineering Committee originated and maintains. ARINC publishes documentation.	Numerous ATE manufacturers for the airlines, including MMC, Bendix, Collins, Hawker-Siddeley Dynamics, Sperry, etc. frequently implemented in a modified form.
CLASP Originated by Logicon, Inc. for NASA and maintained by UDIG (Users, Designers, and Implementers Group)	Currently under development.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Documentation 4	Naturalness of Statement Structure 5
ATOLL Specifications of the Operating System for the Saturn V Launch Computer Complex, Vol. II, IBM No. 66-232-0001, NASA, MSFC, Appendix A provides the necessary information for using and documenting ATOLL test programs.	Statement structure lacks readability due to use of abbreviations, specific nomenclature of interfaces, and measurement terms. Comment cards provide understanding of program intent, but as comment cards are not interpreted by compiler, comment and statement may not coincide.
ATLAS ARINC Specification 416-1, ARINC Report 418	Statements start with flag and step number, then verb and other easily understood fields and/or phrases which are generally self-identifying.
CLASP Report "Flight Computer and Language Processor Study" Logicon, Inc. NASA CR-1520. UDIG is currently developing a language standard.	Straight-forward to programmers experienced with higher-level languages.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Self-Extension Capability 6	Consistency of Rules 7
ATOLL No self-extension capability other than the capability for naming a subroutine or subprogram which will be executed by an EXEC operation.	The primary example of rules inconsistency is related to the operator's DSFG, RECD, and RCDC in regard to the condition field. For the most part, (in relation to display), a "1" in the condition field clears the display before presenting the new message. In the above operators, the condition field is 1, 0, blank, or C and CLR must be placed in the variable field to clear the display.
ATLAS No provisions for new primitives to be defined.	Very consistent rules.
CLASP Function defining capability only.	Consistent between types of statements.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Self-documenting capability 8	User Program Maintenance 9
ATOLL ATOLL has self-documenting capability through the use of comment cards (Remarks Cards).	Modifications to programs must be submitted to NASA-MSFC for approval, then to IBM for implementation, and finally verified on the S-TB or S-V Breadboard Facility before acceptance into the program.
ATLAS The intent is that the statements, as interpreted by the compiler, should suffice for documenting the procedures. Comments, on separate cards, have no restrictions.	Compiler accessibility is a common problem and has resulted in some development of language subsets or adaptations for use in (object) test equipment. Change (configuration) control may dwarf such problems in some instances.
CLASP Primitives enable reasonable self-documentation without comments, and commentary can be inserted in any statement.	Maintenance of user programs may be complicated by the target machine dependent features of CLASP. Compiling computer accessibility and project change controls may also be more significant.
ADAP: a "reverse compiler" prints out explanations of statement actions.	TOOL: Very abbreviated statements designed for interactive review and alteration by the operator with limited display.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

General Characteristics of Compilers 10	Format 11
ATOLL The ATOLL Computer is a stand alone program which may be run on any Saturn ground computer (RCA-110A) The program is self contained and requires no additional software.	The format provides for both fixed and variable fields. The compiler provides for multicard statements. Continuation cards are permitted whose number may vary dependent upon the operator code. Only the variable field is read on continuation cards. Remarks cards (comments) have formats but are not interpreted by the compiler.
ATLAS ATLAS compilers have been implemented by several companies, including MMC. MMC's ATLAS compiler is written in FORTRAN for use on a 360/65 computer and generates code for use in an H-316 computer. The compiler is complex and requires considerable memory. See APPENDIX A.	Restricted by generally logical arrangement of types of fields or phrases. No limit on field or statement lengths after the statement numbers.
CLASP Not currently available.	Free Form.
	MOLTOL, ATOLL-II: Format is variable with control statement.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Character Set 12	Significance of Blanks 13
ATOLL A - Z 0 - 9 *, 1 ( )	Blanks have no specific significance. Dependent upon certain operators, a "0" or a "1" must be provided to indicate action and must be blank. However, the same field may be blank for another operator.
ATLAS Upper case letters A thru Z Numberals 0 thru 9 (also A thru F for hexadecimals) Symbols + - * / , ( ) . ' $ = Blank (Lower case letters represented by /A thru /Z when required for interface identifications)	Blanks generally ignored in identifiers but used as delimiters in lists of connections. Successive blanks ignored.
CLASP Letters A thru Z (upper case) Numberals 0 thru 9 Symbols + - * / , ( ) . ' $ = Blank	Blanks used to delimit identifiers, numbers, and primitives. Successive blanks ignored.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Comments or Noise Words 14	Operators 15
ATOLL Comment or REMARKS cards are utilized but are not interpreted by the compiler. A "*" in column 1 indicates a remarks card.	No logical or arithmatic operators other than Limit check and average are provided.
ATLAS Allowed only on comment cards with "C" flag or "B" flag.	Arithmatic Relational Logical (Trivial usage)
CLASP May be inserted anywhere in a source program or statement. Delimited by double apostrophes at beginning and end.	Numeric Logical Relational Boolean

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Primitive Terms 16	Delimiters 17
ATOLL All operators, table names, time cells, and program names are primitives.	Card columns, commas, and blanks in the variable field.
ATLAS 40 verbs (operation or action codes) 30 nouns (types of signals) 80 modifiers (characteristics of signals) 58 misc. (RANGE, FOR, BY, TO, etc.) 15 Connections Fields (GND, (Phase) A, B, C, etc.)	Blank $ ' ,
CLASP 79 primitives, 56 are reserved and cannot be used as programmer defined identifiers.	Blank $. " ' ,

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Identifiers (labels & names) 18	Arrays, lists, structures 19
ATOLL Labels (applied through DECLARE statements) and NAME statements.	Tables, time cells, and discrete lists
ATLAS Since all identifiers are inclosed in quote marks, primitive combinations generally allowed except $ ' ( ) symbols. Identifiers can be applied to data, dummy parameters, functions, specified characteristics of nouns, messages, predefined procedures, etc.	No general provisions except an ability to pass a two dimensional "table" of values into a predefined subprocedure when it is called.
CLASP Must begin with a letter: limited to 8 characters: may be used for data, statements (follow with period), subroutines (preceded by a period).	Arrays allowed with up to three dimensions. Subscripts can be declared to be integer constants, integer variables, or implicitly defined. Subscripts in references can be integer formulas or non-scalar (referencing an entire dimension). Limited structure capability via data grouping. No arrays of groups.
	ATOLL-II, MOLTOL: Broad capabilities are provided, including lists, pair-lists, arrays

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Program Structures 20	Block Structure 21
ATOLL First and last ATOLL operators of every test program will be NAME and END respectively. Subroutines start with BEGIN and end with RETURN. First card following NAME must be a REMARKS cards associated with NAME. CODE card to follow NAME and REMARKS card which specified consoles to be allowed interaction with a specific program. Machine Language subroutines require NAME, REMARKS, CODE and MLSR followed by the binary deck and concluded with an END card.	ATOLL programs are restricted to blocks of 500 machine statements. Provisions are made for communicating desirable block separation points in ATOLL with the use of BLOK.
ATLAS All declaratives (SPECIFY and DEFINE) are contained in the "preamble" which must precede the imperatives of the "procedure."	No specific provisions except a compiler directive flag to identify an allowable (manual) entry point. This flag is sometimes used to define allowable block boundaries, which are compiler determined if necessary in the object system.
CLASP Data declarations must appear before the main body of the program. The main program consists of imperative statements and compiler directives. Finally, procedures are located at the end of the main body of the program.	Provided for conditional statements and loops. Also appears as a subroutine-like group of statements identified by the primitive CLOSE.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Loop Structures 22	Subroutines Structures 23
ATOLL Loop structures are provided with the use of GOTO, EXEC, BEGIN and RETURN are the start and end points of an ATOLL subroutine. EXEC executes subroutine by name.	ATOLL subroutines referred to by name and called by EXEC operator. BEGN and RETN operators identify starting and concluding statements. In addition, capability provided for calling machine language subroutines.
ATLAS GO TO, ALTER, REPEAT--- TIMES, PERFORM ---- TIMES can be used to construct loops. Some loops are implied by specific verbs such as MONITOR & ADJUST and still others could be the result of object machine and compiler implementation, examples are START WHEN, STOP WHEN, DELAY, WAIT FOR, etc.	All subroutines and procedures are defined using ATLAS imperative statements; if in the "preamble" (DEFINE; ---; PROCEDURE,---) they are called by PERFORM, '---', if within the "procedure" they are recalled by REPEAT, STEP-number-THRU STEP-number-.
CLASP Simple loop capability provided of form FOR K = n_1 by n_2 to n_3 where n_1 is an integer variable or constant.	Procedure capability provided. Multiple input parameters and multiple output parameters.
ATOLL-II, MOLTOL, TOOL: DO statement provided	

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Direct Code Capabilities 24	Interaction with Operating System 25
ATOLL No direct code capability. ATOLL loses control during machine language program execution.	Execution of Machine Language subroutines or test programs causes ATOLL Executive to lose control. If errors occur during program run, system halts and displays error message to test engineer. In certain cases, test engineer alternatives are displayed.
ATLAS Capability is provided to LEAVE ATLAS and RESUME ATLAS, (presumably to allow direct or machine code insertion, but would also allow any other language insertion. There are no communications rules for such inserted segments except that a RESUME ATLAS statement in the direct code segment would terminate the direct code execution.	No provisions.
CLASP Direct Code Capability provided.	Undefined at present.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Data Types 26	Formula Types 27
ATOLL Lacking arithmatic and logical capabilities, this category is not applicable to ATOLL.	NOT APPLICABLE (Same reason as #26)
ATLAS Real Complex Floating Point Text	Numeric - Real Complex Floating Point Logical - Bit String construction and comparisons using LOGIC SIGNAL Boolean Relational - Criteria: GT, LT, EQ, NE, UL, LL Results: GO, NOGO, HI, LO, EQ, NE
CLASP Fixed point Floating point (when hardware capability is provided) Integer Boolean Test (character strings) Hardware (depends upon object computer, allows direct reference to machine registers)	Numeric - Fixed point Floating point (when hardware capability is provided) Integer Boolean Mixed Logical - Bit by bit manipulation of numeric, textual, and logical formulas. Boolean - True or false results from evaluation of relations.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Assignment Statements 28	Sequence Control 29
ATOLL Assignments provided by language include program names, set time, set indexes, increment indexes, set flags, discretes which can be issued, establishing profiles, changing profiles, assigning termination procedures, declaring names for PCM addresses.	Programs normally run in sequence by step-substep number. Succeeding numbers must be increasing but not necessarily incremented by one. Subroutines may be called by EXEC operator and may appear anywhere in program. GOTO operator causes unconditional transfer. Branching statement number contained within program statements causes conditional transfers. Executive can force terminate program according to termination procedure included in program.
ATLAS DEFINE and CALCULATE can assign data values and text. SAVE can assign a new identifier (label) to an identified value, e.g., SAVE, 'MEASUREMENT', 'DRIFT'	GO TO, ALTER and REPEAT allow transfers, GO TO can be conditional with multiple destinations determined only by Boolean Relational Results which must have been set in previous statements. ALTER STEP- - means change some field values as specified and re-execute step- -; it is a "non-preferred" verb and may someday be deleted.
CLASP Simple, multiple, nonscaler, and exchange assignment statements are provided.	CLASP statements are executed in the sequence in which they appear except as altered by control statements. Simple GOTO statement for transfer to a statement label. Switched GOTO for transfer to one of many statements or CLOSES depending on an index value. Statement of IF, THEN ELSE type used to transfer control or execute a section of code based on the evaluation of a Boolean formula.
MOLTOL, ATOLL-II; Provisions and rules for assignment are very extensive, including access to individual bits of a BITS variable.	

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Error Conditions 30	Operating System and Equipment Handling 31
ATOLL Hardware failures are sensed during run and are displayed as are certain discretes which require test console switches to be in outs position. Diagnostic error table printed after compilation of program contains explicit errors. Undefined branch table will be output to printer if undefined branches are found during compilation.	Language accommodates the outputing of discretes and analogs, and the inputing of discretes and analogs. Display of vehicle status, print, and record on magnetic tape are included. Calling of machine language Test Programs are also provided. In addition, program control via hardware response is included.
ATLAS No provisions.	Input/output provisions are made for operator interfaces, records, and definitive unit-under-test interfaces. There are no provisions for storage or memory allocations, etc.
CLASP No specific error condition sequence control statements provided.	No language statements relating to an operating system are defined. Many computer oriented language statements are defined. No input/output capabilities are provided. Use direct code and hardware declaration capability. Several arithmetic and data manipulation functions are provided in a subprogram library. Subprogram names are considered primitives.
MOLTOL: Assures a defined segment doesn't get divided by overlays when IMMED is used.	

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Compiler Directives 32	Test-Orientation 33
ATOLL Compiler control cards direct input/output options (input from tape or cards) output to printer only or to printer and tape) whether edit operation is to follow, and whether two or more object tapes are to be copied onto one tape. Comparison between master tape and several other tapes is provided.	Objectives of language directed toward Test Engineer's needs for inputing his test requirements into the computer checkout system.
ATLAS Compiler may use "flags" (if needed) from the first column of the first line of a statement, and the DEFINE and SPECIFY statements of the preamble. Flags include S (repeated later), E (entry point), B (destination of GOTO), C (commentary) and M (retain programmed connections).	Actions include CLOSE, CONNECT, DISCONNECT, OPEN, READ, SETUP, ADJUST, APPLY, MEASURE, MONITOR, REMOVE, VERIFY and WAIT FOR, all of which are reasonably accurate representations of expected actions. Others, such as DELAY, CALCULATE, COMPARE, DISPLAY, INDICATE, PRINT, RECORD, are common actions that are well understood by test personnal. The remaining actions are not difficult for test engineers.
CLASP Compiler directives are provided for debugging and timing (used with a target computer simulator), and for space and time optimization and direct code usage.	CLASP has no test oriented characteristics explicitly defined.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Engineering-Orientation 34	Man/Machine Interfaces 35
ATOLL While many terms are more engineer oriented than test oriented, the language itself is geared toward the handling of test operations rather than toward engineering problems.	The language provides for test engineer intervention from the test console. Information is also displayed to aid his decision making. Test results can be displayed and/or printed. SEMI operator and error conditions provide for predetermined operator intervention. During course of test run, test engineer may take over control of his functions and conduct test locally.
ATLAS Actions are test and programming terms familiar to most engineers. The NOUNS and their MODIFIERS (which are the largest group of primitives) are engineering terms such as AC SIGNAL, SYNCHRO, TRIANGULAR WAVE SIGNAL, TACAN, VOLTAGE, PHASE SHIFT, IMPEDANCE, MOD-INDEX, etc. In general, these terms would be meaningless to other than engineers and would be used only when the test required them. Other terms include dimensional units and pin designators (HI, LO, GND, etc.)	PRINT, DISPLAY, and INDICATE are specific operator interface terms. Messages and variables are adequately accommodated. Since ATLAS is otherwise test equipment independent, there are no provisions for operator inputs but an implication that they exist as illustrated by WAIT FOR, MANUAL INTERVENTION.
CLASP Provides engineering oriented mathematical capabilities but is primarily a programmer oriented language.	Provides message text input and output capabilities.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Records and Logs, Time Tags 36	Multiple/Parallel Actions 37
ATOLL Provisions are made for recording results on hard copy or on magnetic tape. Where time is relevant, time tags appear in the statement.	The language does not provide for multiple operations to be performed simultaneously. IUDC controllers permit simultaneous internal operations but the language does not implement this feature.
ATLAS RECORD and PRINT are obviously intended to make records. There are no specific provisions for time tagging records.	Not definable in ATLAS.
CLASP No capability provided.	No capability provided.
	MOLTOL, ATOLL-II: Include statements to start and synchronize, parallel programs.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Monitoring 38	Interrupt Initiated Routines 39
ATOLL SCAN and TEST provisions provide a limited monitoring capability. However, no continuous monitoring capability is provided by the language. Function Executors can be called from the ATOLL program (MNTR) and the frequency of tests specified; but the ATOLL program is not interrupted or alerted to any out-of-tolerance conditions.	Essentially no interrupt capability provided by the language. Conditional provisions can give control to the test console or branch within the program.
ATLAS MONITOR provides for repetitively measuring and displaying a single parameter until stopped by operator intervention. There are no parallel actions definable in ATLAS.	No general provisions except operator interrupt of MONITOR routine.
CLASP No capability provided.	Inhibit/Enable interrupt capability provided. Capability for execution of sequence of statements upon occurrence of enable interrupt and return is provided.
	MOLTOL, ATOLL-II: Include provisions to POST procedures to be executed for specific interrupts or operator intervention.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Test System Dependency 40	Program (Project) Orientation 41
ATOLL Considerable test system nomenclature included within ATOLL statements.	Language has been implemented to provide for specific test system and for the Saturn IB and V programs.
ATLAS There is dependency only to the extent that display, recording and a universal UUT signal interface capabilities are assumed.	ATLAS is not project or program oriented except by the limitation of its engineering terminology.
CLASP CLASP has many computer dependent features, such as in-line assembly code capability, hardware register control, and dependency on computer arithmetic functions available. No test system dependency since CLASP is not designed for test orientation.	No orientation to any specific project.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Clock & Time Controlled Actions 42	Program Controlled Indicators 43
ATOLL Language provides for countdown and Greenwich time activities. Delayed time is utilized by operator. CDT or GMT can be compared with test event occurrence and program halted until desired time is reached.	7 Index registers can be used for counters or program control. 48 flags can be defined.
ATLAS There are no explicit provisions for external clock or time inputs. Individual statements allow time delays to be specified relative to other statements.	The Boolean Relational Results (GO, LO, etc.) are set and reset by evaluations and remain set until further evaluations occur. Any other status indicators or flags are awkward to create and maintain in ATLAS.
CLASP No specific capability included in language. Response to clock interrupts could be handled via interrupt capability.	No specific indicators provided.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Multiparameter Tests 44	Special Discipline Provisions 45
ATOLL No provisions for multiparameter testing.	Checkout system provides for special disciplines but language has not unique capabilities for hydraulics, pneumatics, or propulsion.
ATLAS Each action verb involves a single parameter, however, a group of such tests could be programmed in a DEFINE procedure and subsequently called by a single PERFORM statement.	ATLAS contains many discipline oriented noun-modifier sets, e.g., PAM, TACAN, VOR, MANOMETRIC. (NOTE: When MANOMETRIC is used the signal is actually pressure--not an electrical analog--the transducer is a part of the test system controlled by ATLAS).
CLASP No capability provided.	Specialized terms available with regard to target computer machine instruction operations.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Interface Characteristic Specifications 46	Test Level (Unit, Subsystem, System) 47
ATOLL Language does not accommodate interface characteristics of the unit under test. (Input impedance, output impedance).	Language, checkout system, and unit under test are dependent upon installed subsystems and systems test only. Checkout system structure change would provide black box test capability but would be inefficient.
ATLAS Interfaces, physical as well as electrical, can be fully specified, as can accuracies, connection identification, loading, and ranges. ATLAS has some signal type deficiencies that are to be added later.	ATLAS is intended for unit (black box) and lower levels of testing by a very general purpose test system. It can accommodate higher levels, but not efficiently.
CLASP No capability provided.	No capability provided.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Digital Interfaces 48	Execution Rate Control 49
ATOLL Digital interfaces are accommodated by machine language test programs which can be called by ATOLL operator. ATOLL executive loses control during machine language test program operation.	No capability provided by the language to vary rate of statement execution. DELY operator may be used to do this but would be highly inefficient.
ATLAS LOGIC SIGNAL includes same provisions but lacks complete definition. Additions are being studied.	There is no explicit execution rate control. Statements are assumed to be executed asynchronously as rapidly as the test system can respond.
CLASP No capability provided.	No capability provided.
MOLTOL; and VTL provide for uplink digital transmission, execution of UUT digital program, downlink transmission of digital data, and verification of transmitted data.	MOLTOL provides for assuming 100 milliseconds between execution of succeeding steps until a statement changes the interval.

TABLE 1. LANGUAGE COMPARISON TABLE (CONTINUED)

Unique Capabilities 50	
ATOLL	
ATLAS ATLAS is a test system programming language rather than a computer language. Inputs and outputs are hardware system interfaces and may be physical as well as electrical.	
CLASP CLASP has several capabilities that are considered unique with respect to other higher level languages: Extensive fixed point capabilities with respect to scaling. Temporary variables which take on variable attributes depending on usage. Provision for modifiable constants (those which are fixed for execution of a program but can be modified between executions) Scaling control over intermediate results. Compiler optimization directives. Stress on object code efficiency.	

IV. DEFICIENCIES AND PROBLEMS OF EXISTING LANGUAGES

This section summarizes some of the major deficiencies of the languages studied. Section II contains additional discussions relative to the specific languages.

The most common complaint about any TOL is that it is too difficult to learn to read and understand. This is usually brought about by the use of mnemonics, fixed fields to distinquish parameters, etc. and by the use of terms that the language designers erroneously considered to be generally understood.

The facilities of the language, such as declarations, specifications, and definitions, can usually be used by the writer to simplify his writing task at the expense of readability. Since these same facilities can be used to enhance readability (and sometimes <u>are</u> so used) the result becomes more a function of the writer's motivation than of language definitions and rules. Some tendency to overwork DEFINE capabilities has been noted in ATLAS test procedures.

The absence of arithmetic capabilities has been noted for several of the languages. It is possibly true that few system level test procedures would require the capability, and also possible that vehicle test philosophies, and organizational charters might influence the language definitions in this area.

The absence of digital data transfer and computer intercommunications capability might be due to the fact that the user is programmer-oriented and the use of lower-level languages and codes is not so much a problem. Also, the language designer may find it particularly difficult to adequately provide the necessary terminology.

Inconsistency of rules is a problem mentioned for some languages, but does not seem to be widespread. It is suspected that this type of problem is a result of "minimum-impact" type of change decisions after a language has been implemented.

V. SAFEGUARDS AND CHECKING PROVISIONS

Ideally, the design of a Test Oriented Language should provide limited entries and few rules to be followed to insure a minimum of human errors. With few entries and few rules, program errors can be easily detected and corrections to the program made. Simple form and text would provide for manual checking to be accomplished. However, safeguards to the system-under-test require knowledge of the checkout system and the unit under test. As the program length increases, (the number of operations to be performed and the number of program statements increase) the building in of safeguards and the provisions for checking become difficult, if not impossible.

The language may provide for restricting the number of consoles which may interact with the program and thereby prevent an inadvertent action from a non-allowed console. A specific subsystem under test will have a well defined command interface, and any required stimuli (analog or discrete) may be identified in the preamble. If, during the procedure portion of the program, an illegal command function was issued, the compiler would not accept the command, and the error would appear on the listing. Several languages provide this capability by listing allowable command functions.

The language should have a capability for establishing a course of action in the event a problem is encountered during the run of a program. A problem that is potentially hazardous or one left unattended, could cause degradation of the test article. A backout sequence or a shutdown sequence should be provided and the test console operator informed of the event and action being taken. In some languages this capability was implicit. In other languages, where a definite operator did not exist, subroutines could be established to provide this capability.

Provisions should be made in the test language for informing the test console operator of marginal conditions, halting the program at a convenient step, and allowing the test console operator to terminate or continue with another sequence, dependent upon the information presented on this display.

The more flexible the program input format and the greater the number of rules, the more likely human error will result. Including safeguards and checking provisions within the language appears to offer a limited capability due to the relationship which exists between the test article, the checkout system, the language and the human involved in the writing the program. The majority of the checking and safeguards must reside in the compiler for format errors and illegal operations and in the operating executive for protection of the system under test.

VI. PROGRAMMING AND READING AIDS

In almost every instance, the extensive use of a programming language has resulted in the generation of specific writing aids. Not so common, however, is the origination of reading aids. This is considered unfortunate since there are generally many times as many readers as writers, and decisions to automate testing and implement languages are more commonly made by readers.

One conclusion of this study task is that the greatest possible aid, once the language has been defined, is a good user's manual with clear and concise rules and explanations using terminology that is readily understandable by engineers. It is unfortunately true that a good TOL can be defeated because concepts used in defining and explaining it are totally new and difficult for a potential user. In some cases, simply the organization of a manual may be a hurdle. Language specifications often utilize terms, conventions, and concepts which make it unsuitable for use as a user's manual. In such cases, a separate manual should be provided. Examples, syntax diagrams, tabulation of rules, explanations of each form and option, and a glossary of terms and primitives should be included.

Probably the most common writing aid with fixed format languages is the coding form. In some cases, such as ATOLL, such a form is really mandatory. Unfortunately, the reader with a print-out of the test program has no such assistance. In the case of ATOLL-II and MOLTOL, the fixed format fields can be respecified by the user, which might really complicate the problem for readers.

Another common aid is a listing and brief explanation pamphlet of key terms and concepts.

A reading aid in the ACE-S/C ADAP inplementation provides for the compiler to print out, in pre-stored message form, an explanation of the actions directed by the ADAP "statement" or code.

In order to decrease the writing time for some languages, many words have standard (primitive) abbreviations that are recognized by the compiler or translator as identical to the unabbreviated words. When this is the case, as it is with CTL and VTL, the compiler may use the full word in print-outs and thereby restore readability.

The implementation of the compiler to check source language programs and provide output messages (perhaps codes) that enable the source code writer to rapidly pinpoint errors is considered a necessity.

Perhaps the most promising aid to a writer is an interactive system which allows the writer to construct statements and programs by yes/no, multiple choice, and fill-in-the-blanks answers as directed or requested by a display. The TOOL system has such a provision. It is described here as an illustration of the concept.

TOOL Interactive Programming Aid

The TOOL system is a highly interactive system providing the user, in this case a test engineer or astronaut, with complete instructions on the use of the system. Messages to the user appear in three forms; status indicator lights, variable messages output on a CRT-like plasma display, and fixed messages placed on microfilm and automatically displayed as required.

The lights generally indicate the present status of the system in response to operator pushbuttom activity. The variable messages are displayed in response to the operator's action of typing in language statements or in response to review requests. Error messages and the current language statement are displayed. The fixed messages are displayed to guide the user in the proper use of the system and to instruct the user in the syntax and semantics of the test language itself.

A generalized example of the operation of the TOOL system will serve to clarify the preceding ideas. Consider the case where a test engineer knows the sequence of steps to accomplish a desired test in a conceptual form but has no experience in using TOOL. He sits at the OCS console and pushes the "select sequence" pushbuttom. The select light is turned on and a microfilm frame is displayed asking the user to type in a name for his test and explaining how this is done. Upon entry of the name, a fixed message informs the user that the name he has used is not already taken and what he must enter next. The plasma screen displays the name.

The user continues to type in the statements required to establish priority, protection, and password requirements for the test. The information is build up by the user as explained by the microfilm frames and appears on the plasma screen.

At the completion of this operation, a microfilm frame displaying the allowable choice of primitives is displayed. The user selects one, and the name of the primitive appears on the plasma. At the same time the information concerning the first modifier required appears on the microfilm display. The user responds accordingly and continues on in response to other microfilm frames until he completes a statement in the language. The statement is build up on the plasma display during this process.

When the statement is completed, the microfilm frame displaying the choice of language primitives available appears again. A new statement is constructed as above and this process continues until the user has created a complete test.

A microfilm frame then appears giving the user instructions and options as to the proper disposal of the test he has just written. Should he desire to review his test he may do so, looking at each statement in total, or at each modifier in order. Information with respect to altering or adding to his statements appears during this review cycle.

When the user has completed the review and/or modification of his test; a microfilm frame again appears giving him instructions as to the disposal of his test. He may choose to save it for future use, execute it, or discard it. Proper disposal of the test is selected, completing the process of creating an executable test using the TOOL system.

VII. CONCLUSIONS

This study task has investigated both the languages and some of their applications and usage environments. Those that have been applied have indeed aided in accomplishing automation by aiding the communications problem. All could be improved.

It is interesting to note that practically all Test Oriented Languages (TOL's) established the same objectives to direct the design of a language useful in accomplishing automatic checkout tasks. This is not incongruous as the requirements which originated the need for a TOL were basic to all automatic checkout systems: time, schedule, cost, communication. The test procedure writer was not intimately familiar with the computer or automation techniques; the programmer was not familiar with the engineering terms used in testing and had no knowledge of the vehicle to be tested and little inclination to learn. On the other hand, the test engineer was completely swamped with the requirements of his own task and felt put-upon by having to learn how to fill out the forms necessary to provide test data to the programmers. Communication between these two worlds was often a real obstacle.

It was logical to expect that similar testing groups throughout the country would independently assess the problems of automatic checkout and immediately determine similar methods for improving communications, providing for accomplishing many tasks with one by specifying the requirements for a TOL, and reducing schedule and cost impacts.

However, few TOL's have been able to accomplish all of these objectives. No doubt, compromise has affected the end result. Having a specific test article in mind which was well developed and test equipment available undoubtedly affected the resulting TOL. In all examples studied, with one exception, this has been the case.

A common tendency with all of the TOL's studied (perhaps ATLAS excepted) is for the language to be writer oriented. The common reader oriented objectives seem to be subverted by the writer's natural desire to reduce the number of characters to be written on the coding form. This results in abbreviations, mnemonics, fixed formats, unnatural (but shorter) word usage, and other forms of coding that are non-English like and require study by engineers who should be able to understand the test programs but don't really have the time. The writer is generally supported by the compiler designer because of the simplifications possible in recognizing and analyzing source language primitives and statements. Since the writer is frequently dedicated full-time to the task, the time to learn the language is not of major significance. He is also more apt to be involved in the language definition.

The CLASP language, as presently defined, is not suitable for direct use in a test-oriented environment, although some of its features in the non-test-oriented terms are as suitable as those of most TOL's.

None of the TOLs considered in this study would fully satisfy the broad test-oriented applications area as envisioned by the authors. Work would need to be done to further the goal of test system independence of the languages other than ATLAS. ATLAS would need additions for system-oriented functions, and probably deletions in detailed specifications areas.

It is believed that almost every desirable feature of a new language is provided by one or more of the languages studied. The next phase of the study will concentrate on the identification and justification of these desirable features.

VIII DEFINITIONS

Assembler	A program that prepares a machine language program from a source program which consists of symbolic notation for both operation codes and addresses in a one-to-one relationship with machine language.
Asynchronous	Occurring without a regular time relationship. The occurrence of an asynchronous event is unpredictable with respect to instruction sequence.
Attribute	A characteristic attached to a data item.
Compiler	A program that prepares a machine language program from a source language program by making use of the overall logic structure of the source program, or generating more than one machine instruction for each source language statement, or both.
Compiler Directives	Information supplied to a compiler to provide assistance in translation of a source program which does not result in direct creation of executable code.
Concurrent Execution	Execution of computer programs in a multiprogramming mode.
Cuing	The assistance provided a program writer by messages automatically generated by a computer in an interactive environment.
Declarative Statement	A special case of compiler directive which provides information to the compilter concerning the data elements of a program.
Delimiter	A character (or characters) that categorizes, separates and/or organizes items of data or language statements.
Function	A special case of a subroutine with a single output which can be used within expressions just as a number or variable may be used.

Global Scope	That scope of a variable which specifies that the variable has the same meaning for each use of the variable throughout a computer program and all its subunits.
Higher Order Language	A language which enables the user to write programs for a computer without the need for detailed knowledge of the actual workings of the computer. Generally requires significantly fewer statements than a lower order language.
Identifier	A character or set of characters whose purpose is to identify, indicate, label or name a body of data, such as a procedure, function or variable. It is assigned, determined by the programmer rather than the language: contrasts with primitive.
Imperative Statement	The language statements that specify executable actions to be performed by the programmed system or change the sequence of such actions.
Interactive	Pertaining to a system in which a user can actively communicate with a computer while creating and/or executing programs.
Interpreter	A program which executes a source program on a unit-by-unit basis. In the OCS application, a program that takes the data output from a translator, which represents source program statements, and passes it to appropriate routines for execution.
Literal	A string of characters which represents itself rather than the location of something else.
Local Scope	That scope of a variable which specified that the variable have the same meaning for use only within a particular subunit of a computer program and is undefined for any references outside of that subunit.
Machine Language	A language that is used directly by a machine. It consists of the actual binary bits which are interpreted by the computer hardware to control instruction execution.

Macro Capability	A language capability that allows a user to specify a number of language statements via the use of a single language statement. The single language statement is, in effect, a new primitive of the language.
Multiprogramming	Pertaining to the interleaved execution of two or more programs by a computer.
Optimization	Refers to techniques for the generation of space or time efficient machine code output from compilers.
Primitive	The set of basic elements of a language as opposed to user defined identifiers. Primitives consist of graphic operators, those characters which have a defined semantic meaning as an operator; keywords, those words which have a fixed meaning in the language; and punctuation characters which serve as delimiters.
Problem Oriented Language	A programming language designed for the convenient expression of a given class of problems.
Procedure Oriented Language	A programming language designed for the convenient expression of procedures used in the solution of a wide class of problems.
Relational	Pertaining to the relationships between quantities, i.e., equal to, greater than, less than, etc. Sometimes called Boolean Relational because of true or false type of results.
Relocation	The process of moving a program from one location in storage to another and adjusting the necessary address references so that the program can be executed in its new location.
Scope	Pertains to that portion of a computer program throughout which a variable has meaning. See global scope and local scope.
Self-Extension	That capability of a language which allows the user to define new primitives for the language. Most commonly represented by a macro and a function capability.

Semantics	The relationships between symbols and their meanings.
Special Purpose Language	A language designed to satisfy a single objective. One such objective is the solution of problems in a particular application area.
Source Language	A language used to write computer programs for input to a given translation process.
Subset	A language which contains some of the features of another language but not all the features and/or contains restrictions not present in the original language.
Syntax	The rules covering the structure of expressions in a language.
Target Computer	The computer for which executable machine instructions are produced by a compiler and not necessarily the same computer utilized by the compiler.
Translator	A program that converts source language statements into another form or language for further processing.

IX. BIBLIOGRAPHY

1) Programming Languages: History and Fundamentals, Jean E. Sammet, Prentice-Hall Inc., Englewood Cliffs, N. Y., 1969, Chapters 1, 2 and 3.

2) Flight Computer and Language Processor Study, R. J. Rubey, W. C. Nielsen, and L. Bentley, July 1969, Contract No. NAS 12-2005, NASA, Prepared by Logicon, Inc.

3) Spaceborne Software Systems Study, Technical Documentation Report No. SSD-TR-67-11; Vol. 1, Summary; Vo. 2, Survey, Analysis and Recommendations; and Vol. 3, Recommendation for a Common Space Programming Language.

4) Space Programming Language/Mark II Programmer's Manual, 20 February 1970, SAMSO TR 69-421, Developed by System Development Corporation for the Air Force.

5) Design of an Onboard Checkout System, Final Report, Volume II - Technical Results, NAS 9-4899, MCR-66-12, March 1966.

6) Prototype Digital Test Set for the Checkout Systems Experimental Facility, Operations/Maintenance Manual, NAS 9-6630, MCR-67-260, July 1967.

7) Flight Packaged Onboard Checkout System Development Unit, Operation and Maintenance Manual, NAS 9-8000, MCR-69-399 (Rev. 1), November 1969.

8) Flight Packaged Onboard Checkout Systems Development Unit, Software Documentation, Overall Specification - Level 2, TOOL - Test Oriented Onboard Language System, NAS 9-8000, MCR-69-192 (Rev. 2), March 1970.

9) ATLASS - "Abbreviated Test Language for Avionics Systems," ARINC Specification 416-1, 1 June 1969, Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401.

10) ATOLL - "Acceptance Test or Launch Language," Appendix A, Specifications of the Operating System for the Saturn V Launch Computer Complex, Vol. II, IBM No. 66-232-0001, NASA MSFC.

11) MOLTOL - "MOLTOL Test Writer's Reference Manual," First and Last Version, Friday, June 13, 1960.

12) ATLAS - A Standard Compiler Input Language for Commercial Airlines, Thomas A. Ellison and Laurence S. O'Neill, Proceedings of the Automatic Support Systems Symposium for Advanced Maintainability. St. Louis Section, IEEE, November 1968.

13) ATOLL-II Language Reference Manual, by General Electric Company, Huntsville Operation, for NASA MSFC Computation Laboratory.

14) CTL - Cage Test Language Description, Martin Marietta Corporation, 1 February 1968.

15) VTL - Viking Test Language Description, MMC Contract No. NAS1-9000, 29 December 1969.

BIBLIOGRAPHY (Cont)

16) ADAP - MA0201-D499, Intercommunication Subprogram Specifications, North American Aviation Training Manual, ACE-S/C Programming. General Electric Company, Volume 3, Computer Programming, 24 January 1969.

17) ACEP - ATM-U002-0, Automatic Sequence Execution and Processor (ASEP), General Electric Company, Huntsville, Alabama.

18) Advanced Software System Study, Final Report, 69-811-2102, General Electric Company Contract NASw-410 S/A No. KSC-360, December 1969.

19) Automatic Checkout in the Saturn Program, Charles O. Brooks, Jr. and Max E. Rosenthal, NASA MSFC, from Automation in Electronics Test Equipment, Volume III, New York University, April 1967.

APPENDIX

AN ATLAS COMPILER

The Martron Systems of Martin Marietta Denver Division is currently producing an ATLAS compiler and ATLAS test procedures for airlines use. A team of test writers who have little or no programming experience has been recruited to provide the required procedures. A training program has been instituted which requires a new test writer to create an ATLAS test sequence from a sample problem which contains 80% of the typical problems for which the ATLAS language is used. The learning curve for these personnel varies from one week to one month, depending partially on their background in test specifications. At the end of this time they are successfully producing ATLAS test procedures.

Sample ATLAS test procedures have been reviewed by airlines personnel with 20 or more years experience in testing of aircraft components, but with little or no programming experience. These procedures have been judged to be very understandable and have been well received by the test engineers. Part of the reason for this acceptance has been Martron's policy of keeping all procedures simple and straightforward. Complex statement configurations possible in ATLAS have been avoided in production of the test procedures.

Test procedures have been implemented for the checkout of both analog and pneumatic units. The language has been capable of meeting all requirements in these areas. However, it appears that the language may need expansion to better fit the requirements of airline checkout. One area of expansion concerns better synchronization capabilities which are required for the testing of digital and logic type units.

The Martron Systems' ATLAS compiler implements a major portion of the ATLAS language. It is written in FORTRAN IV for use on a 360/65 computer system with 228 K user core, (1) 2314 disk, and (1) 800 BPI 9 track tape drive. The compiler produces object code for the Honeywell 316 computer which is part of the Martron 1200 test equipment. The basic Martron equipment includes the following stimulus and measurement devices.

1 - 512 pin programmable switching matrix.
1 - D-A converter, capacity + 100 volts DC at 1 amp.
1 - D-A converter, capacity + 10 volts DC at 100 ma.
3 - D-A converters, capacity + 100 VAC or + 10 VDC
2 - Digital to Synchro converters.
1 - Analog to digital converter with input conditions
1 - Synchro to digital converter.
26 - Constant voltage power supplies

The compiler is divided into five main sections (setup, pass 1, pass 2, pass 3 and the object code lister).

APPENDIX

AN ATLAS COMPILER (Cont)

I. Setup

The setup module divides blank common into 100 equal units and allocates these units to 8 separate tables, each table receiving a percentage of the available storage.

After allocating blank common setup scans the subroutine library files and builds an entry in the procedure reference table for each program in the library.

After completing the procedure reference table, setup then calls the load module of pass 3 to load each resident program and complete linkage between resident programs. The resident system is then written on the output tape and becomes the real-time monitor for the MARTRON 1200.

II. Pass 1

This module reads one statement at a time, and performs syntax checking. In doing so, it translates the statement into a coded form, which is a fixed format determined by its verb. Numbers are replaced by literal table pointers, symbols are replaced by symbol table pointers, and connector identifiers are replaced by interface table pointers.

III. Pass 2

Pass 2 translates the coded verbs generated by Pass 1 into the appropriate machine instructions, subroutine calls, and data to perform the verbs. In doing this, counts are kept of the number of words generated and the total size of subroutines which have been called. When the amount of object memory available has been exceeded, the program is segmented and an end of assembly flag is generated.

Translation is then resumed at the point where the program was segmented, after setting the above-mentioned counters to zero.

The result of Pass 2 is a series of relocatable object modules which reside on intermediate disc storage.

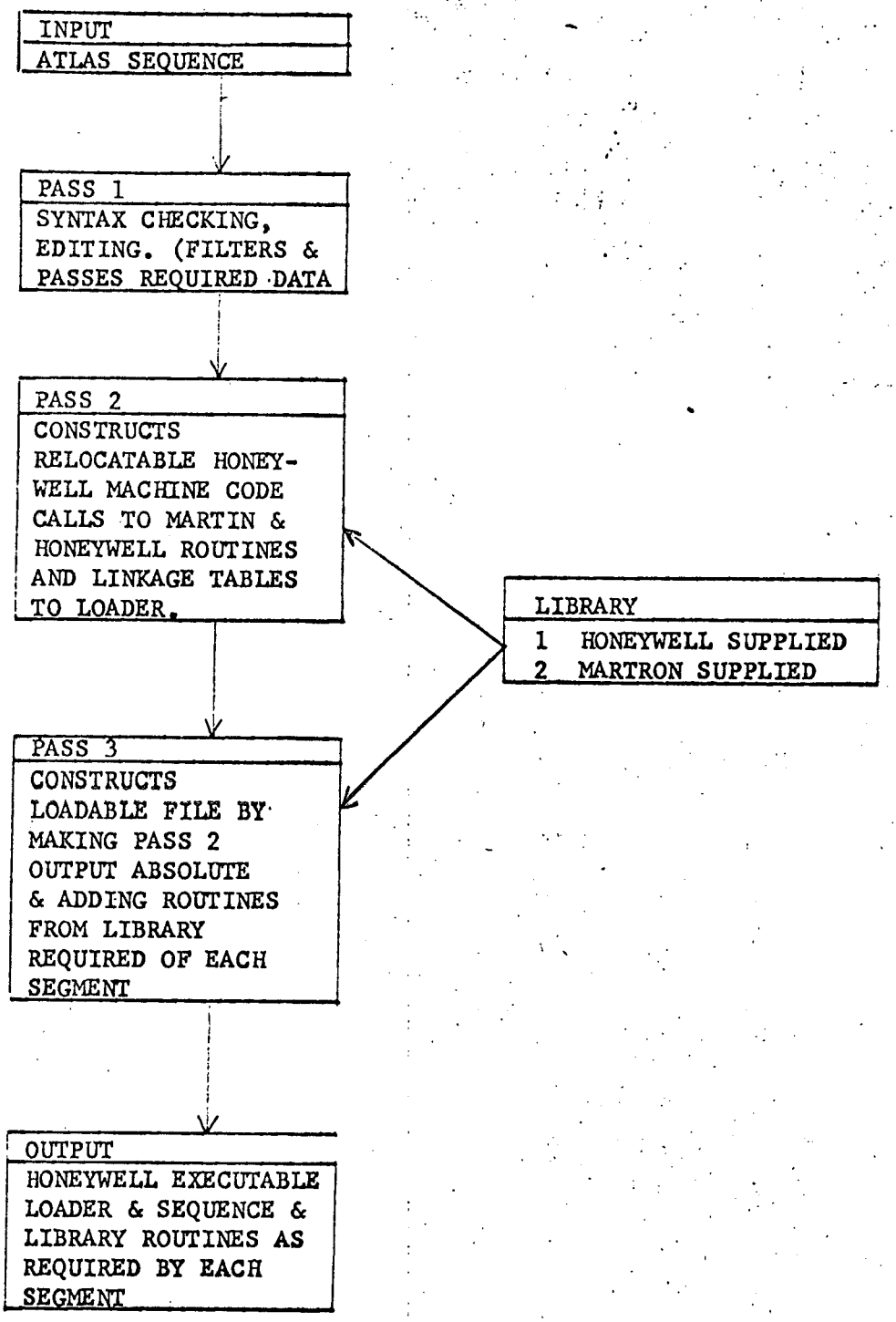
IV. Pass 3

The relocatable object modules generated by Pass 2 are read in and loaded into a core-image array, one at a time. After loading a segment, the subroutines which it requires are loaded and linkage is completed. The completed segment is then written on the output tape.

This process is repeated for each segment of the ATLAS program.

V. Object Code Lister

The object code lister utilizes the output of Pass 2 to create a listing which contains octal representations of machine code generated, identification of subroutines called, and those statements flagged as branch destinations.


INPUT
ATLAS SEQUENCE
PASS 1
SYNTAX CHECKING, EDITING. (FILTERS & PASSES REQUIRED DATA
PASS 2
CONSTRUCTS RELOCATABLE HONEY-WELL MACHINE CODE CALLS TO MARTIN & HONEYWELL ROUTINES AND LINKAGE TABLES TO LOADER.
LIBRARY
1 HONEYWELL SUPPLIED
2 MARTRON SUPPLIED
PASS 3
CONSTRUCTS LOADABLE FILE BY MAKING PASS 2 OUTPUT ABSOLUTE & ADDING ROUTINES FROM LIBRARY REQUIRED OF EACH SEGMENT
OUTPUT
HONEYWELL EXECUTABLE LOADER & SEQUENCE & LIBRARY ROUTINES AS REQUIRED BY EACH SEGMENT