Report Number: WUCS-79-5

1979-06-01

Verification Procedures Supporting Software Systems Development

Gruia-Catalin Roman

A software system development methodology is proposed. Its significance lies in the capacity to support a systematic and well-formalized error detection strategy extending from requirements definition through program implementation. A minimum set of checkpoints is suggested and verification procedures are detailed for each. The significant cost reducing potential of the approach and the way it was implemented in a production environment are also discussed.

VERIFICATION PROCEDURES SUPPORTING SOFTWARE SYSTEMS DEVELOPMENT

Gruia-Catalin Roman

WUCS-79-5

June 1979

Department of Computer Science
Washington University
St. Louis, Missouri 63130

As appeared in Proceedings of the 1979 National Computer Conference, June 1979, pp. 947-956.

ABSTRACT

A software systems development methodology is proposed. Its significance lies in the capacity to support a systematic and well-formalized error detection strategy extending from requirements definition through program implementation. A minimum set of checkpoints is suggested and verification procedures are detailed for each. The significant cost reducing potential of the approach and the way it was implemented in a production environment are also discussed.

Keywords: software systems development, methodology, verification.

Verification procedures supporting software systems development

by GRUIA-CATALIN ROMAN
Washington University
St. Louis, Missouri

INTRODUCTION

Particularly in the context of large software systems, prevention and early detection of errors during product development are critical factors in controlling cost and quality. Top-down design, structured programming, informal program verifications, and code inspection are some of the tools presently being used in order to reduce the probability of error. However, current methodologies fail to provide a systematic, comprehensive and well formalized error-detection strategy. In response to this need, a new software development methodology is proposed. The approach, described in the next section, incorporates a cohesive set of verification procedures that enables the validation of each development stage, from requirements definition through individual program implementation. The description of the verification procedures is part of the third section, while the fourth section deals with some managerial aspects related to the practical implementation of the advocated techniques in an industrial environment. A summary and conclusions are presented in the fifth section.

The approach described in this paper has already been adopted as standard practice by the MIS Department of the Monsanto Company of Saint Louis with the anticipation of considerable savings in software development and maintenance costs. Preliminary data already indicate widespread acceptance and significant qualitative improvements. However, quantitative data that would allow for a complete evaluation of the methodology will not be available for quite a while due to the considerable development time required by most systems currently being built at Monsanto. A detailed evaluation of the sociologic and economic impact of the method will be made public at a later date.

THE METHODOLOGY

The development of a software system typically has five stages:

- Requirements definition
- System architecture design
- Program design
- Program implementation and testing
- System integration and testing

The high level of convergence on the basic issues is reflected in a wide variety of approaches. Differences in managerial procedures, company standards, specification techniques, and design strategies make each methodology unique (e.g., References 6, 8, 10 and 11). The methodology proposed here distinguishes itself by the strong emphasis it places on verifiability, not on originality of design and specification techniques, which are described in the remainder of this section. The verification procedures will be presented separately in the next section.

The *requirements definition* stage establishes what functions are to be implemented by the target system. The functions reflect the user's needs and are the basis for the implementation. The requirements definition stage consists of a data-gathering phase (interviews with the user), a synthesis phase (formulation of a functional model) and an evaluation phase (study of feasibility, profits, user environment impact, resources, scheduling, etc.). The functional model supports the design process by providing a complete, precise, relevant and easy-to-access description of the problem at hand. Elements that relate to possible implementations rather than the problem description ought not to be included.

The formalism chosen to specify the functional model is a set of top-down functional diagrams. Each diagram, whose graphic symbols are explained in Figure 1, results from the decomposition of a single function present on an immediately higher level of abstraction. The "permanent record" symbol is used to indicate explicitly the memorization function while "external input" signifies an interaction with the outside environment. The only interdependence that can be expressed between functions is the relation "function F1 provides information I12 to function F2." It was found that this particular relation is the only relevant one since it suffices both for the purpose of designing the system and for establishing its correctness. Figure 2 contains a sample functional model. Although an English narrative is required to accompany each diagram and to explain its items in order, the narrative was omitted for the sake of brevity.

It is usually necessary to break down complex functional models into several parts describing related subsystems. The

a) Function that will be decomposed on the next level.

b) Function that will not be further decomposed.

c) Information; each arc must include an information symbol.

d) External input; the dotted line represents a feedback arc.

e) Conditional flow of information.

f) Permanent record; the information it saves may be used or updated.

g) Fork.

h) Join.

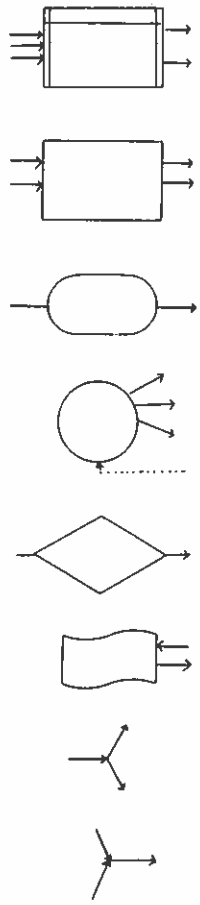


Figure 1—Functional diagrams symbolism.

criteria listed below can assist in the selection of groups of functions that are to establish the requirements for each subsystem:

- Parts of the functional model that are independent, having no connections, could represent separate subsystems.
- Functions that must be present concurrently on-line or satisfy other temporal and spacial relations should be part of the same subsystem.
- Logically-related functions should be associated together.
- Subsystems that are too large are difficult and costly to develop, while those that are too small may be wasteful.
- Few, simple and stable interfaces between subsystems assure a better chance for success.
- The selection of subsystems should assure that the potential new links between functions (due to changes in specifications) either are confined to one subsystem at a time or may be achieved by a minimum number of changes in, preferably, one interface.

The list given above is by no means complete. Furthermore, caution needs to be exercised in achieving a proper tradeoff among conflicting criteria.

During the *system architecture design* stage, programs, flow of control, input-output devices, and relations between programs and data are identified. The system architecture specifies the manner in which the functions considered in the requirements definition stage are to be implemented. Top-down stepwise refinement was selected in order to generate a set of top-down *flow diagrams* expressing the system architecture. The building blocks for the flow diagrams are shown in Figure 3, and examples of flow diagrams are provided in Figure 4. The flow diagrams and the narrative that is required to accompany them cannot completely specify the system architecture; a detailed design of the data structures needs to be included as a separate document. The relation between functional and flow diagrams will be identified better in the section to come; however, it must be pointed out that the top-down nature and labeling conventions of the functional diagrams assure quick access to requirements information, which is critical for a fast and error-free design of the system. As the need for more detailed flow diagrams arises during design, additional decomposition of the functional diagrams has to be carried out based on new interview data.

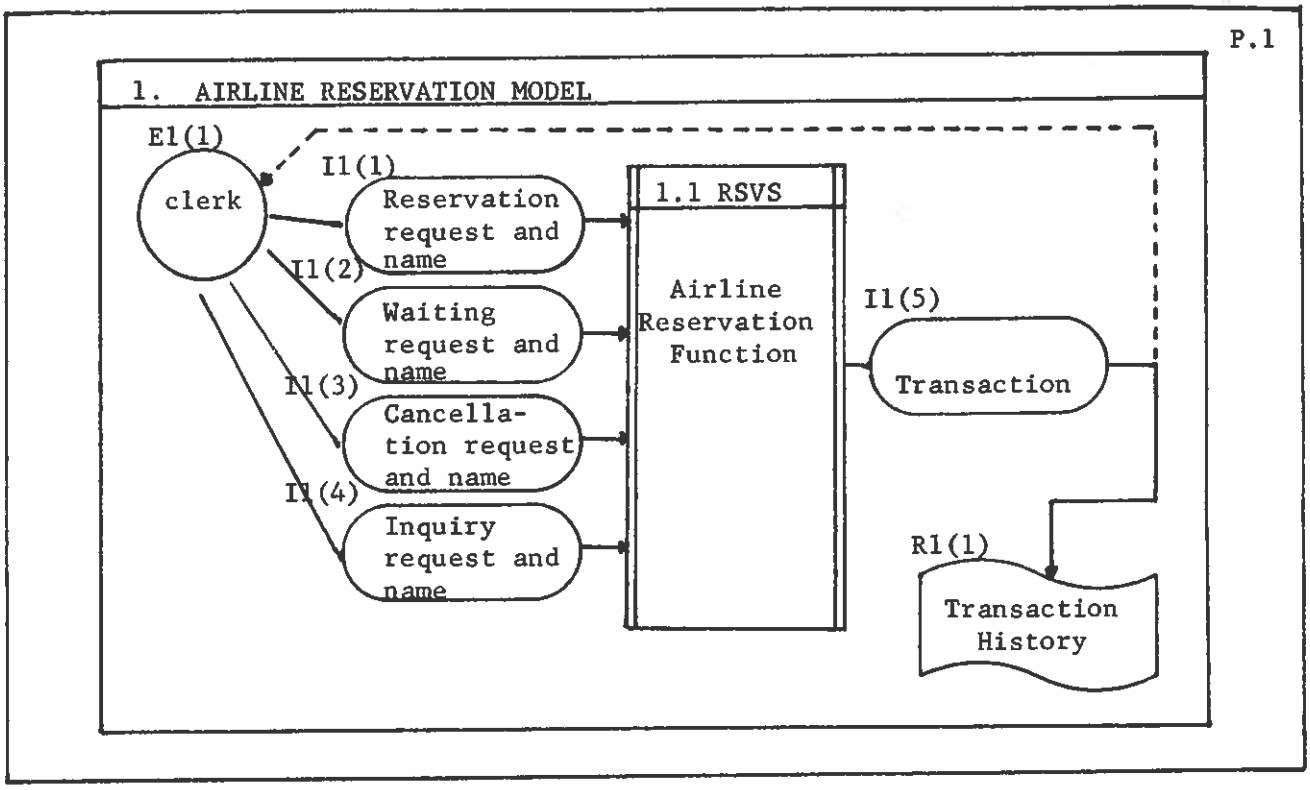
It is necessary to split the development of large and complex systems into several stages which can be implemented serially or in parallel. (The method is highly advisable for smaller systems as well.) There are two basic ways to carry out this multi-staged development:

1. *Parallel development* (also called system modularization). Parallel development is advisable primarily when working on a very tight timetable and should not be overused. The approach consists of dividing the system into several subsystems which are then developed in parallel. The key to success is to assure minimal need for communication and coordination between teams working on different subsystems. Therefore, in selecting the various subsystems one should assure that:

- The subsystems have very few interfaces.
- The interfaces are simple in nature and unlikely to change.
- The subsystems are conceptually independent and can be independently tested.

2. *Iterative development or system growth.* A more effective and safer approach to constructing large systems is the iterative method. This technique consists of developing an initial incomplete system which is later augmented by incorporating new features. Selecting the initial system, called the core, is the critical aspect of this method. Here are the basic guidelines to be used in choosing the core system:

- The core system must be general and highly flexible.
- The core system should allow for extensive growth at very low cost.
- Additions should not change the core (they may expand it) and should not affect its functions.



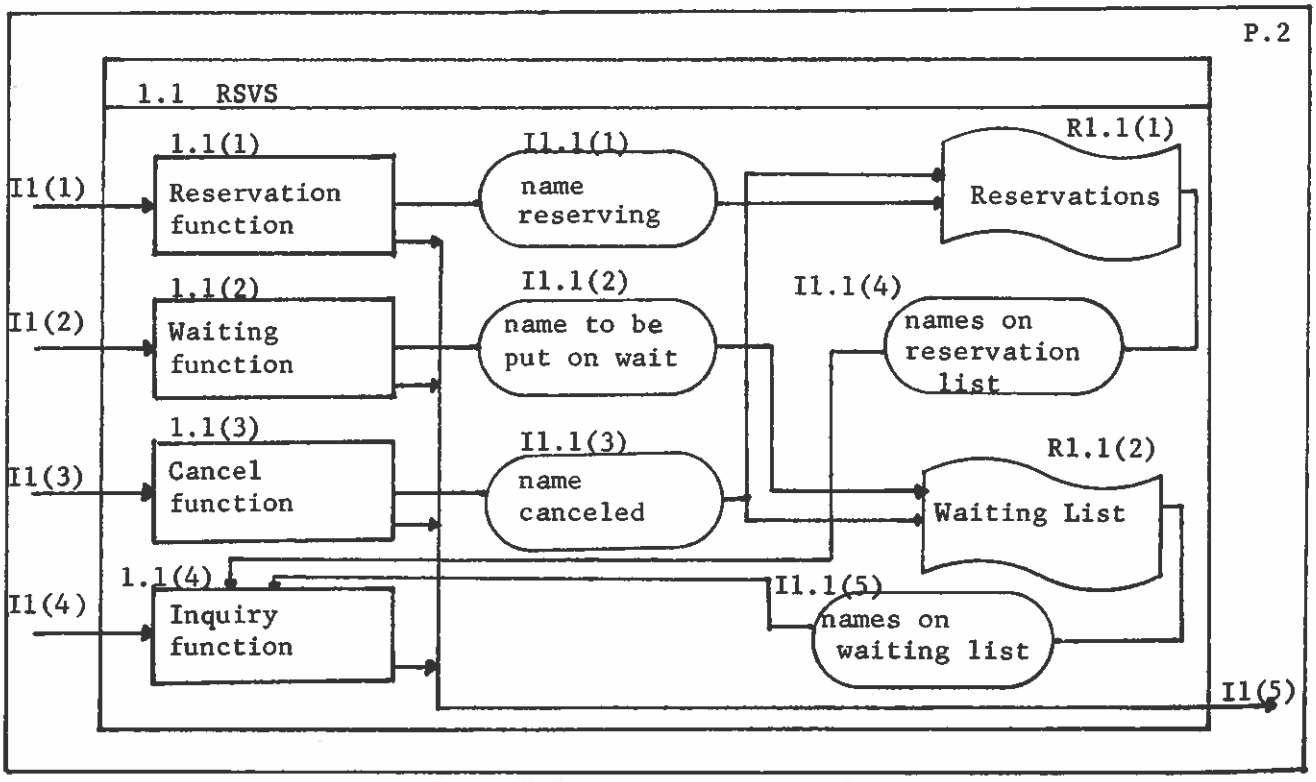


Figure 2—Requirements definition (function diagrams).

a) System component that needs to be further decomposed.

b) Program (will not be decomposed any further).

c) Devices (scope, card-reader, printer, tape, disk, etc.).

d) Human decision or clock signal enabling a certain flow of control.

e) Flow of control.

f) Program call.

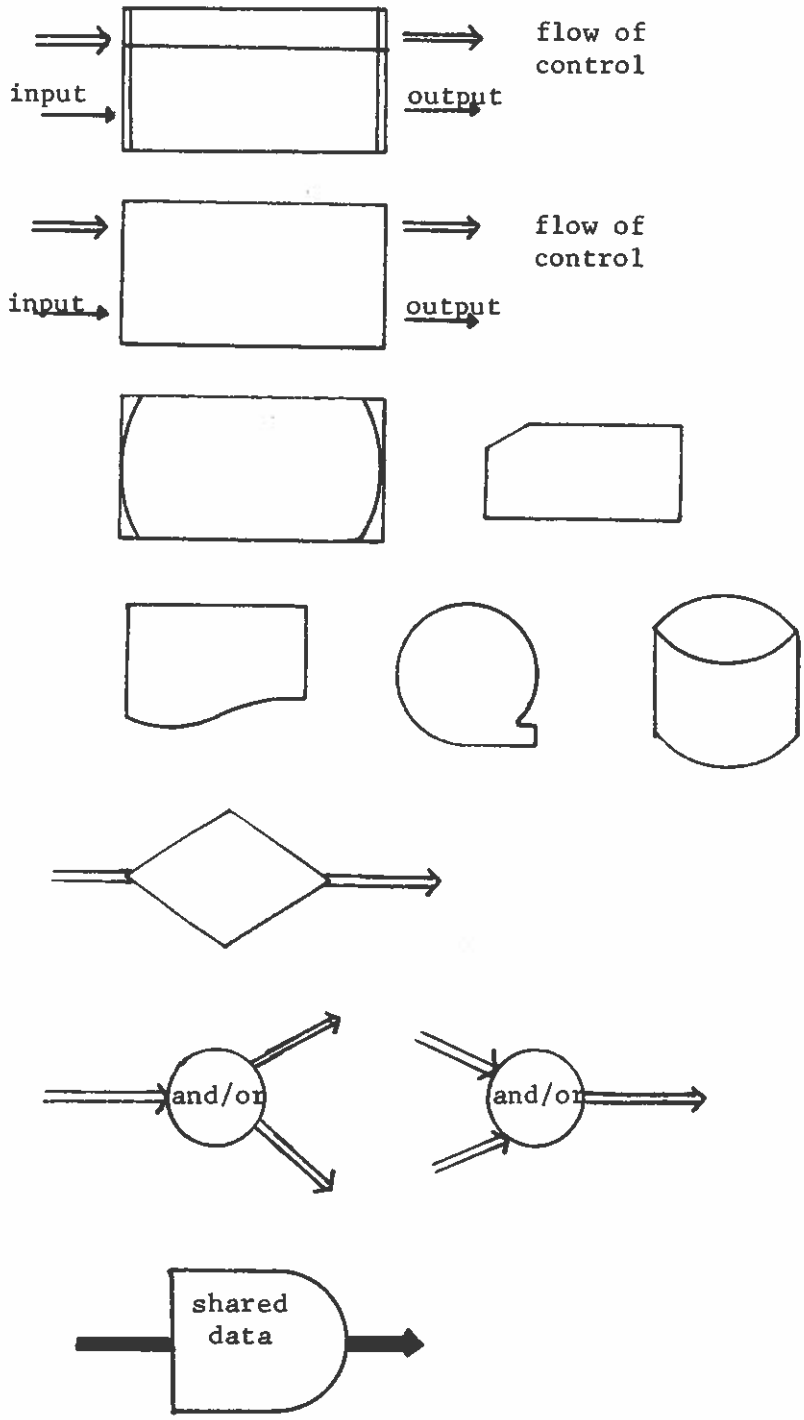
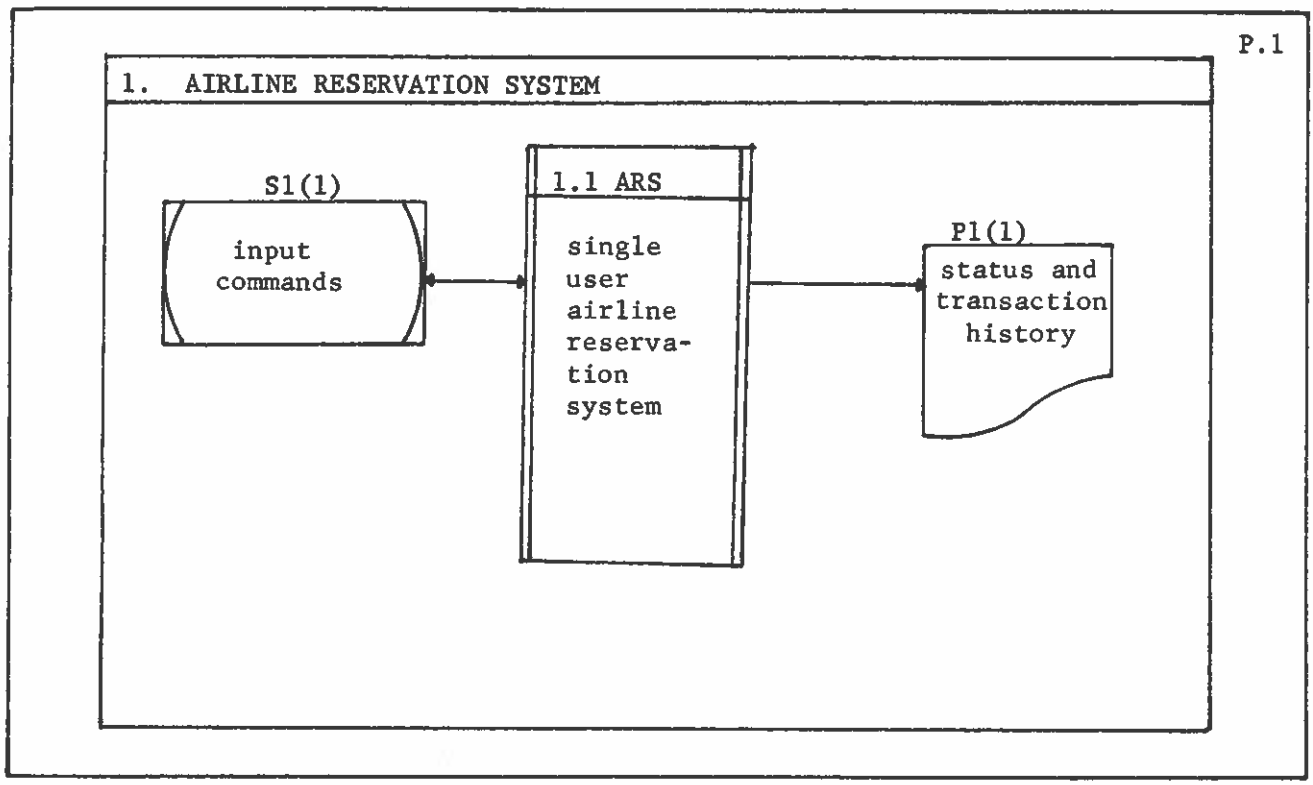


Figure 3—Flow diagrams symbolism.

- The growth should not severely impact the performance of the system.
- Growth should be achieved primarily by creating or implementing new *outgoing* interfaces.
- The number of "not yet implemented" incoming interfaces should be minimal.
- The core system should implement several key functions which would allow for its early installation and/or convincing demonstrations.

When the *program design* stage is entered, the various programs that make up the system have already been identified and the detailed design of their inputs and outputs has been completed. During this stage the selection of the major



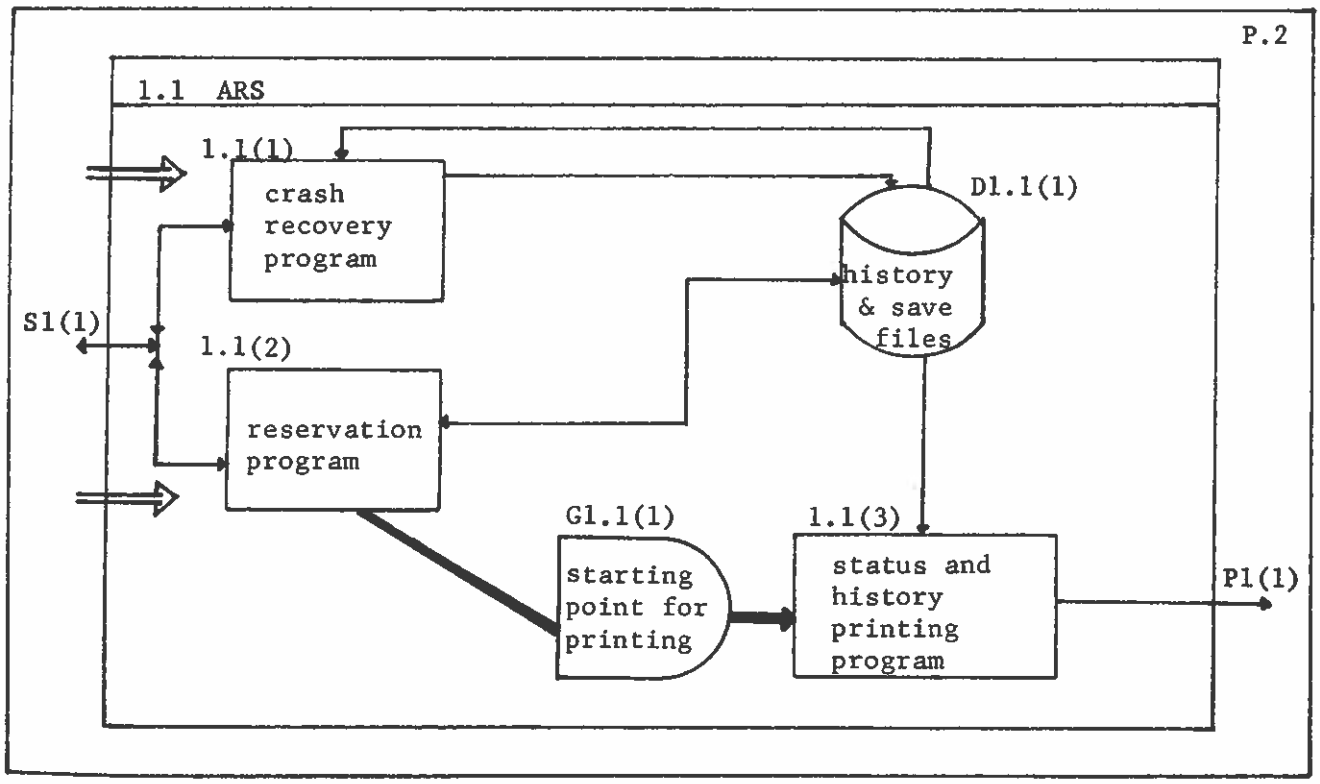


Figure 4—System architecture (flow diagram).

```
PROGRAM NAME:     1.1(2) RESERVE

MODULE NAME:      1.     CONTROL

REMARKS:          This is the top module of program RESERVE and is strictly
                  an initialization and driver module.

ACCESS:           COMMAND, WAITLIST, RSVLIST.  (These data structures are
                  initialized here.)

KEYWORDS:         None.

INPUT ASSERTION:  COMMAND, WAITLIST, AND RSVLIST are initially empty.  The
                  file "SAVE" indicates which names have reservations and
                  which ones are waiting.  The "HISTORY" file is empty.

Load WAITLIST and RSVLIST from the "SAVE" file.
ASSERTION:  WAITLIST and RSVLIST have been restored to their last known values.
     DO
     Read next command into COMMAND and put it on the "HISTORY" file.
          CASE
               WHEN (print command)    INVOKE (1.1(3) PRINT).
               WHEN (reserve command)  CALL(1.1) RSV to put name in RSVLIST.
               WHEN (wait command)     CALL(1.2) WAIT to put name in WAITLIST.
               WHEN (cancel command)   CALL(1.3) CANCEL to remove name from
                                           RSVLIST and WAITLIST.
               WHEN (save command)     CALL(1.4) SAVE to save current RSVLIST
                                           and WAITLIST.
               WHEN (inquire command)  CALL(1.5) INQ to print status of the
                                           name.
               WHEN (halt command)     BREAK
               WHEN ( )                print error--command not recognized.

          ENDCASE
     ASSERTION:  A single command was fully processed.
     OD
ASSERTION:  A halt command was detected.
CALL(1.4) SAVE to save the current RSVLIST and WAITLIST in file "SAVE" and
          to clear "HISTORY."

OUTPUT ASSERTION:  The last state of the reservation and waiting lists has
                   been saved on the "SAVE" file.  The "HISTORY" file is
                   empty.
```

Figure 5—Program design (pseudocode).

program data structures and the algorithms that act upon them takes place. The program design is done top-down following generally accepted structured programming practices. Pseudocode is used to formalize the flow of control. The data structures are described separately. A program is made up of several program modules or subroutines hierarchically organized. The entry and exit points of each program module must include input and output assertions showing what is assumed to be true at the beginning of, and what should be true upon return from, the module. The assertions are important not only in understanding the algorithms but also in the process of establishing the correctness of the

design. For example, the reader is directed to Figure 5 which shows the design for the top module of a program identified in Figure 4.

Program implementation and testing is perceived as a single process that is carried out top-down in the tradition of structured programming. The program design is used as the basis for implementation and testing. If the program was carefully designed, if a good set of implementation standards was selected, and if the design was shown to be correct, then the actual implementation and testing should require little effort.

In its final form the program should exactly mirror the program design. One should be able to identify clearly the modules and the various levels of the top-down design. Each module should *at all times* be consistent with its design specifications. If, when coding, the need for changes in the design becomes apparent, the design should be altered and reverified. Only then should one proceed to recode or modify the module. Upon coding each module, one should verify that the module is in agreement with all implementation standards, and its correctness should be established by mental simulation before any testing takes place.

The *system integration* stage will not be discussed here as it is outside the scope of this paper.

VERIFICATION PROCEDURES

In the context of the methodology presented above, the term "verification" needs to be understood as meaning nothing more than a convincing demonstration that a certain formalization describes, implements, or computes the subject of that formalization, e.g., the agreement between the functional model and the user's needs. One would want to prove that the formalization is correct but, since such an approach is unrealistic at the present time, informal methods need to be used instead. The immediate objectives of the verification are to provide guidelines for the various stages of the development and to allow for effective and systematic reviews at preselected checkpoints. Ultimately, these procedures establish a unique design and error detection strategy capable of significant impact upon the overall system development productivity. This being the case, some methodological details that have been omitted in the previous section will become apparent as the verification techniques are discussed.

The number of checkpoints that one selects depends upon the personnel and the project involved. Nevertheless, there are four checkpoints that need be considered every time: (a) after the requirements definition is completed, (b) when the system architecture is entirely selected, (c) when the program design is finished, and last, (d) when the program is fully implemented. More checkpoints may be added in between, if the size of the project makes them necessary, without modification of the verification techniques. At every checkpoint one must verify that the formalization produced thus far is self-consistent and in accordance with the standards. Furthermore, one needs to establish the consistency between the formalization being reviewed (e.g., flow diagrams) and the formalization that preceded it (e.g., functional diagrams) which, in part, describes the correctness criteria. Let us next discuss each checkpoint in detail.

- *Requirements Definition Checkpoint*—At this checkpoint the verification is carried out in three distinct steps. First, the self-consistency of the functional diagrams is checked. This step includes both syntactic (form) and semantic (content) analysis. Second, one must verify the fact that the functional diagrams indeed correctly reflect the analyst's understanding of the business environment which is being modeled. Last, it is necessary to assure that the model meets the customer's approval.

Self-consistency check—It proceeds top-down starting at the root of the tree formed by the functional diagrams.

1. Start with the top level functional diagram.
2. Make sure that:
   a. The diagram is syntactically correct.
   b. One single function appears in the top diagram, call it F_o.
   c. All external inputs to the model are present and necessary.
   d. All externally visible permanent records generated by the model are present and necessary.
   e. Given the information available on the incoming arcs, the function F_o can generate the information described on the outgoing arcs.
   f. All information coming in from the external inputs is necessary.
3. Move one level down.
4. Consider the next not yet verified diagram on the current level. (Assume that (1) the diagram contains functions F_i with incoming information IN_{ip} and outgoing information OUT_{iq} and (2) the diagram is a refinement of the function F_k whose incoming and outgoing information are referred to as IN_{km} and OUT_{kn}, respectively.)
5. Make sure that:
   a. The diagram is syntactically correct and its incoming and outgoing information are in agreement with the description of the function F_k which is being refined.
   b. Given the corresponding incoming information, each function F_i can generate the corresponding outgoing information.
   c. The information provided for each function F_i is indeed necessary.
   d. Each permanent record introduced in the current diagram contains the information that will be required from it and nothing else.
   e. The current diagram is correct with respect to F_k, i.e., it is a correct refinement of the function F_k.
6. If there is a not yet verified diagram on this level, go to 4.
7. If there is a next level, go to 3.

Completeness and accuracy check—The functional diagrams are constructed based upon the information that has been obtained from the user or customer during re-

peated interviewing sessions. Therefore, it is important to determine the fact that all that information, to the extent to which it can be incorporated into the functional diagrams, was included. This may be verified by selecting, one by one, the functions identified during the interviews and searching for their presence in the flow diagrams—the top-down organization considerably reduces the search effort.

User agreement—Regardless of the correctness of the requirements definition, its value is limited unless the user's agreement is secured. Since the user may not be sophisticated enough to follow and understand the functional diagrams, a simplified document, verified to be fully consistent with the functional diagrams, may need to be created and passed on to the user for approval.

- *Case Study for the Reader.*—Let us consider a user that needs a small airline reservation system. From interviews it becomes apparent that the system is to be accessed by a single reservation clerk that can make reservations, put passengers on "wait," cancel their names from the reservation or waiting list, and request information about either one of the two lists. Furthermore, the user specifies that a written record of all transactions needs to be saved. Based upon all these data the functional diagrams of Figure 2 may be conceived. As a simple exercise, the reader may want to verify the functional diagrams by employing the first two verification procedures.

Consider, for instance, Step 2 of the first procedure:

a. Diagram 1 is correctly constructed (standard symbols are used, on each arc there is an information box, etc.).
b. The diagram contains a single function, RSVS.
c. A single external input was specified by the user, the reservation clerk.
d. The only visible permanent record is the transaction history. The reservation and waiting lists are internal to the RSVS function and need not be introduced yet.
e. Trivial.
f. All information coming in is necessary, e.g., reservation request and name to be entered on the reservation list are needed in order to reserve.

Similarly, Step 5 of the first procedure may be used to verify Diagram 1.1.

In carrying out the second procedure, one establishes the "coverage" relation between the interview data and the functional diagrams: each function identified in the interviews is covered by a subset of the functional diagrams. For instance, the canceling function is covered by function 1.1(3) from Diagram 1.1 (in conjunction with the information boxes I1(3), I1.1(3) and I1(5). In general, the coverage relation is not necessarily one to one, but the verification is easier if that is the case.

- *System Architecture Checkpoint*—The system architecture poses for the designer a much more complex and varied set of problems. While the requirements definition involved mostly the formalization of amorphous information, the conception of the system architecture is a complex creative process which must result in a product that possesses a large set of attributes, often in conflict with each other. The result is a more complex set of verification procedures, each reflecting the concern with specific system architecture viewpoints. Thus, besides establishing self-consistency, correctness with respect to the requirements definition and user approval, one must also evaluate the system architecture from the point of view of fault tolerance, hardware compatibility and anticipated performance. However, discussion will be restricted to self-consistency and correctness, which form the main subject of this paper.

Self-consistency check.

1. Start with the top-level diagram.
2. Make sure that:
   a. Only one system component is included.
   b. Each external input from the top functional diagram is covered by one or more devices.
   c. Each permanent record present in the top functional diagram is covered by some storage or output device.
   d. Any other files present in the diagram (preferably on the lower side) are created and used by the system component and are important enough as to be identified at the top level.
   e. All inputs are necessary.
   f. All outputs are required and can be produced from the available inputs.

   (NOTE: This step assures that the top-level flow diagram covers the top-level functional diagram. This relation is not necessarily true for any subsequent levels.)
3. Move one level down.
4. Consider the next not-yet-verified flow diagram from the current level. (Assume that the diagram contains the system components C_i with the inputs $INPUT_{ip}$ and outputs $OUTPUT_{iq}$ and is a refinement of component C_k with inputs $INPUT_{km}$ and outputs $OUTPUT_{kn}$.)
5. Make sure that:
   a. The diagram is syntactically correct and its inputs and outputs are in agreement with the description of the component C_k which is being refined.
   b. Given its inputs, each component C_i can compute its outputs.
   c. No unnecessary inputs are provided.
   d. Each file is created by some component and the data is all there before being used.
   e. There are no unwanted loops in the flow of control.
   f. Each component that is marked as not being further decomposed can be implemented as a single program, (based on previous experience with problems of similar complexity).
   g. Every arc that indicates a call has associated with it a description of the global data being shared between the two programs.

h. The diagram faithfully implements C_k.

6. If there is a not yet verified diagram on this level, go to 4.
7. If there is a next level, go to 3.

Correctness check—The system architecture is correct if it implements the model defined by the functional diagrams. The verification procedure will determine the coverage relation, thus indicating what was left out as unimplemented.

1. Show that all external inputs are covered by some input devices.
2. Show that each permanent record is covered by output devices, one or more files, or by being internal to some program.
3. Show that each function, regardless of its position in the functional diagrams, is covered.
4. Show that all key relationships between functions are preserved by the system architecture.

- *Case Study for the Reader*—Figure 4 does not provide enough information so as to allow for a conclusive verification of the self-consistency and correctness of the system architecture. The missing data is to be found in the narrative that must accompany each diagram and explain each item in the diagram, and in the detailed design of program interfaces (D1.1(1) and G1.1(1)). The relatively simple nature of the system described in Figure 4 allows the reader to supply the missing details. It may also be an interesting exercise to see under which assumptions the verification is successful and when it is not.

However, before going any further, some interesting facts should be pointed out. First, when trying to establish the correctness of the system, the coverage relation is bound to point out that the crash recovery program does not cover any function appearing in the functional diagrams. The justification for introducing that program is not in the problem being solved but in the means by which it is solved, the architecture. If the program had not been included, the fault-tolerance studies would have signaled the fragility of the system. Secondly, the reservation program covers more than one function, actually four, and also two permanent records. Such complicated coverage relations are by no means unusual and further point out the great distinction between functional and flow diagrams.

Lastly, let us observe more closely the connection between self-consistency and correctness. Self-consistency is a conclusive demonstration of the fact that the architecture is a viable one, could be actually implemented, and carries out the basic intent expressed in the top level functional diagram, but is not necessarily a true realization of the requirements definition. In contrast, the correctness check tries to identify the level to which the intent of the requirements definition is reproduced under the assumption of self-consistency. Therefore, one can see that self-consistency is only a stepping stone, a weaker condition to be verified first.

- *Program Design Checkpoint*—It is intended that the program design checkpoint determines the correctness of the program before the actual coding is started. The detailed design of its interfaces (accomplished during the system architecture design) in combination with a program abstract that indicates the input/output relation represent the criteria against which the program design is to be judged. The presence of assertions facilitates the verification and helps in the understanding. However, the informal (and incomplete) nature of the assertions prohibits one from carrying out a formal proof of correctness. Therefore, as in previous steps, the procedure employed is informal and does not represent any guarantee that all errors have been eliminated.

The verification ought to establish that:

1. The program design paper is in agreement with the program design standards (the pseudocode is correctly used, the format and organization standard, the level of detail adequate, etc.)
2. The data structures have been designed properly.
3. Each program module is correct with respect to its input and output assertions.
4. The interfaces between program modules have been correctly designed and provide all the data required by each module. Also, there is agreement in the communication signals.
5. The program always terminates (if termination is desired).
6. The proper relation between inputs and outputs is achieved.
7. Adequate performance is to be anticipated.
8. The overall design will result in a maintainable program.

- *Implementation Checkpoint*—In the verification scheme proposed here, the last check takes place at the time when the program is being coded and tested. This code inspection (the term seems to have caught on lately) is designed to investigate four distinct problems:

1. Compliance with the implementation standards.
2. Consistency between the code and the design paper.
3. Correctness of data structures implementation.
4. Reevaluation of the correctness problem to a new level of detail.
5. Proper trade-off between efficiency and maintainability.

Particular mention needs to be made of the fact that the process of establishing program correctness could benefit significantly from already available theoretical results. While it is doubtful that the average programmer could be asked to formally verify his product, informal use of formal techniques is bound to prove itself highly effective. Furthermore, informal correctness proving should not be very difficult to teach.

IMPLEMENTING THE VERIFICATION PROCEDURES

The existence of a well defined and systematic set of verification procedures plays a fundamental role not only in

the discovery of errors but also in the avoidance of them in the first place. The verification may be conceived of as more than a post-factum investigation, which has its own merits. It may be carried out as an integral part of the design itself. Since most verification procedures involve a top-down analysis, it is only natural for the designer to employ them as the design progresses top-down. Furthermore, familiarity with the verification procedures and the prospect of an imminent check are bound to generate questions in the designer's mind that otherwise would have passed unattended.

Like any other methodology, the proposed verification scheme is worthless if it is not strongly enforced. The methodology as a whole was developed in such a way as to assure that auditing be economically feasible and humanly possible: a uniform approach to design including great emphasis on standardization makes the documentation readable, the consistent top-down approach assures fast access to information and a good organization of the material, and the verification procedures provide uniform and relatively precise evaluation criteria. Nevertheless, a very strong commitment of the management is absolutely necessary in order to overcome the initial resistance to such a novel approach. At the same time people need to be convinced that the audit is not going to be used as a means of evaluating personnel, but rather of evaluating or improving products. If this philosophy is truly implemented, supervisors should not be part of the team auditing the products of those under their authority.

The solution that was finally adopted during the first implementation of this methodology is an audit team including one of the authors of the document being reviewed, persons familiar with the project but not involved in that particular design aspect and complete outsiders to the project. Each member of the audit team is to carry out the verification independently and to reveal his finding during group discussions following the author's oral defense of the material. The task of the audit team could be simplified significantly by providing some adequate software support that would analyze the documentation, enforce the standards, ease the search for information and its retrieval, and perform some primitive self-consistency and correctness checks. Such a system is presently under consideration.

SUMMARY AND CONCLUSIONS

A relatively straightforward methodology for software systems development augmented by a systematic error discovery strategy, a set of verification procedures, was proposed and justified. It was also argued that the informal character of the verification procedures makes them both practical and effective. The discussion emphasized the role played by the verification procedures during software development.

At the present time one large company has adopted the approach as standard practice due to the realization that the cost of carrying out the verification (especially when projects tend to extend over three to five years) represents only a very small financial investment compared with the significant benefits of early error detection. However, the reader is advised that more work needs to be done. First, the full impact of the methodology has to be established, which in turn should result in improvements to the verification procedures as well as the audit methods. Second, the development of software supporting the enforcement of the methodology and the verification process is needed to further increase their effectiveness while decreasing the time spent during auditing. Third, research needs to be directed toward the development of more formal methods that would support partially-automated consistency and correctness checks.

ACKNOWLEDGMENT

This research was supported through a contract between the Management Information and Systems (MIS) Department of the Monsanto Company and the Computer Science Department of Washington University. The author extends his thanks to David R. Wilson and Edgar J. Kline of Monsanto for reviewing the material and contributing ideas and for the efforts involved in introducing the approach to the MIS community.

REFERENCE LIST

1. Baker, F. T., "Structured Programming in a Production Programming Environment," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 241-252.
2. Blazer, R., N. Goldman and D. Wile, "Informality in Program Specifications," *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 2, March 1978, pp. 94-103.
3. Belford, P. C., A. F. Bond, D. G. Henderson and L. S. Sellers, "Specifications Key to Effective Software Development," *Proceedings of the 2nd Software Engineering Conference*, October 1976, pp. 71-79.
4. Enger, N. L., *Documentation Standards for Computer Systems*, The Technology Press, Inc., 1976.
5. Hamilton, M. and S. Zeldin, "Higher Order Software—A Methodology for Defining Software," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, March 1976, pp. 9-32.
6. Hammer, M., W. G. Howe, V. J. Kruskal and I. Wladawaky, "A Very High Level Programming Language for Data Processing Applications," *CACM*, Vol. 20, No. 11, November 1977, pp. 832-840.
7. Mills, H. D., "Syntax-Directed Documentation for PL360," *CACM*, Vol. 13, No. 4, April 1970, pp. 216-222.
8. Myers, G. J., *Software Reliability Principles and Practices*, John Wiley and Sons, 1976.
9. Roman, G.-C., "An Argument in Favor of Mechanized Software Production," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 6, November 1977, pp. 406-415.
10. Ross, D. T., "Structured Analysis(SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1976, pp. 16-34.
11. Tausworthe, R. C., *Standardized Development of Computer Software*, Prentice-Hall, 1977.
12. Teichroew, D., and E. A. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1977, pp. 41-48.
13. Willis, R. R., "DAS—An Automated System to Support Design Analysis," *Proceedings of the 3rd Software Engineering Conference*, May 1978, pp. 109-113.