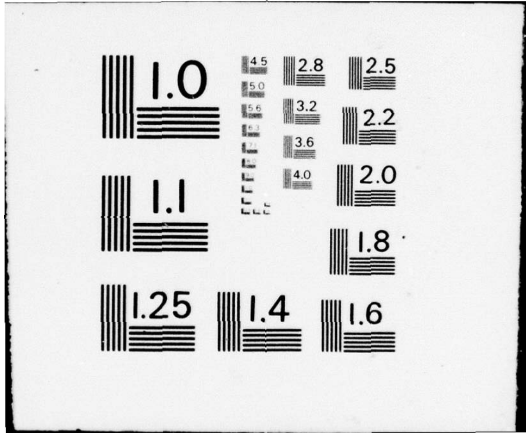
AD-A032 814 HAMMER (MICHAEL) NEWTON MASS F/G 9/2
APPROACHES TO SOLVING THE SOFTWARE PROBLEM: SOFTWARE ENGINEERIN--ETC(U)
MAY 76 M HAMMER, B LISKOV N00014-76-M-0024

UNCLASSIFIED NL


1.0
4.5
2.8
2.5
5.0
3.2
2.2
5.6
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

AD A032814

Approaches to Solving the Software Problem:

Software Engineering and Automatic Programming

Contract No. N00014-76-M-0024
Final Report
Date: 26 May 1976
Source: Michael Hammer

Michael Hammer
Barbara Liskov

DDC
RECEIVED
NOV 30 1976
B

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Table of Contents

1. Introduction 1

2. Software Engineering
  2.1 Introduction 4
  2.2 Programming Languages 6
  2.3 Specification Techniques 9
  2.4 Verification 13
    2.4.1 Testing 13
    2.4.2 Program Proving 15
  2.5 Management of Programming 17

3. Automatic Programming
  3.1 Introduction 19
  3.2 Nonprocedural Languages 20
  3.3 Psychology of Programming 23
  3.4 Program Optimization 24
  3.5 Data Management Systems 26
  3.6 Data Semantics 30
  3.7 Knowledge Based Systems 32

References 36

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION Per ltr on file
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

Foreword

This document is the final report for contract number N00014-76-M-0024 with the Office of Naval Research.

1. Introduction

It is generally recognized today that the "software problem," more than any other factor, is the principal impediment to a far wider usage of computers in many applications. What is the software problem? Briefly, it is the high cost and the poor quality of much of the software produced today. Software is expensive to construct and rarely delivered on time; as the cost of computer hardware continues to fall, software costs will increasingly dominate the economics of computer applications. Software is unreliable; it works incorrectly much of the time and sometimes fails completely. Software is unresponsive to the needs of the user, and does not adapt well to changing user demands, since it is difficult to understand and to modify. Conventional techniques, methodologies, and tools for software production have shown themselves to be inadequate for the demands of large and complex application systems. This situation is well appreciated by government, industry, and the universities, and there is a great deal of current research being conducted, whose goal is the amelioration of the software problem.

In this document, we shall endeavor to provide a broad thematic overview of proposed approaches to solving the software problem. We will survey what we feel to be the most important research being done today and also those emerging research directions that will be of prime significance in the future. Within each research area, we will indicate representative projects that are now underway.

In a sense, all computer science research may be viewed as ultimately contributing to the solution of the software problem. Every advance in the technology of computer usage, or deeper understanding of the nature of computation, will eventually bear fruit

in terms of less expensive and more reliable computer systems. But for the purposes of this survey, we shall concentrate on two areas of current research that are directly oriented to the production of better software. For convenience, we shall identify these two areas as "software engineering" and "automatic programming." These terms have been misused and overused, but we believe they roughly characterize the two most important themes in contemporary software research.

Software engineering takes as its goal the development of principles, guidelines, methodologies, and standards for the construction of computer systems (especially large and complex ones); in short, the steps needed to turn programming from a haphazard and mysterious process into a disciplined and systematic one. Research in this area can be described as developing engineering techniques, and designing software development tools (such as programming languages) that embody them. Automatic programming seeks to transfer responsibility for programming tasks from the human to the computer: in this way, the programmer's task is lessened, and the decisions that he must make are reduced in number and scope, so that the programming process becomes both easier and less error-prone. Research in this area is concerned with studying how to automate increasingly large segments of the software production process.

We expect that both software engineering and automatic programming (especially the particular research topics surveyed in this paper) will have a major impact on the software problem.

Indeed, these two approaches are complementary, although they are based on conflicting assumptions as to whether people or programs are primarily responsible for producing software. In the near future, almost all programming will continue to be done by programmers: only a few, well-understood problem domains will be supported by automatic programming systems. Consequently, the results of software engineering research can and will have major impact in the near term. In the longer term, most application domains will be eventually supported by automatic programming systems, which will lower the level of expertise needed for computer usage. However, software engineering will continue to be important for the novel applications that will inevitably arise. In addition, automatic programming systems themselves will be affected by principles of *software engineering*, both in their own structure and in the structure of the programs they produce.

2. Software Engineering

2.1 Introduction

Software engineering encompasses a number of general, domain-independent methods of coping with the software problem. The approach is predicated on the belief that programmers will continue to be responsible for the construction of software systems, and that tools and techniques can be devised that will enable them to construct programs of better quality: more reliable, understandable, modifiable and maintainable. The methods to be described are derived from the study of programming methodology, in which the process of programming and the structure of programs are studied in order to identify techniques and structures conducive to better quality software. Study of the process of programming has led to such concepts as stepwise refinement,[1,2,3] and in addition to the recognition that removal of the goto statement leads to program structures that are simpler and more understandable. Study of the structure of programs has led to the concept of modularity[4] and to the identification of the multi-procedure module as a particularly useful component in software construction.[5,6]

Programming methodology is a relatively new area of computer science, but it has already had a substantial impact on the way that programmers think about programming. This impact can be observed in the current furor over "structured programming" and in the proliferation of papers describing, for example, structured FORTRAN. Although such activity is beneficial to software quality, it is clearly not research (and much of it has missed the point: structured programming is not synonomous with goto-free programs). In this section, we will concentrate on research areas

that we believe will contribute to our understanding of the programming process and lead to further improvements in the quality of software.

The research areas to be discussed can be related by considering how an ideal method of program construction would proceed. The method is to solve problems by means of decomposition based on recognition of useful abstractions. Starting with a problem described by means of a specification, the following four steps are performed:(7)

1) identify a number of data and functional abstractions that are useful for solving the problem;
2) give a specification describing each abstraction precisely;
3) write a program that uses the abstractions to solve the original problem;
4) verify that the program really solves the problem.

After the four steps have been completed, one of the abstractions is selected as a new problem and the process is performed again. The process terminates when all abstractions not supported by programs are supported by the programming language in use. Note that although the process described above is top-down, this is not essential. Once an abstraction has been specified, it is possible to consider it as a new problem immediately, and return to steps 3 and 4 of the original problem later.

In the remainder of this section, we will discuss research aimed at supporting this process. We will discuss research in programming languages, specification techniques and program verification. We will also discuss tools and techniques for managing the program construction process.

2.2 Programming Languages

The most essential tool in program construction is the programming language: the language influences the way that the programmer thinks about problem solving, and in addition, a properly defined language can simplify the transition from program design to implementation. In this section we discuss programming language research that we believe will enhance program quality. Areas to be discussed are: data abstraction, modular programming, parallelism, control abstractions, exception handling, and program optimization.

An important way in which a programming language can contribute to the program construction process described above is by providing the proper abstraction mechanisms, so that the abstractions identified during design can be realized in a program in a natural way. Two kinds of abstractions were mentioned earlier: functional and data abstractions. Conventional programming languages provide procedures or subroutines for realizing functional abstractions, but have no analogous mechanism for supporting data abstractions. A recent significant development in programming language research has been the attention focused on the concept of a data abstraction,[8] and on linguistic constructs for supporting this concept. In this work, a data abstraction or abstract data type is viewed as consisting of a set of objects or values together with a set of operations to manipulate the objects: all uses of the objects are accomplished through invocations of the operations. The benefit gained by restricting manipulations to the operations is that implementation information need be accessible only to the operations; in this way, the separation of use from implementation that is provided by procedures for functional abstractions is also possible for data.

In order to fully realize the benefits obtainable from data abstractions, new linguistic constructs supporting abstract type definitions have been developed. These mechanisms permit a type to be implemented as a unit including a storage representation for objects of the type and algorithms for each of the type's operations. The constraint that objects of the type may only be manipulated via invocations of the type's operations can be enforced at compile time. Classes in SIMULA[9] may be used as type definitions, but do not protect the storage representation from direct manipulation by programs outside of the class. New linguistic mechanisms supporting abstract type definitions are being developed at MIT (clusters in the CLU language[10]) and at Carnegie-Mellon (forms in the Alphard language.[11]); similar mechanisms are being incorporated into many other languages.[12] Many practical problems still remain to be resolved, for example, efficient storage allocation for objects of abstract type.

The four step process described earlier is inherently modular: each abstraction (data or functional) of the design is intended to be realized by a separate program module. These modules must all be independent of one another, with explicitly defined interfaces consisting of the input and return parameters of the module. The resulting program is not block-structured; instead there is a set of program modules all existing on the same level in the sense that they are all equally usable in writing new programs. In reality, however, modules are not intended to be equally usable; for example, some modules are intended to be used by only one other

module. In addition, it is convenient in programming to take advantage of the close relationship among some modules and permit them to share data implicitly. It is unclear how modules should be grouped into programs in the absence of block structure; work on modular program construction is reported in.[13,14,15]

Most work in programming languages has concentrated on sequential constructs, while many interesting and difficult programs must take advantage of parallelism or contend with it. In general, mechanisms suggested for parallelism[16,17] are deficient in that not enough structure is provided; the programmer is given little guidance in using the mechanisms in a safe manner. Hoare's monitor construct[18] provides a synchronization mechanism with much more structure: the programmer is encouraged to limit synchronization interaction to the set of operations of a type, operating on a particular object of that type. Research on monitor-like constructs is underway at Carnegie Mellon,[19] the California Institute of Technology,[20] and M.I.T.[21] In addition, better ways of spawning and destroying processes than "parbegin" and "parend" must be developed.

Control abstractions permit the programmer to abstract away from control details. One type of control abstraction that appears to be useful is the iterator, which iterates over all the values in a collection in some specified order. The user of the iterator is not concerned with the details of the iteration; instead he makes use of a generalized loop construct. Iterators are present in SETL[22] for sets: new mechanisms permitting their use for user-defined types are being defined at Carnegie-Mellon[11], M.I.T. and Los Alamos.

There is growing recognition that a program must do more than produce correct output given well-behaved input; reasonable behavior in the face of erroneous input or machine failures (e.g., parity errors) is also important. Exception handling mechanisms support the development of such well-behaved programs. Work in this area is reported in [23,24].

Research in program optimization techniques to provide efficient implementation of domain-specific, very high-level languages is discussed in Section 3 of this report. The techniques described there are important for general-purpose languages as well. The programmer's job is to produce a well-structured and correct program, but, of course, the program must also execute with acceptable efficiency. If the compiler is capable of transforming a well-structured program into an efficient program, then the programmer need not perform the transformation himself, and the complexity of his task is reduced.

2.3 Specification Techniques

A specification of a program is a description, in some well-defined, formal language, of the behavior on which a user of the program can depend. The program itself is a specification under this definition; however, a specification in a different language is desirable for several reasons. Hopefully, a specification language permits a shorter description that states what the program does without saying how the task is accomplished (see [25] for a discussion of desirable properties of specification techniques). In addition, it is intended in the methodology described above that a

specification for an abstraction be given in advance of the implementation; the specification not only states the properties to be relied on by the user of the abstraction, but also the properties to be provided when the abstraction is implemented. Finally, the specification is an independent statement of the meaning of the abstraction that can be used for verification of both uses and implementations of the abstraction.

Specifications must be given for both functional and data abstractions. For functional abstractions, input/output specifications that describe properties of the program state are given for each procedure; the input specification describes properties of the state holding prior to procedure execution, while the output specification describes properties holding after execution has completed. Functional specifications are no more than Floyd inductive assertions[26] attached to an entire procedure. Hoare has studied how functional specifications are used in proofs.[27]

Specifications for data abstractions are not as well understood as are specifications for functional abstractions (see [25] for a recent survey of this material). Input/output specifications are not necessarily the method of choice here. Instead methods that describe the abstraction as a whole in terms of interactions among the operations are preferable. Three techniques are currently under study: abstract model, state machine,[28] and algebraic.[29,30] None of these techniques has been fully defined as yet. The main work remaining to be done is to define precise semantics for each method and to develop proof techniques that make use of them. These questions are being studied at SRI[31], ISI, Carnegie-Mellon[11].

and at M.I.T. In addition, the utility of the techniques must be studied by using them to specify complex data abstractions; only the state machine technique has been applied to reasonably large problems so far.[32]

In addition to techniques for specifying data and functional abstractions, work is needed in specifying the behavior of parallel programs. Here, the most promising approaches associate the specifications with program units similar to modules supporting data abstractions.[19,33,34] Parallel program specifications are still in a rather rudimentary state. Work in this area is likely to impact the choice of linguistic constructs to support parallel programming.

There are two difficult problems associated with specifications. First, the specifications describe functional behavior, but many interesting procedures have side effects. Non-functional behavior is usually modelled by extra assignments; this technique works as long as no sharing of data is taking place. Unfortunately, sharing of data is useful in many applications. Some work in this area is reported in[34].

The second problem with specifications is the sheer complexity of the tasks being specified. The difficulty is such that no real task has ever been fully and formally specified. Even for fairly small programs, it is necessary to invent abstractions (subsidiary definitions) in order to write an intelligible specification. Inventing the appropriate abstractions for specifying a large program clearly requires considerable intellectual effort.

It is not known at present how well the techniques will scale up to large programs. The techniques are appropriate for describing the meaning of individual program modules. To the extent that the meaning of an entire system can be expressed in terms of the meaning of a collection of modules, the techniques can be used to specify the entire system as well. Thus, the techniques may be useful for describing the design of actual systems. It is unclear whether the techniques are also useful for requirements specifications; research is needed in this area.

Certain extensions to specification techniques are needed if the techniques are to be applied to real systems. The behavior described by the specifications must be defined even when problems arise (poorly formed input, machine malfunction); most specification techniques are deficient in this respect. Moreover, in addition to specifying behavior, constraints on program performance must also be specified. Very little has been done in this area; an understanding of how the performance of a module is related to the performance of the modules it uses is needed.

Since specifications are to be written in formal languages with well-defined semantics, it is clearly possible to write programs that "understand" specifications. There are several uses for such programs. A specification is intended to describe a concept which exists in someone's head, and, of course, it is possible that the specification does not describe the concept correctly. Properties of specifications are being defined, (especially for specifications of data abstractions[29,30]) that can be checked automatically; if the properties do not hold, the specification is probably

incorrect. In addition, a specification can be made to run; a formal specification is a program, albeit a very inefficient one. By running the specification, the system can give the programmer additional information about what the specification means. A running specification can be used to simulate the behavior of a lower level abstraction, so that a module using an abstraction can be run in advance of the existence of the implementation of the abstraction. Finally, it is even possible that a specification can be automatically translated into an efficient implementation, although the translation procedure must be extremely clever;[35] there is an obvious connection between this work and automatic programming (discussed in Section 3).

2.4 Verification

Verification is the activity of demonstrating that a program performs in accordance with its specification. Note that verification does not establish the correctness of the program; it merely establishes the consistency of the program with the specification. If verification fails, either the specification or the program may be in error. If verification succeeds, our confidence that the program really does what it is supposed to do depends on our confidence in the verification process itself, and on our confidence that the specification really describes the desired behavior.

2.4.1 Testing

Traditionally, the method of establishing that a program performs as desired is to test the program by running it on a number of test cases. Testing is often divided into two activities: module testing or debugging, and system integration and testing. In module testing, single modules, or small groups of modules, are tested using tests that take into account the internal structure of the modules. Then the independently tested modules are combined

and the combination tested to see how they perform together; at this stage the internal structure of the individual modules is ignored as much as possible.

This traditional method is in ill-repute nowadays; not only is it extremely costly, but it is also ineffective: it is commonplace that many errors remain in programs that have undergone the testing process. Actually, the method is not inherently evil in itself; the problem is that it is performed so haphazardly. First of all, rarely are there specifications stating with any precision what program modules are supposed to do; debugging is based on the programmer's intuitive understanding of the required behavior. It is no wonder that severe difficulties arise at system integration: the likelihood that both the user of a module and its implementer agree in their understanding of what the module does is quite small.

Dijkstra has stated that "program testing can be used to show the presence of bugs, but never to show their absence."(36) In fact, testing could be used to show the absence of bugs if it were accompanied by an analysis showing that all meaningful cases had been tested. However, such an analysis may be fully as difficult as a proof that the program is consistent with its specification. Note that the analysis cannot be based only on the external specification of a module; the structure of the module must also be taken into account. In program proving, both external behavior and program structure have a natural role in the proof process.

A certain amount of work has been done on automatic system testing in which the system is run on large numbers of test cases and the testing system keeps track of which program paths are exercised by the tests. The existence of unexercised paths at the end of the test indicates that the tests are inadequate (or that some parts of the program cannot be reached). If all paths are exercised, however, this does not mean the program works correctly (for example, a loop may fail on the second iteration); again, an analysis is needed to establish that the testing is adequate. In the absence of the analysis, the tests can increase confidence in the program, so this kind of testing can be valuable. When the test cases are developed by a different group of people than the system implementers, the cases constitute a second, independent test of the program. Also, the test cases (with some modification) can be run after a system modification to detect unwanted changes in system behavior. However, when program proving becomes a practical method, it will be preferable to testing.

2.4.2 Program Proving

Program proving has recently become an area of practical interest because of advances in programming methodology. What is particularly important is the modular nature of well-structured programs. Programs produced by the four-step process described earlier will be composed of a hierarchy of modules, one for each abstraction. The verification of such a program can proceed in a hierarchical fashion. Each module is proved independently of the others; the theorem to be proved is that the module meets its specification. In doing this proof, specifications of the lower level modules are used as axioms, as are the properties of constructs in the programming language. Stating the theorems and axioms remains relatively simple, since the specifications describe abstractions.

It is clear that if proofs are to be given about programs of practical size, then mechanical aids are needed to cope with the large amount of detail involved. By focusing on well-structured programs, machine-aided proofs of realistic programs will be feasible in the near future. The proofs of individual modules are of a size that existing automatic program provers can manage, and an entire program can be verified by proving each of its modules. In unstructured programs without a good decomposition, all parts of the program can interact with each other in complicated ways; modular decomposition of the proof is not possible, and the complexity and size of the proof is unmanageable.

Research in automatic verification systems is in progress at ISI[37], SRI[28], Stanford[39], IBM[40], and Edinburgh[41]; a survey of this work may be found in [42]. Many problems remain to be solved before these systems can be used to prove practical programs. It is particularly important that the systems be extended to support hierarchical program proofs. Some work has been done in using the specifications of a functional abstraction in proving a program using that abstraction; work on using specifications of data abstractions is just beginning. Ultimately, proofs should be carried out in the context of a data base containing information about previously proved theorems and lemmas, so that hierarchical proofs can be performed as easily as possible.

2.5 Management of Programming

There are very serious issues involved in organizing and administering a group of skilled but individualistic programmers in the production of a large and complex software system; the challenge is to ensure that their individual products will be available on time, and can then be combined into a unified whole. There are a number of directions from which this problem is being attacked: formulation of organizational structures that contribute to and enforce good programming practice; study of the psychological and behavioral mechanisms that are at work in the programming process; and development of automated tools to aid in the organization and management of a software production effort.

Over the years, there have been numerous ad hoc managerial techniques developed in the data processing context. But it was only with the advent of structured programming and software engineering that some principles were adumbrated for the organizational structure of a programming team. The prime mover in this effort has been IBM, with their development and espousal of "improved programming technologies."

There is a computer folk law that the structure of a program reflects the structure of the organization that produced it. This concept is reflected in the idea of chief programmer teams, an organizational structure that encourages the use of top-down design and structured programming techniques.[43] In this approach, the chief programmer focuses on the design of the top level modules, delegating to his subordinates the detailed tasks of implementing lower level modules. This organization has demonstrated a significant degree of success, and efforts continue on its refinement and extension.

The question of the effect of human factors on software development has only recently begun to be seriously studied. We shall consider in Section 3 relationships between programming languages and human cognitive processes. But there have also been studies of the motivational factors underlying programming performance and the human behavioral aspects of the programming activity.[44] Methodologies and concepts are only now being formulated in this field, and its full application is still somewhat in the future.

There are a number of groups working on automated or semi-automated aids to the program development process. For example, IBM's Program Development Libraries are computerized records of the various stages of system completion. One of the most ambitious of these efforts is being conducted by the ISDOS group at the University of Michigan[45]. Though this project has other goals as well, it can be viewed as a system that maintains an on-line data base about each of the various modules of the system under construction: its function, who is responsible for it, its sources of input, the destinations of its outputs, and the like; in addition, ISDOS provides a unified data dictionary and directory for the entire system. Use of such a system provides a manager with an effective tool for coping with the complexity of a large-scale project.

We can expect to see more systems along the lines of ISDOS in the future. A worthwhile project would be to develop a system in which the programming language, the specification language, and a verification system are made available to the programmer in a unified manner. Such a system would be very useful in following the four-step program development process described earlier.

3. Automatic Programming

3.1 Introduction

Automatic programming research seeks to lessen the cost and complexity of software construction by limiting the decisions that need to be made by people during the programming process. The goal of this work is to reduce the number of decisions that a programmer must make, and confine even these to issues that he presumably understands well. The intent is to allow the programmer to concentrate (in greater or lesser degreee) on his problem domain and on problem solving in its terms, rather than on computer-like data and algorithmic structures. The level of man-machine interaction is thus moved away from the machine and closer to the person, and the responsibility for making decisions on matters that relate only to the computer are delegated to the computer itself. If the programmer's task is made easier, he should be able to perform it more quickly and reliably. In addition, it is possible that the computer, which is adapted to dealing with large-scale complexity, will be able to make many programming decisions better than people customarily do.

The assumption of programming responsibility by a computer is made possible by restricting the scope of the programming effort to an area that is well-understood and consequently subject to automation. The premise of automatic programming is that extensive programming practice in a particular area develops human understanding of relevant programming techniques in that domain

to the point where they can be automated. Conventional programming language compilers were one of the first applications of this idea, building on much accumulated expertise in assembly language programming; later, compiler-compilers were able to automate the compiler building process, once that task was well understood. In this section, we shall survey three research areas that are based on the automation of human decision-making: nonprocedural languages, data management systems, and knowledge-based systems.

3.2 Nonprocedural Languages

One proposed way of removing decision-making from the shoulders of the programmer is by having him use so-called very high-level (VHL) or nonprocedural programming languages.[1] The precise meaning of this term has been the subject of much debate and remains somewhat imprecise. But the basic idea is clear: a VHL language enables the programmer to express his problem in terms that are meaningful to him and relevant to his application domain; it requires less specification of computer-related details than does a conventional programming language. Thus, the programmer is free to concentrate on an abstract view of the problem and his algorithmic solution to it, while ignoring irrelevant details. This, of course, is a relative definition; and, in fact, FORTRAN may be regarded as nonprocedural when compared with assembly language. Current research, however, is concerned with languages that are nonprocedural with respect to FORTRAN or PL/I. The belief is that using such languages will be significantly easier than programming in conventional languages, and so will result in less expensive and more reliable software.[2] A secondary effect should be the lowering of the expertise level needed to effectively utilize computers.[3]

The procedural nature of conventional programming languages stems from the resemblance of their data structuring facilities and underlying computational models to the capabilities supported by conventional von Neumann architecture machines. Nonprocedural languages embody various features that reject this model. Some of the more important of these are: aggregate data structures, such as sets and relations, together with facilities to readily manipulate them; associative referencing, with retrieval of data afforded by content rather than context (location); pattern-matching facilities; very powerful, data-oriented control structures, such as implicit loops, backtracking, and data flow specifications. (4) The underlying argument is that features such as these are more natural for the expression of algorithms than those provided by conventional languages. We observe that in all these cases the representation of the high-level concepts in terms of conventional machine structures is not immediate, and requires decision-making in order to choose and specify an implementation. The conversion from abstract computational structures to those that resemble machine features, normally done by the programmer, is here performed by the language compiler.

Research in nonprocedural languages has two goals: to design languages that are truly easy to use; and to construct implementations of these languages that are acceptably efficient, that are able to specify the detailed control abdicated by the programmer. In both of these areas, the tradeoff between generality on the one hand, and efficiency and simplicity on the other, looms very large. A language that is applicable in many contexts is unlikely to be finely tuned for use or performance in any of them. Consequently, the non-procedural languages that have been and are being proposed are usually of prime applicability to just one problem area. Such a language possesses just the right set of data structures, operators, and control structures appropriate for the natural expression of algorithms in the domain for which it was designed; in addition, the compiler for the language can "know" about good implementation techniques for this problem area. The point is that if we are to automate decision-making in the realm of programming, it must be for decisions that we thoroughly understand; and at this time, our knowledge and experience of programming style and implementation techniques is sufficiently highly developed only within the prescribed limits of various individual problem domains.

Most nonprocedural languages to date have been intended for one of two problem areas: business data processing or artificial intelligence.(5) Examples of such languages are SSL,(6) BDL,(7) PLANNER,(8) SAIL,(9) and QA4.(10) One of the earliest nonprocedural languages was SETL,(11) ostensibly a general-purpose language, but actually best-suited for applications of combinatorial mathematics. There are more recent efforts to design languages for other areas, such as mechanical assembly.(12) As expertise is developed in desirable programming styles for other application domains, new languages for these areas, which incorporate this knowledge, will be designed.

3.3 Psychology of Programming

To date, most language design efforts (including those for non-procedural languages) have been based on the intuition and experience of the design team, with respect to what makes programs easy or hard to write and read. A new research movement in the psychology of computer programming is attempting to put language design on a more scientific basis. The goals of this work are to identify the inherent computational models that people naturally possess and thereby determine how various linguistic features contribute to the difficulty of programming. Computer scientists and psychologists are working towards developing experimental methodologies for the design and evaluation of programming languages.[13,14] A number of results have already come out of this work; for example, the data base access language SEQUEL has gone through several redesigns, principally on the basis of the results of its psychological evaluation.[15,16,17] The center for this type of work is currently IBM Research (Yorktown), with efforts also underway at Toronto, USC, and Indiana. Though still at a very early stage of development, this kind of research will undoubtedly grow as nonprocedural languages are required and designed for more fields. One of the long-range goals of this work is to reduce the amount of training and expertise needed to write programs, and thereby make computers more accessible to end users (as distinct from specialists in programming).

3.4 Program Optimization

Research in program optimization belongs in a survey of work that is attempting to solve the software problem because the successful design and use of easy-to-use systems (such as non-procedural languages) is entirely predicated on the development of advanced optimization technologies that can turn high-level specifications into efficient machine code. A bad implementation strategy for a non-procedural language can lead to a possibly exponential degradation of performance when compared with conventional coding; some loss of run-time efficiency can be tolerated because of gains in other parts of the system life-cycle, but it should be kept within a (small) linear factor. Therefore, we shall need optimizers that can analyze a nonprocedural program, glean high-level information from it, and then synthesize a near-optimal implementation for it.

There is a long history of research in code optimization for conventional programming languages, and some contemporary research is an outgrowth of these earlier efforts. The work of Allen's group at IBM Research has largely been done in the context of PL/I, but has been instrumental in the development of innovative analytic methods for extracting information from program text. They are now looking at novel structures for optimizers, especially for application to the non-procedural environment.[18] Similarly, Cheatham's group at Harvard has been focusing on the optimization of ECL, a LISP-like language.

The largest amount of work on nonprocedural optimization has been done on APL. The popularity of this language, and the potential inefficiency of its implementation, have motivated several groups to attempt its compilation. Much of this research has concentrated on using algebraic properties of the APL operators to effect transformations on the program text.[19] More recent efforts focus on global restructuring of the specified algorithm. Perlis' group at Yale, and the ALTO group at Xerox PARC, have considered co-routine implementations for arrays and implicit loops, as well as other large scale transformations.[20,21]

The SETL group at NYU has for several years been studying the problem of achieving an efficient implementation of that set-oriented language. They have been particularly concerned with determining facts about the data structures of a program in order to choose good representations for them.[22] Low and Rovner at Rochester have been working on a similar problem in the context of the SAIL language.[23]

The Protosystem I group at M.I.T. is developing an efficient implementation of the data processing language SSL.[24] Their optimizer is of interest for two reasons. It makes use of various heuristic techniques for performing global optimizations (involving such issues as file design and aggregation of computations), whose complete analysis would entail unacceptable combinatorial explosion. And it utilizes novel sources of information, other than the static program text, in making decisions; specifically, it refers to a data

base of environmental information provided by the user, and may explicitly query the user for data not provided therein. These two themes will be of increasing importance in the future: global optimization using heuristics; and the gathering and use of dynamic information (such as typical values of variables, frequency of execution of various program segments, and the like). One important source for information will be previous execution histories of the program in question. A program may well be recompiled several times, on each occasion using information gathered from earlier execution histories.

3.5 Data Management Systems

One application area where a great deal of current research is directed towards the automation of human decision-making is in the field of data base management. There is a large class of computer applications that are characterized by a large, centralized collection of data that is used by many different application programs. Decisions must be made in designing these data bases, and programs written to access, maintain, and update them. Opportunities for error are present in all these cases because of the difficulty of performing these tasks.

In a conventional data management system, a great amount of responsibility lies on the shoulders of the data base administrator and of the application programmer who uses the data base. The DBA must design the data base, choosing a logical organization for the information of the enterprise; this logical organization must then be expressed in terms of the particular data structuring facilities

that the data base system provides for modelling purposes. The data base user in turn must be familiar with this structure and write his programs to "navigate" through it, locate the data items of interest to him, and perform the appropriate computations on them. In addition, someone must write programs to examine all new data for the data base to determine correctness and validity. Conventional systems only assume from the programmer the responsibility for knowing, using, and maintaining the physical representations of the data structures that comprise the data base. Though this approach does represent a major improvement over earlier file systems, it still leaves the responsibility for a great deal of low level decision-making in the hands of people. Current research seeks to automate many of these tasks, and lead to more reliable and easier to use data base systems.

Probably the most unified approach to these problems is being conducted in the context of relational data bases.[25] The relational model of data proposes a single, very simple and uniform data structure with which to model the user's logical view of his data base. This logical view is entirely divorced from any aspects of the data base's physical organization; the responsibility for choosing representational mechanisms and access structures for the data is transferred from the DBA to the system. Similarly, the application programmer is presented with a much simpler view of the data, since he transacts with the logical version of the data and not its physical representation.

Virtually all of the information in a relational data base is carried in data values rather than in data structures. Such a data base lends itself to access by means of nonprocedural techniques, especially associative referencing. There have been a number of nonprocedural languages proposed for accessing relational data bases.[26,27] These languages are intended both to provide easier access to data bases for programmers writing complex programs in some conventional programming language, and to enable direct access to data by non-programmers. Research is being conducted on making such languages so easy to learn and use that in many cases the need for writing "programs" would be obviated.

Thus, the relational model of data is a "higher-level" data model, providing a less machine-like view of data than those afforded by conventional systems, and, therefore, easing the tasks of both the DBA and the user. The superiority of this approach for many applications is becoming widely accepted. But again, the critical question is how the kinds of decisions normally made by people can be competitively made by the data base system. The issues in this context are the choice of physical representations for the (logical) relations, the selection of auxiliary structures to provide better access to these storage structures, and the implementation of the non-procedural queries put against the data base. In a conventional environment, these decisions are all made by people. There are several research groups working on the technology of relational data base implementation, attempting to put the needed "intelligence" into a data management system. The largest of these efforts

is the System R project at IBM Research in San Jose.[28] Although they are quick to disclaim any official product implications for their work, this group is making a major attempt to develop a relational system that can be competitive with conventional systems on large-scale data bases. Other efforts in this area include the INGRES system at Berkeley, PRTV at IBM United Kingdom, and ZETA at the University of Toronto.[29,30,31]

The approach taken by the above-mentioned groups is that of dynamic optimization of non-procedural accesses with respect to static representational structures. That is, a representational scheme is chosen for a relation at the time of its definition; the system exercises "intelligence" in using this structure in the most effective way to respond to queries. Future research will also include intelligence in the adaptive selection of physical representations based on the pattern of use of the data; some work is being done on this now at M.I.T. and SRI.[32,33] There is also preliminary interest in the design of novel computer architectures, not based on the von Neumann model, to provide hardware support for relational data bases; the major concept here is to enable parallel associative access of the data base, by means of distributed logic, placing a microprocessor on each head of a disk. Prototype systems are being developed at Toronto, Utah and Florida.[34]

3.6 Data Semantics

A new wave of research in data management, currently getting under way, seeks to further automate various design decisions associated with the data base, and make it even easier for non-experts to interact with it. The common theme running through this work is the concept of data semantics; that is, the data management system is to possess an "understanding" and "knowledge" of the meaning of a data base, as well as of its structure, and utilize this information to perform tasks usually done by people.[35]

The most widely discussed application of data semantics is the provision of various aids to the DBA to assist him in coping with the complexity of a large-scale data base, with respect to its design, maintenance, use, and reorganization. (This idea motivated the proposal for a conceptual schema level of description for data bases, by the ANSI/SPARC Committee on Data Base Management Systems.)[36] The simplest such application would be a "semantic data dictionary," which would provide the DBA with a map not only of the data base's structure, but of the meaningful relationship among its components. The DBA would use this map in organizing (and reorganizing) the data base. Such a dictionary would also be useful in resolving user queries, and could eventually enable user interface languages that would not demand that the user know even the logical structure of his data base, but rather allow him to express his queries in terms semantically meaningful to him. Such a language could, in turn, provide a needed intermediate level of support for natural language access to a data base.

A more ambitious application of data semantics is an automatic data base designer, which translates specifications of the semantic characteristics of the application into recommended logical (and physical) data structures for the data base representation. Similarly, such semantic information could be used for intelligent optimization and access path selection for user queries. Another use of data semantics would be in semantic integrity checking, using knowledge of the data base's meaning to detect and correct erroneous data being submitted to it.

Research is focusing on developing the representational mechanisms needed to express the semantics of data, and designing the systems that apply semantics to the above problems. One approach is that of higher-level data models, ways of describing data not in terms of representational data structures (even simple ones like relations), but using conceptual primitives such as entity, attribute, and relationship. The DIAM model of Senko at IBM Research is an example of such work.[37] A similar effort, which focuses on the semantics of relational data bases, is being conducted by Codd at IBM, while a group at Toronto is using semantic networks for representing the meaning of a data base.[38] Semantic descriptions of data bases are being used for automatic data base design at the University of Michigan; IBM has announced a product, based on similar principles but much more primitive, for generating an IMS data base from a semantic description.[39,40]

Constraints (logical predicates that define legal states of a data base or legitimate changes to it) are being used to express

the semantic information needed for data integrity checking. Prototype systems exist at Berkeley and IBM Research, and research is continuing there as well as at M.I.T.[41,42]

3.7 Knowledge Based Systems

The research described above seeks to lessen the duties of the programmer by automating many of the decisions that he would otherwise have to make. The "knowledge" that these systems use in making decisions is derived from extensive human experience in the field in question: when people really understand how they make decisions, they are able to program computers to make those decisions for them. However, the approaches we have discussed so far are intrinsically viable only for limited domains; that is, "programming expertise" for a particular area is deeply but implicitly embedded in a programming system for that area, with the result that its generality is strictly limited. Furthermore, those aspects of programming that the foregoing work seeks to automate are by and large restricted to "coding": the production of algorithmic programming language code from a precise (though less procedural) specification of the algorithm to be implemented.

There is another theme in automatic programming research, the so-called "high-road", that seeks to understand and automate the entire human intellectual endeavor called programming, beginning with the early stages of problem acquisition and strategy formulation.[43] The goal is to have the computer assume a dominant share of responsibility in the production of software. This work is

closely allied to research in artificial intelligence in its quest for an understanding and simulation of human cognitive processes and problem solving. One distinguishing aspect of most of this research is that it utilizes an explicit "knowledge base," which contains information about some application domain or about programming in general, and uses this base to generate, understand, or verify programs. In addition, the scope and range of these efforts are considerably more general than those that we have discussed above. In these "knowledge based" systems, universal programming mechanisms applicable to any problem domain are utilized, resulting in a high degree of generality. Such an understanding of the programming process can be used in a variety of advanced applications.

One of the most ambitious applications of knowledge based systems is the generation of a program from a high level, declarative specification of its intent, which is devoid of any algorithmic information whatever. In order to accomplish this, a system must possess a semantic knowledge base, expressing information about the application domain in question. The system must also have an understanding of the programming constructs available to it, and their utility in expressing algorithms for the domain. The system first acquires the specifications of the desired program, relates them to its model of the domain, and produces a strategy for the program structure; it then uses its programming knowledge to synthesize this program using available constructs. This approach is much

more general than the work on nonprocedural languages because of the enormous distance of the program specifications from machine-like algorithmic structures, as well as the applicability of these results to many different problem areas. In general, this research is more speculative, with a longer time-scale and a potentially greater payoff, than what we have discussed above. Its success will entail not only the automation of computer-related aspects of programming, but also of the creative aspects of heuristic problem-solving that precede algorithm formulation.

There are many subproblems that must be addressed in such research. Principal among these is the question of the representation of knowledge, how the information about the application domain is to be expressed for use by the system. Another major issue is the isolation of the conceptual fundamentals of programming, breaking them down into a collection of primitives that can be used in a synthesis procedure, and then devising a system to build programs using these abstract primitives. Finally, of course, there is the problem of determining from the functional specifications provided by the user, and the base of domain dependent knowledge, a problem-solving approach to be synthesized with the available language constructs. Most current researchers address this last issue by concerning themselves with one domain at a time, which has a particular problem solving methodology that the system can use.

There are various possibilities for the medium of expression for program functional specifications. The ultimate choice would be natural language; and since many researchers feel that the knowledge needed for program generation is closely related to that needed for natural language understanding, the two efforts are often closely linked. Interim approaches involve some high-level but formal descriptive language, or a list of input-output pairs representative of the transformation the program is to effect. In any event, a dialogue between the user and the system is envisioned as being necessary, whereby the system completes the user's initially specified model by means of a series of questions and answers. Specific applications on which current research efforts are focusing include list processing programs, message distribution systems, and inventory control systems.[44,45] It should be re-emphasized that these are merely specific testbeds being used to determine general programming mechanisms and forms for their representation.

There are several other areas of research that are knowledge based in one way or another. One of these is the study of automatic program debugging.[46] Here the system uses its explicit knowledge of general programming and of the particular problem domain to detect, diagnose, and treat logical programming errors. While perhaps somewhat more tractable, this problem is clearly related to program synthesis; in fact, it has been suggested that program generation can be viewed as the debugging of an initially trivial program. A related idea is that of program explaining or program documenting systems, which can relate the intent of a program to its textual body and thus assist those who wish to understand or change it.[47]

References -- Section 2

1. Wirth, N., "Program development by stepwise refinement," Commun. ACM 14, 4 (1971), pp. 221-227.

2. Dijkstra, E. W., "Notes on Structured Programming" in Structured Programming (APIC Studies in Data Processing, No. 8, Academic Press, New York, 1972, pp. 1-81.

3. Mills, H. D., "Structured programming in large systems," Debugging Techniques in Large Systems (ed. R. Rustin), Prentice Hall Inc., Englewood, Cliffs, New Jersey, pp. 41-55.

4. Dennis, J. B., "Modularity," Computation Structures Group Memo 70, Laboratory for Computer Science, M.I.T., Cambridge, Mass., 1968.

5. Parnas, D. L., "Information distribution aspects of design methodology," in Proc. Int. Fed. Inform. Processing Congress., August, 1971.

6. Liskov, B. H., "A design methodology for reliable software systems," in 1972 Fall Joint Computer Conf., AFIPS Conf., Proc., vol 41, Montvale, N.J., AFIPS Press, 1972, pp. 191-199.

7. Good, Donald I., "Provable programming," ACM SIGPLAN Notices, 10, 6, (June 1975).

8. Liskov, B. H. and Zilles, S., "Programming with abstract data types," in Proc. Ass. Comput. Mach. Conf. Very High Level Languages, ACM SIGPLAN Notices, 9, (April 1974), pp. 50-59.

9. Dahl, O. J., and Nygaar, K., "The SIMULA 67 common base language," Norwegian Computing Center, Oslo, Publication S-22, 1970.

References -- Section 2

10. Liskov, B., "An introduction to CLU," Computation Structures Group Memo 136, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1976.

11. Wulf, W. A , London, R., and Shaw, M., "Abstractions and Verifications in Alphard," to be published.

12. Johnson, R. T. and Morris, J. B., "Abstract types in the model programming language," Proc. of the Conf. on Data: Abstraction, Definition and Structure, ACM SIGPLAN Notices 11, 2 (1976), pp. 36-46.

13. DeRemer, F., and Kron, H., "Programming-in-the-large versus programming-in-the-small," Proc. of the International Conf. on Reliable Software, ACM SIGPLAN Notices 10, 6, (1975), pp. 114-121.

14. Thomas, J., Module Interconnection in Programming Systems Supporting Abstraction, Ph.D. Thesis (forthcoming), Brown University, 1976.

15. Koster, C. H. A., "Visibility and types," Proc. of the Conf. on Data: Abstraction, Definition and Structure, ACM SIGPLAN Notices 10, 2 (1976) pp. 179-190.

16. Dijkstra, E. W., "Cooperating sequential processes," Programming Languages (Ed. F. Gennys), Academic Press, New York, 1968.

17. Brinch-Hansen, P., "Structured multiprogramming," Comm. ACM, 15, 7 (July 1972), pp. 574-577.

18. Hoare, C. A. R., "Monitors: an operating system structuring concept," Comm. ACM, 17, 10, (October 1974).

References -- Section 2

19. Habermann, A. N., "Path expressions," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., (June 1975).

20. Brinch, Hansen, P., "The programming language concurrent Pascal," IEEE Trans. on Software Engin., SE-1, 2 (1975), pp. 199-206.

21. Hewitt, C. and Atkinson, R., Synchronization in Actor Systems, to be published.

22. Schwartz, J., "On programming, an interim report on the SETL project," Department of Computer Science, Courant Inst. Math., Sci., New York Univ., New York, 1973.

23. Goodenough, J. B., "Exception handling: issues and a proposed notation," Comm. ACM 18, 12 (1975), pp. 683-696.

24. Randell, B., "System structure for software fault tolerance," IEEE Trans. on Software Engin., SE-1, 2 (1975), pp. 220-232.

25. Liskov, B., and Zilles, S. N., "Specification techniques for data abstractions," IEEE Trans. on Software Engin., SE-1, 1, (March 1975).

26. Floyd, R. W., "Assigning meanings to programs," in Proc. Symp. Applied Mathematics, vol. XIX, Mathematical Aspects of Computer Science, American Mathematical Society, Providence, R. I., 1967, pp. 19-32.

27. Hoare, C. A. R., "Procedures and parameters: an axiomatic approach," Symposium on the Semantics of Algorithmic Languages, (E. English, Ed.), Springer, Berlin-Heidleberg-New York (1971).

References -- Section 2

28. Parnas, D. L., "A technique for the specification of software modules with examples," Commun. Ass. Comput. Mach., vol 15., pp. 330-336, May 1972.

29. Zilles, S. N., "Data algebra: A specification technique for data structures," Ph.D. dissertation (forthcoming), Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1976.

30. Guttag, J., "Abstract data types and the development of data structures," to be published.

31. Robinson, L., Levitt, K. N., Neumann, P. G., Saxena, A. R., "On attaining reliable software of a secure operating system," ACM SIGPLAN Notices, Vol. 10, No. 6, Proc. International Conf. on Reliable Software, (June 1975) pp. 267-284.

32. Price, W. R., "Implications of a virtual memory mechanism for implementing protection in a family of operating systems," Department for Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1973.

33. Robinson, L., and Holt, R. C., "Formal specifications for solutions to synchronization problems," Stanford Research Institute, Computer Science Group.

34. Grief, Irene, Semantics of Communicating Parallel Processes, Laboratory for Computer Science Technical Report #154, Massachusetts Institute of Technology, 1975.

References -- Section 2

35. Okrent, H., Synthesis of Data Structures from Their Algebraic Descriptions, Ph.D. thesis (forthcoming), Massachusetts Institute of Technology, 1976.

36. Dijkstra, E. W., "Structured Programming," Software Engineering Techniques, Report on conference sponsored by the NATO Science Co. Rome, Italy (ed. J. N. Buxton and B. Randell), 1969, pp.84 - 88.

37. Good, D. I., London. R. L., and W. W. Bledsoe, "An interactive program verification system", IEEE Transactions on Software Engineering, SE-1, 1 (March 1975) pp. 59-67.

38. Boyer, R. S. and Moore, J. S., "Proving theorems about Lisp function," J. ACM, 22, 1, (January 1975), pp. 129-144.

39. Suzuki, N., "Verifying programs by algebraic and logical reduction," Proc. of International Conf. on Reliable Software, (April 1975), pp. 473-481.

40. King, J. C., "A new approach to program testing," Proc. of the First International Conf. on Reliable Software, SIGPLAN Notices, 10, 6, (June 1975), pp. 228-233.

41. Darlington, J., and Burstall, R. M., "A system which automatically improves programs," Department of Machine Intelligence, University of Edinburgh, Edinburgh U. K. (July 1974).

42. London, Ralph L., "A view of program verification," ACM SIGPLAN Notices, 10, 6, Proc. International Conf. on Reliable Software, (June 1975), pp. 534-544.

References -- Section 2

43. Baker, F. T., "Chief programmer team management of production programming," IBM Systems Journal, 2 (January 1972) pp. 56-73.

44. Weinberg, G. M., "The psychology of improved programming performance," DATAMATION, (November 1972), pp. 82-85.

45. Teichrowe, D., and Sagani, H., "Automation of system building," DATMATION, (August 1971), pp. 25-30.

References -- Section 3

1. Leavenworth, B. M. , and Sammet, J. E., "Overview of nonprocedural languages," ACM SIGPLAN Notices, 9, 4, (1974).
2. Hammer, M., "The design of usable programming languages," Proceedings of the 1975 ACM National Conference, October 1975.
3. Goldberg, P., "Structured programming for non-programmers," IBM Research Report RC5318, March 1975.
4. Leavenworth, B., "Nonprocedural programming," IBM Research Report RC4968, August 1974.
5. Bobrow, D., and Raphael, B., "New programming languages for Artificial Intelligence research," ACM Computing Surveys, 6, 3, (September 1974).
6. Ruth, G., "SSL: A language for specifying business data processing systems," unpublished memorandum, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., 1976.
7. Hammer, M., Howe, W. Kruskal, V., Wladawsky, I., "A very high level programming language for data processing applications," to appear in Comm. ACM.
8. Hewitt, C., "Procedural embedding of knowledge in PLANNER," Proceedings of the IJCAI, London, September 1971.
9. Feldman, J., et al., "Recent developments in SAIL - An ALGOL-based language for Artificial Intelligence," Proceedings of the 1972 Fall Joint Computer Conference

References -- Section 3

10. Rulifson, J. et al., "QA4, a language for writing problem solving programs," Proceedings of the 1968 IFIP Congress.

11. Schwartz, J. T., On Programming: An Interim Report on the SETL Project -- Installment I: Generalities, Computer Science Department, Courant Institute of Mathematical Sciences, New York University (1973).

12. Lieberman, L., "AUTOPASS: a very high level language for mechanical assembly systems," IBM Research Report RC5599, August 1975.

13. Shneiderman, B., "Experimental testing in programming languages," Proceedings of the 1975 National Computer Conference.

14. Miller, L., "Programming by non-programmers," IBM Research Report RC4280, October 1973.

15. Reisner, P. et al., "Human factors evaluation of two data base query languages," Proceedings of the 1975 National Computer Conference.

16. Thomas, J., and Gould, J., "A psychological study of query by example," Proceedings of the 1975 National Computer Conference.

17. Weissman, L., "Psychological complexity of computer programs," Technical Report CSRG-26, Computer Systems Research Group, University of Toronto, 1973.

18. Carter, J., "Preliminary study toward a better compiler," IBM Research Report, 1975.

References -- Section 3

19. Abrams, P., "An APL machine," Technical Report STAN-CS-70-158, Stanford University, 1970.

20. Perlis, A., "Steps Toward an APL compiler," Lecture notes from International Summer School in Structured Programming, 1973.

21. Wegbreit, B., "Goal-Directed program transformation," Proceedings of the Third ACM Symposium on Principles of Programming Languages, 1976.

22. Schwartz, J., "Automatic data structure choice in a language of very high level," Proceedings of the Second ACM Symposium on Principles of Programming Languages, 1975.

23. Low, J. and Rovner, P., "Techniques for the automatic selection of data structures," Proceedings of the Third ACM Symposium on Principles of Programming Languages, 1976.

24. Ruth, G., "Automatic design of data processing systems," Proceedings of the Third ACM Symposium on Principles of Programming Languages, 1976.

25. Codd, E. F., "A relational model of data for large shared data banks," Comm. ACM, 13, 6, (June 1970).

26. Boyce, R. F., Chamberlin, D. D., King, W. F., and Hammer, M. M., "Specifying queries as relational expressions: SQUARE," Comm. ACM, 18, 11, (November 1975).

27. Chamberlain, D. D., and Boyce, R. F., "SEQUEL: A structured English query language," Proc. ACM-SIGMOD Workshop on Data Description, Access, and Control, May 1974.

References -- Section 3

28. Astrahan, M. et al., "System R -- A relational data base management system," to appear in ACM Transactions on Data Base Systems.

29. Held, G. D., Stonebraker, M. and Wong, E., "INGRES: A relational data base system," Proceedings of the 1975 National Computer Conference.

30. Todd, S. J. P., "Peterlee relational test vehicle PRTV, a technical overview," IBM United Kingdom Scientific Center Report UKSC 0075, July 1975.

31. Mylopoulos, J., Schuster S. A., and Tsichritzis, D., "A multi-level relational system," Proceedings of the 1975 National Computer Conference.

32. Hammer, M., and Chan, A., "Index selection in a self-adaptive data base management system," Proceedings of the 1976 ACM-SIGMOD Conference.

33. Pease, M., "Self-adaptive file structure," Technical Report 8, Stanford Research Institute, 1973.

34. Ozkarahan, E. A., Schuster, S. A., and Smith, K. C., "RAP: An associative processor for data base management," Proceedings of the 1975 National Computer Conference.

35. Modelling in Data Base Management Systems, Proceedings of IFIP TC-2 Working Conference, to be published.

36. ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim Report, FDT, 7 (2), 1975.

References -- Section 3

37. Senko, M. E., "Data description language in the context of multilevel structured description," in Data Base Description, (ed. B. Dougue and G. Nijssen), North-Holland, Amsterdam, 1975.

38. Roussopoulos, N., "Using semantic networks for data base management," Proceedings of the 1975 International Conference on Very Large Data Bases.

39. Mitoma, M., "Automatic data base schema design and optimization," Proceedings of the 1975 International Conference on Very Large Data Bases.

40. Hubbard, G., "Automating logical file design," Proceedings of the 1975 International Conference on Very Large Data Bases.

41. Hammer, M. and McLeod, D., "Semantic integrity in a relational data base system," Proceedings of the 1975 International Conference on Very Large Data Bases.

42. Eswaran, K., "Functional specifications of a subsystem for data base integrity," Proceedings of the 1975 International Conference on Very Large Data Bases.

43. Balzer, R., "Automatic Programming," Technical Report, USC Information Sciences Institute, 1972.

44. Green, C. et al, "Progress report on program-understanding systems," Technical Report STAN-CS-74-44, Stanford University, 1974.

45. Martin, W., "Automatic generation of customized, model based information systems," Proceedings of the Wharton Conference on Research on Computers in Organizations, 1973.

References -- Section 3

46. Sussman, G., "A computational model of skill acquisition," Technical Report AI TR-197, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., 1973.

47. Mikelsons, M. and Wladawsky, I., "On the formal documentation of programs," IBM Research Report RC5788, January 1976.