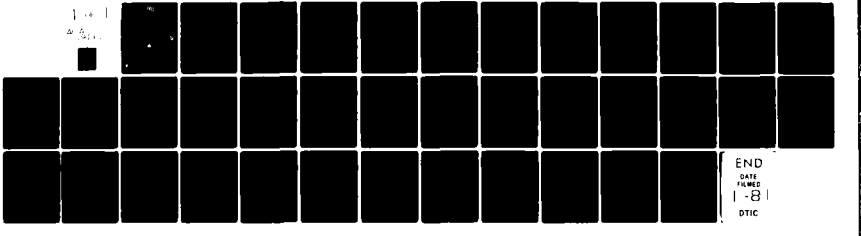
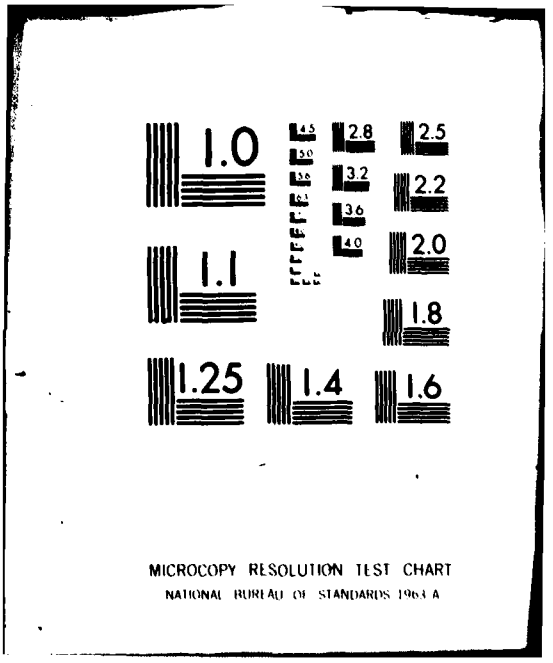
AD-A092 621 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
COMPUTER AIDED PROGRAM SYNTHESIS.(U)
JAN 80 R J WOOD N00014-76-C-0477
UNCLASSIFIED TR-861 NL



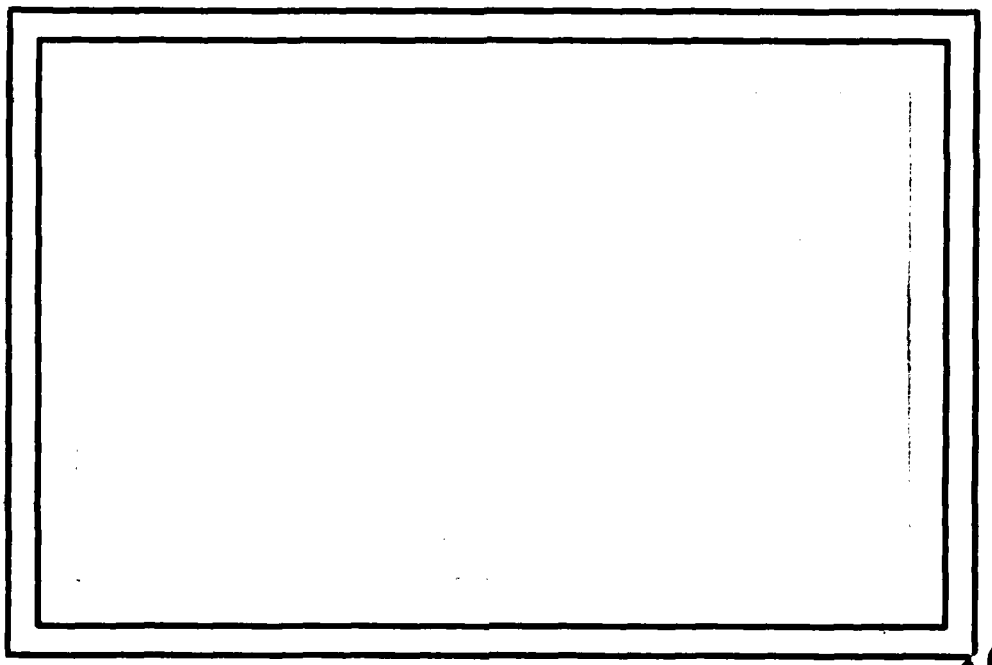
END
DATE FILMED
1-81
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

LEVEL II

1

SC



UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND
20742

TR-861
N00014-76C-0477

January 1980

COMPUTER AIDED PROGRAM SYNTHESIS

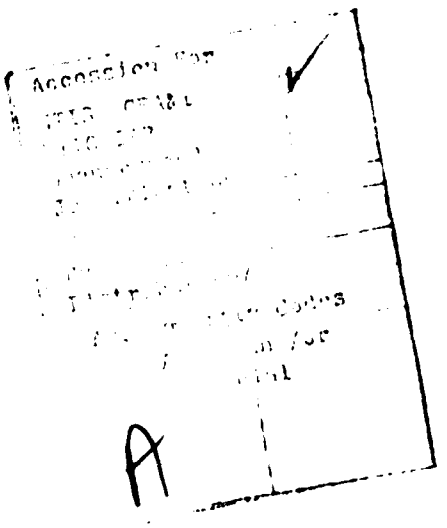
Richard J. Wood

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract: This paper proposes the construction of a programming system that will interact with a domain expert user to develop a computer program. Activities evident during a client-consultant interaction are identified and examined, and a new program structure called the Program Model (PM) is presented. The PM explicitly represents the definitions and refinements of the problem domain, the development of an algorithmic solution, and the programming knowledge used during synthesis. The types of processing required for a computer aided synthesis system are examined and a control scheme is proposed for managing non-linear program development.

The research described in this report is funded by the Office of Naval Research under grant N00014-76C-0477. Their support and encouragement are gratefully acknowledged.

CONTENTS



COMPUTER AIDED PROGRAM SYNTHESIS

Richard J. Wood

1. Introduction

Program synthesis is the task of combining the specifications of a desired goal with knowledge of algorithmic processes, design techniques, and computer languages to produce a machine executable program. This transformation from conceptual process to concrete computer program is more complex and of greater scope than once thought. The programmer not only produces code, he assimilates and refines domain specific problem descriptions, selects strategies for achieving the desired goal, verifies that the behavior of the process has been completely specified, and determines the representations for objects and actions in the problem domain. These decisions and activities are crucial to the understanding of the final program, yet are normally manifest only in the program comments, if at all.

The difficulties of constructing a program from an initial problem description and efficiently managing the vast amount of knowledge used during synthesis has long been acknowledged as a source of the high cost of programming. Solutions to this "software problem" include the development of programming systems capable of automatically performing some of the tasks previously done by human programmers. These systems incorporate a model of the synthesis processes and access a data base of programming knowledge and techniques. The success of these systems has been limited and is due, in part, to the high level of abstraction in the domains to which they are applied. This project will investigate assembly language programming on a simple machine. By examining program synthesis in the concrete and uncluttered realm of assembly language programs (as contrasted to abstract high-level languages) progress towards a successful computer aided programming system can advance in much the same manner that advances to general purpose problem solving resulted from investigation into the blocks-world domain.

This investigation will focus on a style of programming, called the client-consultant paradigm, in which a domain expert (the client) and an

adaptable programmer (the consultant) interact to formulate a program. The activities evident during this mode of synthesis will be identified and examined. A record of the processing that occurs during synthesis will be incorporated into a new program structure, called the Program Model. This integration of design activities and knowledge with executable code will yield a program structure that is unlike the sequential nature of programs.

Currently the final program form reflects more the nature of the machine that will execute the program than the thought processes that produced it. Algorithm construction utilizes problem solving techniques to reduce a task into a set of subproblems. The collective solution of the subproblems produces a solution to the overall task. The problem solver accesses a variety of knowledge sources to direct the task reduction and to generate a solution. These sources include: causal knowledge, i.e. knowledge relating actions to their effects (and side effects); hierarchical knowledge, i.e. knowledge relating the decomposition of goals to their subgoals; strategy selection knowledge, i.e. knowledge representing a context sensitive discrimination among alternate methods; and knowledge of logical dependencies among related subgoals to avoid inimical interactions. These currently omitted knowledge sources can be explicitly represented by using a graph structure produced by a problem reduction problem solver.

Such a reduction graph is a dual structure representing not only the actions executed by the problem solver, but also the preconditions, enablements, and effects of the action execution toward the achievement of an intended goal state. When this representation technique is applied to the programming domain, each program statement (i.e. action) and the conditions for its correct execution are linked to the descriptions of the statement's effects. Criteria used in state decomposition (hierarchical knowledge), contextual information employed during algorithm selection (strategy selection knowledge), and computation state descriptions used in deduction (dependency knowledge) are included in the reduction graph and provide a complete history of the program synthesis. This structure contains more information than the current listing of executable statements (i.e. just the actions) and would facilitate other post-synthesis programming activities[1].

---

[1]These activities (e.g. debugging, verification, maintenance) require an

Even with the contributions from all these various knowledge sources, the programmer cannot invent solutions for new problems without aid from a domain expert. The expert is needed to describe the allowable problem reductions, define domain specific terminology, and make the numerous mundane inferences in the problem space that are crucial to the task's solution. When the two sources, the computer expert and the domain expert, are combined, program synthesis can occur. The intent of this project is to identify and codify a subset of the techniques and knowledge sources used by a programmer, and construct a system that will facilitate the development and manipulation of the program structure. We will rely on a human source to provide domain specific expertise.

1.1. The Client Consultant Paradigm

This cooperative style of programming, the client-consultant relationship, is in common practice today. The consultant is a programmer, knowledgeable in general computer science techniques, but unfamiliar with the client's domain. The client, an expert problem solver in his domain, has recognized the need for computerization of a process. Yet he lacks the knowledge to implement the process or even to specify his requirements to a consultant. Together they interact in a cooperative manner to formulate a program, each supplying his expertise.

The client describes his task to the consultant and supplies answers and explanations to the consultant's questions. The consultant applies his knowledge of algorithms to extract process descriptions. Domain terms are identified as control constructs or data objects. Abstract actions are refined (i.e. described) in terms of better understood, more elementary actions, which are, in turn, refined until the process is completely understood. During this goal-directed phase, descriptions of data are accumulated and internal representations are selected. At some point the programmer is satisfied that a program can be written that will achieve the client's specified goal and he starts producing code.

During coding, the requirements of the external algorithms are integrated with the constraints imposed by the internal computer environment. Single

---

understanding of the algorithm's causal structure which is embodied in the program.

steps of the problem solution must be explicitly expressed in the chosen computer language. This mapping is usually one-to-many and depends on the degree of abstraction of the solution step and the actual language. Problems that arise during coding are first handled by alternate strategies available from the consultant's knowlege of the target language, and only when constraints prohibit the realization of a subgoal will the overall plan be modified.

This method of design is not new. The use of abstraction in a problem solution and its successive refinement to a specific solution has been employed in many disciplines [Simon69a]. This technique is present in computer science and is called stepwise refinement[2] [Wirth71a]. Stepwise refinement suggests guidelines for program development and introduces the interdependence of program (i.e. operations) and data refinement. However, it does not account for many of the modes of processing evident during synthesis[3] and does not integrate a record of the problem solving activity in the final program structure.

1.2. Project Overview

Investigations into the field of programming demonstrate that it is a complex task comprising many intricate subtasks. Research into the subtasks has advanced understanding of individual programming phases. However, interactions among these subtasks, both in the constraints each imposes on the other and their requirements on the various knowledge sources are of great importance. Programming is a continual transformation from conceptual process to concrete program. This research will assume that an investigation into the complete process of synthesis should occur and will be an attempt to construct an automated consultant capable of interacting with a user (the client) in the development of a software package that manages a video display buffer.

The programming system will accept English sentences and emit a keyword parse. Each keyword will be linked to either a prototypical linguistic or programming knowledge frame which will provide a basis for generating

---

[2]Similiar approaches include top-down program development, iterative enhancement, and structured programming.

[3]For example, the capability of suspending the current refinement path and switching the focus of the development to another section of the problem.

clarifying queries to the user. During this interaction an internal model, represented as a semantic network, will be constructed from the instantiation of the frames with problem-specific data. When all the domain terminology has been identified and defined, the system will attempt to construct an algorithmic solution, drawing on its base of techniques for solving "simple" programming tasks (e.g. forming expressions, generating table lookups, constructing sorts). These solution fragments will then be combined to form the overall problem solution. The states of the algorithmic solution are the goals of the coding phase. A particular strategy will be selected to obtain the goal state based on the state description, the available language constructs, and the current computation state. The strategy schema will be instantiated and integrated into the final program.

In the following sections a classification of program synthesis activities is presented and related research is reviewed using this classification. The requirements of a computer aided programming system are identified and a new program structure capable of fulfilling those requirements is developed. A preliminary system design is examined and a brief sample synthesis discussed.

2. Background

Program synthesis has historically been examined from two perspectives: software engineering, which emphasises the effects of language design and techniques on the final program; and artificial intelligence, which focuses on the processes of producing programs. The goal of software engineering (SE) is the development of better tools and techniques that will yield programs capable of supporting formal specifications, verification, and other functions. The techniques used in program synthesis, therefore, are interesting with respect to their effect on the final program form. On the other hand, artificial intelligence (AI) views the processes involved in program synthesis as having paramount importance. Programming activities are examined with regard to their interrelationships and effects on each other rather than their influence on the code sequence.

The different research emphases of SE and AI can be traced to different responses to the challenge of surmounting the complexity problem. Programming has evolved from the art of manipulating the bits of machine instructions to the task of analyzing and controlling complex abstract problems. Increased problem complexity caused programmers to search for better methods for expressing their problem solutions in a form that was both meaningful to the human problem solver and to the machine. High level languages (capable of supporting an extended set of abstract operations and data types, rather than just those of the basic machine repetoire) were initially sufficient to meet this need, but the increased demands on the programmer quickly outpaced the development of new languages.

At this point the approaches of SE and AI diverged. SE continued investigating the effects of modifications of languages on the expression of a problem solution, while AI began examining the activities occuring during program construction, most notably problem solving.

2.1. The Artificial Intelligence Perspective

Investigations of the processes involved in program synthesis reveal many intricately related activities. As outlined in the introduction, there appear to be four major categories of programmer activities within the client-consultant paradigm. These are (in roughly chronological order during the development of a program):

Requirement Acquisition: the obtainment of a description of a task and its domain from the client.

Specification Formulation: the transformation, completion, and refinement of the problem requirements into terms recognized by the system.

Algorithm Construction: the selection of known techniques for solving a task's subproblems and the combination of these solution fragments to form an overall task solution.

Code Production: the instantiation of language construct schemata that correspond to steps of the solution and whose execution will achieve the overall program behavior.

This classification encompasses the synthesis of a program and is incomplete with respect to the overall task of programming. Omitted functions include: verification, determination that the program correctly executes and achieves its intended behavior; debugging, location of program errors at the language, algorithmic, or conceptual level; maintenance and modification, support for the continued proper execution in a changing environment or with differing requirements. These later activities consume the majority of programmer effort (and expense) during the software life cycle. If the information present during synthesis can be made explicit and can be associated with the program structure, then the demands of verification, debugging, maintenance, and modification will be met with greater ease and efficiency.

The classification given here provides a framework for the following discussion of previous and current investigations of program synthesis. While the focus is on the results from the AI perspective, significant findings from other fields will be included. This survey is an attempt to identify the themes present in current work; it is not intended to be exhaustive, complete, or even indicative of the vast extent of recent or current research[4].

2.1.1. Requirement Acquisition

Requirement Acquisition is the highly interactive process of transfering an intended behavioral program description from the user to the programmer. During this goal directed activity the user conveys his conceptualization of the problem and its solution to the programmer in a clear and concise manner.

---

[4]See [Biermann76a] for a discussion of some of the projects not covered in this overview.

The programmer's task is primarily one of comprehension. Within the context of requirement transfer, the programmer attempts to understand the dialogue as a behavioral description. Unknown terminology is assumed to be problem specific and is initially classified by its role in the sentence and dialogue structure. This hypothesis of a term's functional role in the program forms the basis for clarifying queries to the user.

Techniques for expressing program requirements differ not only in the mode of communication but also vary in the degree of functionality expressed in the description. Within restricted domains, input/output behavioral descriptions of specific examples [Biggerstaff78a], [Green74a], [Biermann78a] and generic examples[5] [Green74a] has met with some success. Use of sample computation traces is an extension of the input/output techniques that describes the intermediate states of the data and the operations on the data during the transformation from original to final state. This approach has been tried with specific and generic data [Green74a], using pictorial descriptions of the process [Biermann76b], and special-purpose languages [Bauer79a]. Failure of this method in the unrestricted case arises from two sources: the difficulty of extracting the intended behavioral description and the problem of inferring its generalization from the examples' features.

Explicit (as opposed to inferred) behavioral descriptions are the basis of non-procedural expression of program requirements in the majority of task specification techniques. The requirements describe what the program is to accomplish without committing the achievement to a particular method. These descriptions are expressed using languages ranging from first order logic [Manna77a,78a], [Basili78a] to plain English. The formal languages precisely state the task requirements and can be processed by a deductive synthesizer with little modification which reduces possible misinterpretations, but places a burden on the specifer (i.e. the client) to produce the description correctly. Because formal languages are artificial, they require a user to learn a syntax and to express his concept in the language. They are most

---

[5] A generic example aids the extraction of the significant features by using a notation to identify the unimportant aspects of the data (e.g. ellipsis when the relative cardinality of the initial and final values is unimportant, or using variables, as opposed to values, when the type and not the specific identity of the solution is significant).

effective in well understood mathematical domains[6].

A formal language description of a task is closer to a complete program specification than a more general initial statement of requirements. Formal language descriptions hide the processing that occurs during the development of the specification in the same manner that the current program form hides the algorithmic development of the solution. In the client-consultant paradigm this processing is of paramount importance[7]. The computer-naive client must be guided to develop his statement of the task requirements.

Natural language is applicable in all domains, but may be susceptible to ambiguous interpretation. Correct parsing of English is admittedly beyond current computational models[8], but within restricted tasks English can be processed efficiently and with reasonable accuracy [Balzer77d,78a], [Martin76a], [Heidorn76a], [Ginsparg78a]. Potential ambiguities arising during requirement processing can be resolved by allowing the user to select his intended meaning from the set of generated interpretations. The added system burden of processing English is balanced by the client's ease of expressing the requirements.

2.1.2. Specification Formulation

By using English we can examine the process of completing program requirements and identify the knowledge sources that affect the final specification. Because English is an imprecise language, implied and unstated facts must be inferred by the system to complete a description. The imprecision of English can lead to misinterpretation of a client's statement when the implications intended by the client are not correctly inferred by the consultant. Nevertheless, use of a natural language allows the client to

---

[6][Balzer79a] defines two general classes of tasks: algorithmic operations which are characterized by mathematical functions and are generally unaffected by the dynamic environment (other than the input to the function); and process oriented operations which do not have well defined starting and stopping points, but effect the environment in which the process is running (e.g. operating systems). Formal language specifications are better suited for algorithmic tasks than process oriented operations.

[7][Ginsparg78a] states: "The program specification contains a record of everything the user has said (and the interpreter has inferred) which is relevant to the description of the program being written" (p. 108).

[8]See [Hobbs77a] for a discussion of parsing even the "well-stated" algorithm descriptions of Knuth.

describe the significant problem features without stating all the invariant conditions. The consultant can minimize the effects of ambiguity through requests for additional information about unknown terms; he can recognize when the description is at the correct level of detail.

During Specification Formulation the system augments the user's statements with its inferences, referent resolutions, and draws on its programming knowledge base to extract object and process descriptions from the dialogue. Ginsparg [78a] developed a natural language front-end for the PSI project that accesses a programming knowledge base to disambiguate the interpretations generated by his syntatic parser. These assertions are then passed to the Program Model Builder [McCune77a] which incorporates the facts into the system's model of the process. The syntactic approach to parsing differs from current semantic based systems by generating the most general (representing potentially many) parse(s) of the input sentence, and selecting the correct meaning by interacting with the programming knowledge base.

Other projects demonstrate the feasability of completing program requirements initially stated in English. The SAFE system [Balzer77a] accepts a pre-scanned English description of a task and generates a complete problem specification [Goldman77a] (expressed in the specification language, AP2) The activities of specification completion appear to be independent of the problem domain. Employing similiar techniques, process descriptions have been extracted and completed in nonsense domains [Balzer72a] and in domains that use symbols [Wile77a] to replace meaningful semantic labels. The key to the correct extraction of the description is the identification of problem domain operations and their classification into basic categories of processing (e.g. sequencing, iteration, case selection, etc).

2.1.3. Algorithm Construction

The result of Specification Formulation is a set of states that must be achieved by the system's problem solving component during Algorithm Construction. Associated with each state description is a collection of strategies for achieving the state, or for further decomposing the state. These strategies have context sensitive selection criteria that allow for intelligent choice of an appropriate method for achieving a state. The individual problem solutions are combined to form an overall task solution.

Approaches to automatic problem solving have centered around three topics: strategy selection and instantiation, recognition of subgoal interactions, and resolution of inimical interactions. The first of these investigates the representation of the criteria that should be examined before a strategy is used in the problem solution. The criteria can be encoded as state description patterns that are modified as the system learns to identify significant features of a problem description [Sussman75a], as antecedent clauses of production rules describing refinements [Barstow77a] or deductions [Manna75a], deep plan preconditions [Rich76a], [Shrobe79a], [Waters78a], or discrimination networks [Rieger76a]. The problem solver attempts to verify (or achieve) each of the preconditions and if successful incorporates the strategy into the problem solution. Not all the conditions that could affect the strategy can be verified at selection time and thus a choice of a strategy may be retracted if conditions later prohibit its use.

The problem solver accesses a data base that reflects the state of the model during algorithm construction. This data base must recognize contradictions and interactions among subgoals. States that have already been achieved are "protected", a status that preserves the current value during the solution of other subgoals. When a state description contradicting the protected state is asserted, the protection mechanism reacts and halts the current solution derivation. The order of the subgoals may be changed, a different strategy selected, or the problem solver can enter a debugging mode to locate the cause of the interaction [Sussman75a].

Two alternatives to situations that arise from inimical interactions among subgoals are the use of subgoal reordering or alternate strategies. Both of these techniques place a burden on the modelling component of the system to reestablish a previous state of the problem solver. Subgoal reordering permits goals to be passed back [Waldinger77a] or solution steps unravelled [Rieger77a]. Alternate strategy selection masks the effects of the original strategy and selects another method of solving the problem. In either case the assertions associated with a partial solution must be removed from the model and the reasons for the interaction examined. One of the more efficient methods uses a dependency based model [Doyle78a], [London78c], in which assertions are linked to their justifications. This technique supports not only the immediate identification of the states used in deducing the

contradiction, but can also be employed to guide the backtracking mechanism to remove the effects of the interaction.

2.1.4. Code Production

Code Production is the construction of a sequence of instructions whose execution would achieve the desired behavior. During this phase the constraints of the selected programming language are blended into the solution produced by the problem solver. Solution steps of the algorithm are combined to form sequences of statements (by following the action thread of the problem solver), conditional statements (formed by recognizing two opposite states are assumptions to a similiar set of actions), and iterative constructs (by identifying the occurrence of the initial goal state description within the solution graph for that state). While the amount of problem solving activity is less than that of Algorithm Construction, backtracking from situations can occur during Code Production.

The majority of processing is the allocation of space for data representations and the construction of code sequences from the algorithmic solution. The solutions produced during Algorithm Construction use descriptors [London78b] to define a class of machine objects needed to achieve a goal. These descriptors must be identified and actual machine resources committed to a goal. Problems concerning limited resources can occur and some reordering of the solution required.

The generation of the actual instructions varies according to the particular method of problem solving used. In systems that use a production rule approach, programming language instructions are specified as a final rule application. Other deductive approaches augment the derivation of the proof of a program with statements [Manna75a], [Basili78a]. Reduction problem solvers, such as [Rieger76a], explicitly represent the effects of an action on a state and incorporate this knowledge into the solution graph. Surface plans [Rich76a], though used in a recognition technique, could be employed in code production. A system organized around a library of such plans has been proposed as a part of the LISP Apprentice project.

2.2. Other System Approaches

Several projects have attacked the complete task of program synthesis and analysis. Noted among these is the work of the PSI project [Green74a,76a], [Ginsparg78], [McCune77a], [Barstow78a]; the SAFE project [Balzer72a,77a] and [Wile77a]; and the LISP Apprentice [Rich77a], [Shrobe79a], and [Waters79a]. These systems differ in the philosophy of program synthesis they adopt (and the LISP Apprentice is actually an analysis system), but several themes are common to all three.

All the projects accept a description of the intended program and develop an internal representation of the program. In the PSI and SAFE systems the description of the process is refined until a complete specification is produced. The PSI coding expert [Barstow77a] is totally committed to the concept of stepwise refinement and the current state of the model is represented by the fringe of the refinement tree. The system is driven by a base of several hundred programming rules (encoded as productions) and has sucessfully synthesized several sorting and concept formation programs. The PSI system requires a more procedurally oriented specification[9] than the SAFE system, which is designed to accept task specifications from DOD manuals. The SAFE system determines the overall program control flow by identifying producer-consumer relationships between subgoals of the task. This identification is based on the concept that the operation that produces an object must occur prior to the use (consumption) of the object. This relationship is used to determine a partial ordering on the goals of a problem. The proposed specification is then tested using meta-evaluation, a form of symbolic evaluation, to identify goals which incorrectly interact with each other.

The LISP Apprentence is designed to aid an experienced programmer to develop programs. At first the system is given a section of LISP code and the underlying 'deep plan' (i.e. the program specification). The code is transformed into a 'shallow plan' containing data and control flow. The system then attempts to verify that the shallow plan does indeed achieve the deep plan. The deduction will locate errors in the code sequence that do not

---

[9] See [Green77a] for a presentation of a hypothetical dialogue between a user and the PSI system which describes the development of a sorting program.

have a counterpart in the deep plan (and sections of the deep plan that are not coded). The deductive process makes use of dependency links to record justifications for its conclusions. Future modifications to the program can be checked against the existing deduction for inconsistencies.

2.3. Summary

The goals of this project, while similiar in spirit to previous work, differ in several ways. This research will study the synthesis of assembly language programs, a more concrete and low-level domain than has been examined before. This will, hopefully, lead to a better understanding of more general programming activitives. The system, while accepting English input, will direct the development of the program specification, interrupting the user when unknown terminology is presented, and play a more active role in the interaction than other systems. Finally, the development of a new program structure that contains an explicit record of all processing done and knowledge used during the program development, will produce, not only, an executable sequence of code, but also a description of the problem solving activities that produced the program.

3. Organization of the Model

The research proposed in this report embraces two complementary phases: (1) the identification and codification of a subset of the techniques and knowledge sources employed by a human programmer during synthesis; and (2) the design and implementation of a system capable of interacting with a human user to construct computer programs. By casting this investigation in the client-consultant paradigm, activities that are general to programming can be isolated and examined. The interaction between the client and the consultant forms a basis for identifying the constraints each imposes on the other and for defining the interface between a domain expert and an adaptable programmer. The information gained from this phase will aid the construction of an interactive system capable of fulfilling the role of the consultant during the synthesis of a program.

Implementation of the computer aided synthesis system will include the development of a new program structure, the program model, and the design of components corresponding to each of the four phases identified in the client-consultant paradigm. These components will access the codified knowledge sources to construct and manipulate the program model. The program model will contain a record of all the processing that occurs during synthesis. Definitions and refinements specified by the user will be represented as will the selections and decisions generated by the system. The system components will be organized in a process queue that will accommodate user control of the synthesis process, changes to the level of specification, and removal of program model segments because of modified specifications. These modes of processing are found in the client-consultant paradigm and will be modelled by the system. The implementation of the synthesis process will provide feedback on the adequacy of the programming knowledge base and our description of the program synthesis process.

3.1. The Problem Domain

The problem domain selected for this project is the development of a software package that manages a video display buffer and is designed to run on a simple microprocessor, an INTEL 8080. The synthesis of this package contains many subactivities that are common to all programming domains. This examination should further our understanding the processes that occur during

synthesis.

Initially, the domain objects are described to the system. The screen is described and a set of attributes describing its character size and extent is presented. The concept of a window (a rectangular subregion of the screen) is developed and the set of parameters of a particular window are enumerated. Following the presentation of the domain objects, the user describes the functions that manipulate the screen and the windows. The concepts of opening, clearing, framing, labeling, and printing to a window are developed using terminology introduced in the initial description of the domain objects. These functions are then refined in terms of more concrete actions. Thus printing a string of characters to a window is refined to be the transfer of the characters contained in the string to the buffer locations corresponding to the location in the window where the string would appear. The remaining functions of the package are refined in a similar fashion.

The following sections examine in greater detail the organization of the components of the proposed synthesis system. The program model and the knowledge sources are examined with regard to their form and interactions during synthesis. Other forms of knowledge, such as linguistic knowledge, also will be presented. The overall proposed control structure is described and discussed with respect to the fulfillment of the requirements imposed by the interface with a human user.

3.2. The Program Model

The central data structure of the synthesis system is the program model. This new relational description of a program contains representations of not only the final code sequence generated during the synthesis, but also the effects of the various activities that occur during the development of the code. It also contains assertions describing the current development of the program and justification for believing each of the assertions. Explicit representation of information and relationships among facts will support the requirements of the modes of computation evident during program synthesis.

The program model is encoded as a semantic network, a graph-like structure of nodes which correspond to concepts in the program model and connecting arcs which represent relations among the concepts.

The concept node acts as a repository for descriptions of the features associated with each concept. A feature description is a set of facts containing the name of the feature, the type of values that can fill the role of the feature (a link to another concept), and a pointer to either an individual value to be used in the description or a set of criteria that further refines the type of description filler. The feature is connected to the concept node it describes via a link type that corresponds to the method by which the description was associated with the concept.

The links in the semantic network define the flow of information among nodes in the program model. This control is embodied in the state of the connections and the semantics associated with the individual link types. Information passes from one node to another only if they are connected and the information can be transformed or filtered by the semantics of each link type[10]. For example, the ISA link allows the inheritance of a feature description from an ancestor node to its offspring if the offspring node does not contain an explicit modification of the feature. The dependency link can aid the detection of an inconsistency in the state of the model which occurs when two mutually exclusive nodes are asserted into the model. After the contradiction is detected, the set of justifications for the two assertions can be examined and the node that is logically unsupportable can be masked[11]. Processing of information between nodes that are not directly connected to each other is defined by the sequence of processing that occurs at the intermediate links along a path between the two nodes.

The set of links that will be represented in the program model (and considered primitive to programming) will include the following types: inheritance (INH), definition (DEF), representation (REP), refinement (REF), reduction (RED), and logical dependency (DEP). Except for the last type of link, all the link types describe assertions in the program model. The DEP link will support processing involving the belief of assertions in the model.

---

[10]The level of representation associated with this type of semantics is the "conceptual" level of semantic nets. See [Brachman79a] for a complete classification of representational levels of link types.

[11]A masked assertion remains in the program model, but its truth value is hidden from the deduction component. If the set of justifictions for the assertion is later re-established, then the assertion will be unmasked.

3.2.1. Definitions and Representations

The DEF and REP link types are used to associate feature descriptions of an abstract object with the concept node in the program model that corresponds to the object. Objects are described in terms of less abstract objects which are subsequently defined until the object is expressed using terminology recognized by the system. These prototypical programming objects can be implemented using several methods, one of which must be selected for encoding in the program. The DEF link represents a user specified description, while a REP link denotes a system generated selection.

Abstract objects, initially unknown to the system, are defined by the user in terms of other concepts and are integrated into the program model via the DEF link. The user will specify the name of the object and the set of features associated with the new object. These features can be members of the built-in programming knowledge base or can be unknown to the system, in which case they are marked for further processing. Abstract object definitions may have several levels of description before being expressed in known terminology and the degree of specificity will be controlled by the user. The justification for each DEF link will be marked as user specified for use in dependency directed processing. Once an object's definition is complete it will be recognized by the system and can be used in subsequent definitions.

All objects in the problem domain must ultimately be represented by a memory location and a storage descriptor. The REP link denotes the selection of a particular encoding for an object. While programming objects that are generated by user specified definitions are recognized by the system, their final implementation will be influenced not only by the features associated with the object, but also by the object's use in the program[12]. The division of descriptions into two classes parallels the distinction between the two domains of expertise of the client and the consultant. Requests for information from the user will be expressed using terminology identified by

---

[12]It has been demonstrated that the implementation of data types that are considered primitive to a high-level language can best be selected after the complete program has been examined. Compilers that are modified to do this type of analysis (see [Low74a] for an example of a SAIL compiler) are more efficient than traditional compilers that employ the most general encoding for a data type (i.e. the encoding capable of accounting for all possible uses of the data type).

DEF links, while the system will attempt automatically to infer information about REP link concepts. The user will describe his task in domain specific terms and the system will construct an implementation that parallels this description using concrete objects that can be implemented on a computer.

3.2.2. Refinements and Reductions

REF links correspond to a user's perceived decomposition of a problem domain operation. An abstract process is defined by the set of subprocesses required for its execution. For example, to output a character to the video display requires a calculation of the cursor position on the screen, the transfer of the character to the buffer location corresponding to the position, and the advancement of the cursor. This produces three new processes that must be refined. Eventually the process will be described using operators recognized by the system. Once an abstract process has been completely refined it can be referred to during other process decompositions without being redescribed. This technique allows the user to describe an operation without worrying about potential interactions with other operations. The system will recognize these inimical situations and request a new refinement of the abstract operation.

A reduction is a form of goal decomposition that is generated by the system's problem solver, as contrasted with a user supplied refinement. While the actions represented by both the RED and REF links are generated during state decomposition, the system will react quite differently to inimical interactions among the generated nodes depending on the link type. If a refinement state is unattainable, the system will report this to the user and request a new decomposition. If subgoal interaction among nodes generated by a reduction is detected, the system will exhaustively attempt other known strategies before reporting failure back to the user.

This seems to be a natural division of process decompositions occuring during synthesis. The user supplies the refinements that are specific to the problem domain. The system (in the role of a consultant) cannot evaluate the soundness of the decompositions and in the event of interactions among states generated by the user's refinement it can only identify the failure and await directions from the user. The other situation occurs when a decomposition generated by the system fails. This is a common occurrence with problem

solvers that assume initial subgoal independence[13] and can be resolved by alternate orderings or strategies. This situation arises when the system permits the user to specify a refinement of a subtask in isolation. The system recognizes the primitive operators; believes that the task has been sufficiently specified, but due to the particular context of the computation state interactions occur.

3.2.3. Dependencies

The DEP link represents the deductions that occur during the generation of a concept. They reflect the reasons for believing an assertion. During synthesis requests for determining the truth value of an assertion will occur. If the assertion is already in the program model, the current truth value will be returned. If the data is not present, a deductive component will attempt to derive the truth value. The justifications used by deduction will be linked to the conclusion via a DEP link. If the conclusion is later condtradicted, or the truth values of the justifications modified, the DEP link will be invalidated. In such a case the conclusion (or the justifications) will be need to be rederived. The difficulty in most current systems is that the facts used in a deduction are not explicitly associated with the conclusion (or each other). The DEP link provides an immediate identification of this set of assertions and pointers to possible second order effects.

An additional property of the DEP link is its ability to preserve the logical relationships among facts independently of their history of generation. A program synthesis system must support the retraction of a design decision during program development. Yet the information that was gained during the original (and unsuccessful) solution attempt must be retained (and not rederived on subsequent attempts). When a solution path must be removed any assertions whose justifications are independent of the assumptions made along the path can be retained. This is a powerful advantage to previous problem solving modelling mechanisms (e.g. Conniver's context-layered data base [McDermott74a]).

---

[13] The potential for interactions is balanced by the generality of strategies needed by such a problem solver. The strategies do not account for all possible interactions in a given context, but are accompanied by techniques for alleviating subgoal interactions.

3.3. The Programming Knowledge Base

The program model is constructed by instantiating prototypical programming knowledge with problem specific data which is described by the user. The programming knowledge base consists of facts and program construction techniques that are considered primitive to programming and employed during synthesis. This collection includes descriptions of data types and rules for their combination to form new abstract types, criteria required by a type description, techniques for decomposition of states for problem solving and recognition of goal interactions, methods for construction of expressions, conditionals, input and output statements, etc. One characteristic of the knowledge base is that the facts are applicable to many programming domains.

The information will be organized in a frame system [Minsky75a]. A frame is a collection of descriptions of the attributes that define an object. The stereotypical descriptions form a perspective, or view, of the object in a particular role. Objects can have multiple perspectives that correspond to different functional uses. For example, a memory location can be viewed as a program instruction or datum as well as having attributes describing its address, extent, and capabilities. The individual frames are organized in a hierarchy that permits the inheritance and modification of descriptions from one object to another.

Frames provide an efficient organization of knowledge for two activities occuring during synthesis: recognition and inference[14]. Features presented during the user's behavioral task description will suggest potential programming objects that could represent the objects and operations in the abstract domain. Identification of a particular entity will supply information normally associated with the object, but not explicitly stated in the user's discourse. These inferences will provide a basis for additonal queries to the user requesting more information, or selection of a particular object from a set of candidates.

The programming frames will contain information describing the defining

---

[14]This organization of knowledge has also been proposed as a psychological model of human memory (see [Norman79a]).

characteristics and potential roles of an object in a program. The array frame contains descriptions of its dimensionality, extents and indices for each of the dimensions, its static or dynamic behavior during execution, standard and alternative encodings, and other features normally not explicitly associated with an array, such as the ability to define sub-arrays. Operations are represented by the set of states requisite for their correct execution. These states may be ordered or declared independent to the problem solver. The prerequisite and post conditions (including side-effects) also are included in the descriptions of an operator's attributes.

3.4. Linguistic Knowledge

The frame identification process is initiated by the system's keyword parser. This linguistic component accepts English sentences and after rudimentary morphological and phrase-structured analysis emits a parse that identifies not only the known keywords but also groups unknown terms and phrases together according to their location in the original sentence. The keywords are associated with the prototypical frames which aid in the sentence interpretation. The parser has a limited capability and is not intended to be a complete model of natural language processing[15]. Rules for identifying the extent of a phrase, simple word sense disambiguation, and referent resolution are encoded. Terminology not recognized by the parser is considered domain-specific and marked for further definition or refinement. The role of these unknown terms (e.g. operation or operand) is hypothesized and tests generated to confirm or disprove the hypothesis. In the case of ambiguous sentences the user is queried to select the correct sense from those generated by the parser.

During Requirement Acquisition and Specification Formulation the parser plays an active role in directing the session. The system must be capable of recognizing when no new information (other than abstract terminology) is being introduced and pose questions that will direct the discussion in a more fruitful direction. These questions will attempt to identify the general characteristics of the unknown objects, determining features such as

---

[15]Keyword parsers have been sucessfully employed in other projects. Included in this list is PARRY [Colby75a], a model of human paranoia that successfully passed the 'Turing test'.

cardinality, use by other functions, and relationship to other known, or previously introduced, objects. Process definitions will be examined with respect to their (hopefully) defined operands and prototypical operations on the operand data types will be offered for selection. The system performance will degrade gracefully and model the behavior of a human programmer faced with a similar situation.

3.5. Program Synthesis Processes

The construction of a program does not progress in a straightforward manner. A system capable of aiding a human during synthesis must accomodate changes in the levels of specification or the direction of a process refinement, and the removal of sections of already constructed program due to changes in the problem specifications. Even when constrained by a discipline such as step-wise refinement, human programers behave in the same manner.

> "I feel [sic] somewhat guilty when I have suggested that the distinction or introduction of 'different levels of abstraction' allow [sic] you to think about only one level at a time, ignoring completely the other levels. This is not true. You are trying to organize your thoughts; that is, you are seeking to arrange matters in such a way that you can concentrate on some portion, say with 90% of your conscious thinking, while the rest is temporarily moved away somewhat towards the background of your mind. But this is something quite different from *'ignoring completely': you allow yourself* temporarily to ignore details, but some overall appreciation of what is supposed to be or to come there continues to play a vital role. You remain alert for little red lamps that suddenly start flickering in the corners of your eye." - E.W. Dijkstra in [Knuth74a]

This ability to switch the focus of the discussion must be acceptable to the system.

To support this behavior, the control structure of the system is organized in a muti-level queue. The process that is scheduled to run is removed from the queue and initiated. The process may complete and new processes sprouted from the running one will be inserted into the system queue. The priority scheme will attempt to continue the current development of an object, before switching the focus of the dialogue. The user has the option of modifying the sequence of tasks by suspending the current process and specifying a process to be resumed. In this case the currently running process is inserted back into the queue and flagged that it was suspended by the user. Process statuses are: Ready - a process that has been generated and queued, but has never been activated; Running - the currently active process; Suspended - a previously active process that has been requeued pending a

specified condition; Terminated - a process that has run to completion.

An additional mode of processing requires the system to retract assertions associated with a section of the program model. This corresponds to the failure of an attempted solution to the problem. During the construction of the solution, assertions that were based on the solution method were added to the model. Other facts, independent of the strategy, were also incorporated in the program model. By invoking a backtracking mechanism that follows dependency links (instead of being chronologically oriented) the invalid information can be masked, while the independent assertions retained. This allows the system to gain some information even in the case of a partial failure and to behave like a human programmer that learns from his false starts when writing a program.

4. Example Synthesis of Video Display Module

In the scenario chosen as the basis of development for this project, the programming system will interact with a user to develop a software package that controls a video display buffer. The knowledge bases for this application will contain sufficient information to recognize terms that are familiar to consultants. Some of the capablities present in the system will seem too powerful, while the absence of others will make the system appear incomplete. The actual level of detail of the knowledge base is not at issue. The modes of processing that access the knowledge base are of interest. Three glimpses of the synthesis are presented: the interpretation of a scenario sentence, the refinement of a task process, and the generation of a code sequence. The complexity of even a short example prohibits a complete treatment of the synthesis.

Keyword Parsing. The system scanner accepts English descriptions of the process, identifies known keywords, and groups together the unfamiliar words. The scanner performs elementary morophological and phrase structured analysis on the sentence before passing control to the parser. The following sentence is one of the first presented to the scanner (the keywords are underlined).

"The display screen is thought of as a 40 by 86 byte array."

The scanner recognizes articles, numbers, and certain puncuation in addition to the programming keywords. Rules for identifying a word's role in the sentence determines that "display" is an adjective modifying screen, that "byte" is an adjective (it can be a noun, too), that the phrase "40 by 86 byte" modifies array, and that the general structure of the sentence is definitional (i.e. <x> is <y>). This information is asserted into the problem model and control passed to the parser.

The parser locates the frames associated with the identified keywords, and attempts to instantiate each frame with the information present in the sentence. The screen frame asserts the existence of an output device named display. The array frame is more helpful. It asserts that the array has two dimensions, the size of each dimension (40 and 86), the type of each array member (byte), and queues questions to the user regarding the first and last indices, whether the array is static or extensible, what the initial values should be, what operations access the array, etc. The internal program model links the instantiated screen and array concepts with a definition link.

The queries generated by the knowledge frames are processed by the system according to the priority scheme. Before posting a request to the user, the program model is examined for the required information which if present is used by the system. This corresponds to the following situation: between the time the query was originally inserted into the queue and the time it was run the user supplied the required information while responding to another request (or the system was able to infer the information).

During the ensuing exchanges, the user describes the concept of screen rectangular subregions called windows. This attribute is linked to the screen node. Since the screen is defined to be a 2 dimensional array, the array frame is searched for a feature that corresponds to a window. The array concept is augmented with a description of array subregions, which is subsequently linked to the window concept via a definition link. The resultant structure contains the knowledge that a window is defined to be a subregion of the array corresponding to the screen. This hypothesis is presented to the user for confirmation.

The process of integrating additional information into the program model continues until all terms are recognized. In this example, the array concept is known to the system and is not defined further (by the user). The system will use the acquired descriptions of the screen, window, and their associated features to construct process descriptions of the task operations.

Task Refinement. One of the functions of the video display package causes the cursor of the current window to be displayed according to the parameters associated with a window. The initial process description must be refined to identify each of the cursor display modes and their preconditions. The conditions are: if the cursor display is 'visible' then print either the cursor character or reverse the background of window character located at the cursor position. This is refined into a test of the cursor mode attribute and a branch to the appropriate routine. The final process description is: if the cursor display is 'visible' then if the cursor mode is 'reverse figure background' then reverse the background of the character displayed at the cursor position, if the cursor mode is regular then display the cursor character at the current position, and if the cursor display is not 'visible' then do nothing.

Generating a Code Sequence. During coding the machine instructions that will achieve a goal are produced. The process descriptions are further refined until all the states can be achieved by a language instruction (or sequence of instructions). In the video display example, the target language is INTEL 8080 assembly language. The INTEL 8080 is an 8-bit microprocessor containing seven registers that can be accessed individually or as 16-bit register pairs. The language contains a primitive addressing mode and does not have index registers.

One of the goals contained in the algorithm is to store a character in the video display buffer. This goal is split into several subgoals corresponding to the different preconditions for each of the display modes. The difference between two of the display modes is whether the cursor should be displayed normally or with the figure background reversed. The code used for isolating this criterion is constructed, a test instruction for the condition inserted into the program, and the two display mode program segments written. During this phase specific registers and memory locations are allocated according to the descriptors used in the algorithm construction.

4.1. Implementation

The interactive synthesis system will be implemented in LISP, a language developed by McCarthy and based on the lambda notation of Church. Maryland LISP [Agre79a] is a powerful dialect of LISP that supports a relational data base, pattern directed function invocation (demons), and generators similiar to those found in CONNIVER [McDermott74a].

5. Conclusion

Program synthesis is a complex task that is composed of many interacting subactivities and that accesses a variety of knowledge sources during the development of a computer program. Recent investigations have discovered the inadequacies of current synthesis techniques to keep pace with the increasing difficulties of managing large intricate problem solutions. One direction towards breaking this "complexity barrier"[16] is the development of intelligent computer systems that are capable of managing the vast amount of information assimilated and accessed during synthesis. The systems' "intelligence" is characterized not by an innate ability to invent problem solutions, but by the incorporation of an internal model of the problem domain and the corresponding program solution.

The research proposed in this report is directed toward the development of a theoretical model of program synthesis and an implementation of a programming system that incorporates this model. This investigation is focussed on a particular style of programming, the client-consultant paradigm, and is examining a specific task, the development of an assembly language package for managing a video display buffer. By examining the complete transformation from acquistion of the initial statement of the task to the construction of code sequences, we will gain insights into the character and interactions of the knowledge bases and activities of synthesis.

There will be several contributions from this project. A knowledge base of general programming techniques and information employed by an adaptable consultant will be examined and codified. A new representation of a program, the program model, will be developed to record the processing that occurs during synthesis and will be organized to facilitate the requirements of synthesis activities. The control structure of the system will be developed to accomodate the modes of processing evident during the interactive development of a program. Finally, by implementing the program synthesis system and codifying the programming knowledge base, we will have an experimental tool capable of providing feedback on the adequacy of the knowledge bases and our description of the programming process.

---

[16]This term was introduced in [Winograd73a] to describe the difficulties that are encountered in constructing and managing large systems of programs.

6. References

[Agre78a]
Agre, P., Maryland LISP Reference Manual, Univ. of Maryland, TR-678, Jul. 1978.

[Balzer72a]
Balzer, R., Automatic Programming, USC/Information Sciences Institute, RR-73-1, Sep. 1972.

[Balzer77a]
Balzer, R., Goldman, N., & Wile, D., Meta-Evaluation as a Tool for Program Understanding, Proc. of IJCAI-77, Cambridge, Mass., Aug. 1977.

[Balzer77d]
Balzer, R., Goldman, N., & Wile, D., On the Use of Programming Knowledge to Understand Informal Process Descriptions, USC/Information Sciences Institute, RR-77-63, Oct. 1977

[Balzer78a]
Balzer, R., Goldman, N., & Wile, D., Informality in Program Specifications, IEEE Transactions on Software Engineering, Vol. 4,2, Mar. 1978, pp 94-103.

[Balzer79a]
Balzer, R. & Goldman, N., Principles of Good Software Specification and their Implications for Specification Language, Proc. of SRS, Cambridge, Mass., Apr. 1979.

[Barstow77a]
Barstow, D.R., Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules, Stanford A.I. Laboratory, Memo AIM-308, Nov. 1977.

[Basili78a]
Basili, V.R. & Noonan, R.E., A Comparison of the Axiomatic and Functional Models of Structured Programming, Dept. of Computer Science, Univ. of Maryland, TR-630, Feb. 1978.

[Bauer79a]
Bauer, M.A., Programming by Examples, Artificial Intelligence, Vol. 12, 1979, pp 1-21.

[Biermann76a]
Biermann, A.W., Approaches to Automatic Programming, In M. Rubinoff & M.C. Yovits, Eds., Advances in Computers, Vol. 15, New York: Academic Press, 1976.

[Biermann76b]
Biermann, A.W. & Krishnaswamy, R., Constructing Programs from Example Computations, IEEE Transactions on Software Engineering, Vol. 2,3, Sep. 1976, pp 141-153.

[Biermann78a]
Biermann, A.W., The Inference of Regular LISP Programs from Examples, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 8,8, Aug. 1978, pp 585-600.

[Biggerstaff78a]
Biggerstaff, T.J., Factored Specifications in the Synthesis of LISP Functions, Proc. of ACM 78, Washington, D.C., Dec. 1978.

[Brachman79a]
Brachman, R.J., On the Epistemological Status of Semantic Networks, In N. Findler, Ed., Associative Networks-The Representation and Use of Knowledge in Computers, New York: Academic Press, 1979.

[Colby75a]
Colby, K.M., Artificial Paranoia: A Computer Simulation of Parnoid Processes, New York: Pergamon Press, 1975.

[Doyle78a]
Doyle, J., Truth Maintenance Systems for Problem Solving, MIT A.I. Laboratory, A.I. Memo 419, Jan. 1978.

[Ginsparg78a]
Ginsparg, J.M., Natural Language Processing in an Automatic Programming Domain, Stanford A.I. Laboratory, Memo AIM-316, Jul. 1978.

[Goldman77a]
Goldman N., Balzer, R., & Wile, D., The Inference of Domain Structure from Informal Process Descriptions, USC/Information Sciences Institute, RR-77-64, Oct. 1977.

[Green74a]
Green, C., et al, Progress Report on Program Understanding, Stanford A.I. Laboratory, Memo AIM-240, Aug. 1974.

[Green76a]
Green C., The Design of the PSI Program Synthesis System, Proc. of the Second International Conference on Software Engineering, San Francisco, Calif., Oct. 1976.

[Green77a]
Green, C. Barstow, D., A Hypothetical Dialogue Exhibiting a Knowledge Base for a Program-Understanding System, In E.W. Elcock & D. Michie, Eds., Machine Intelligence 8, Sussex: Ellis Harword Ltd., 1977.

[Heidorn76a]
Heidorn, G.E., Automatic Programming Through Natural Language Dialogue: A Survey, IBM Journal of Research and Development, Vol. 20,4, Jul. 1976, pp 302-314.

[Hobbs77a]
Hobbs, J.R., From "Well-Written" Algorithm Descriptions Into Code, Dept. of Computer Sciences, City College, CUNY, Research Report 77-1, Jul. 1977.

[Knuth74a]
Knuth, D.E., Structured Programming with GOTO Statements, ACM Computing Surveys, Vol. 6,4, Dec. 1974, pp 261-301.

[London78b]
London, P., Approaches to Object Selection for General Problem Solvers, Proc. of the Second National Conference of the Canadian Society for Computational Studies of Intelligence, Toronto, Ontario, Jul. 1978.

[London78c]
London, P.E., Dependency Networks as a Representation for Modelling in General Problem Solvers, Dept. of Computer Science, Univ. of Maryland, TR-698, Sep. 1978.

[Low74a]
Low, J.R., Automatic Coding: Choice of Data Structures, Stanford A.I. Laboratory, Memo AIM-242, Aug. 1974.

[Manna75a]
Manna, Z. & Waldinger, R., Knowledge and Reasoning in Program Synthesis, Artificial Intelligence, Vol. 6, 1975, pp 175-208.

[Manna77a]
Manna, Z. & Waldinger, R., Synthesis: Dreams => Programs, SRI International, Tech. Note 156, Nov. 1977.

[Manna78a]
Manna, Z. & Waldinger, R., The Logic of Computer Programming, IEEE Transactions on Software Engineering, Vol. 4,3, May 1978, pp 199-229.

[Martin76a]
Martin, W., Knowledge Based Systems, Laboratory for Computer Science Progress Report XIII, MIT, Cambridge, Mass, Jul. 1976, pp 87-108.

[McCune77a]
McCune, B.P., The PSI Program Model Builder: Synthesis of Very High-Level Programs, Proc. of the Symposium on Artificial Intelligence and Programming Languages, Rochester, NY, Aug. 1977.

[McDermott74a]
McDermott, D.V. & Sussman, G.J., The Conniver Reference Manual, MIT A.I. Laboratory, A.I. Memo 259a, Jan. 1974.

[Minsky75a]
Minsky, M., A Framework for Representing Knowledge, In P.H. Winston, Ed., The Psychology of Computer Vision, New York: McGraw Hill, 1975.

[Norman79a]
Norman, D.A. Bobrow, D.G., Descriptions: An Intermediate Stage in Memory Retrieval, Cognitive Psychology, Vol. 11,1, Jan. 1979, pp 107-123.

[Rich76a]
Rich, C. & Shrobe, H.E., Initial Report on a LISP Programmer's Apprentice, MIT A.I. Laboratory, AI-TR-354, Dec. 1976.

[Rieger76a]
Rieger, C., An Organization of Knowledge for Problem Solving and Language Comprehension, Artificial Intelligence, Vol. 7, 1976, pp 89-127.

[Rieger77a]
Rieger, C. & London, P., Subgoal Protection and Unravelling during Plan Synthesis, Proc. of IJCAI-77, Cambridge, Mass., Aug. 1975.

[Shrobe79b]
Shrobe, H.E., Dependency Directed Reasoning for Complex Program Understanding, MIT A.I. Laboratory, A.I. Memo 503, Apr. 1979.

[Simon69a]
Simon, H.A., The Sciences of the Artificial, Cambridge, Mass.: MIT Press, 1969.

[Sussman75a]
Sussman, G.J., A Computer Model of Skill Acquisition, New York: American Elsevier, 1975.

[Waldinger77a]
Waldinger, R., Achieving Several Goals Simultaneously, In E.W. Elcock & D. Michie, Eds., Machine Intelligence 8, Sussex: Ellis Horwood Ltd., 1977.

[Waters78a]
Waters, R.C., Automatic Analysis of the Logical Structure of Programs, MIT A.I. Laboratory, A.I. Memo 492, Dec. 1978.

[Wile77a]
Wile, D., Balzer, R., & Goldman, N., Automatied Derivation of Program Control Structure from Natural Language Program Descritpions, Proc. of the Symposium on Artificial Intelligence and Programming Languages, Rochester, NY, Aug. 1977.

[Winograd73a]
Winograd, T., Breaking the Complexity Barrier (Again), Proc. of the ACM SIGIR-SIGPLAN Interface Meeting, Nov. 1973.

[Wirth71a]
Wirth, N., Program Development by Stepwise Refinement, Comm. ACM, Vol. 14,4, Apr. 1971, pp 221-227.

7. Appendix

The following excerpt is from the video display package scenario and describes the development of the definition of a window. The sentences that are input by the user are prefaced by a ">". Following the sentence is a partial list of the inferences made by the system.

> The display screen is thought of as a 40 by 86 byte array.

1) The display screen is represented by an array of bytes.
2) The display array is dimensioned 40 by 86.
3) There are 40 rows in the array.
4) There are 86 columns in the array.

> Screen rows are counted from row 0 (topmost) to row 39 (bottommost).

1) The samllest row number is 0.
2) The largest row number is 39.
3) The name of row 0 is topmost.
4) The name of row 39 is bottommost.
5) A valid reference to a screen row number must be in the interval 0<=row-number<=39 for integer row-number.

> Screen columns are counted from 0 (leftmost) to 85 (rightmost).

1) The smallest column number is 0.
2) The largest column number is 85.
3) The name of column 0 is leftmost.
4) The name of column 85 is rightmost.
5) A valid reference to a screen column number must be in the interval 0<=col-number<=85 for integer col-number.

> Each window defines a rectangular subregion of the display screen from size 1 by 1 up to 40 by 86.

1) A window is a subregion of the display screen.
2) The subregion is rectangular.
3) A rectangular region can be defined by giving a base point and an extent in each of the two dimensions. (Another method is to give the coordinates of two diagonal corners.
4) The extent of the row is 1 to 40.
5) The extent of th column is 1 to 86.
6) The window must be totally contained on the screen.
7) A window contains bytes.

> Except for functions which affect a window's position on the screen, positions are expressed relative to the interior of a window.

1) The window has a position on the screen (the base point).
2) A position is in terms of a window's row and column.
3) To specify a position you must name a window and a location within the window.

> The topmost print line of a window's interior is line 0; the leftmost print column is column 0.

1) The first print line is window row number 0.
2) The first print column is window column number 0.
3) Window row number 0 is named topmost.
4) Window column number 0 is named leftmost.
5) The last window row number is row-extent - 1.
6) The last window column is number col-extent - 1.

> Each window has a cursor.

> The cursor is the position within the window where the next character will be printed.

> Overlapping windows are permitted, but operations on a window take place without regard for possible effects on any overlapping windows.

> Each window has a set of paramters.

> The user will directly set the visible cursor and cursor mode to engage or disengage different modes of control.

> The window package functions are responsible for maintaining and updating the other parameters.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A092 622	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) Computer Aided Program Synthesis		5. TYPE OF REPORT & PERIOD COVERED (9) Technical Rept.
		6. PERFORMING ORG. REPORT NUMBER (14) TR-861
7. AUTHOR(s) (10) Richard J. Wood		8. CONTRACT OR GRANT NUMBER(s) (15) N00014-76-C-0477
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of Maryland College Park, MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 12 13
11. CONTROLLING OFFICE NAME AND ADDRESS Information Systems Branch (11) Office of Naval Research Washington, DC 20305		12. REPORT DATE January 1980 13. NUMBER OF PAGES 33
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Knowledge-based systems, Computer aided programming, Automatic programming, Interactive design systems, Artificial Intelligence

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper proposes the construction of a programming system that will interact with a domain expert user to develop a computer program. Activities evident during a client-consultant interaction are identified and examined, and a new program structure called the Program Model (PM) is presented. The PM explicitly represents the definitions and refinements of the problem domain, the development of an algorithmic solution, and the programming

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract (con't)

knowledge used during synthesis. The types of processing required for a computer aided synthesis system are examined and a control scheme is proposed for managing non-linear program development.