IMPRECISE PROGRAM SPECIFICATION

UNIVERSITY OF SOUTHERN CALIFORNIA

PREPARED FOR
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY

DECEMBER 1975

ISI/RR-75-36
December 1975

Robert M. Balzer

Imprecise Program Specification

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RR-75-36	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Imprecise Program Specification		5. TYPE OF REPORT & PERIOD COVERED Research Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert M. Balzer		8. CONTRACT OR GRANT NUMBER(s) DAHC 15 72 C 0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/ Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order # 2223 Program Code 3D30 & 3P 10
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd., Arlington, VA 22209		12. REPORT DATE December 1975 13. NUMBER OF PAGES 30
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)

This document is approved for public release and sale; distribution unlimited.

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

-----

18. SUPPLEMENTARY NOTES

Presented at Consiglio Nazionale delle Ricerche ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE meeting on 20 Years of Computer Science, Pisa, Italy, 16 - 19 June, 1975.

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

Automatic programming, domain- independent, imprecise specification, natural-language, nonprocedural language, nonprofessional computer user, problem specification, specification language

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

The first section of this report attempts to characterize the field of automatic programming through a general model describing the stages and processing required. The second discusses a particular project as a specialization of the general model. The final section focuses on the problem of imprecise specifications and how they can be understood.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

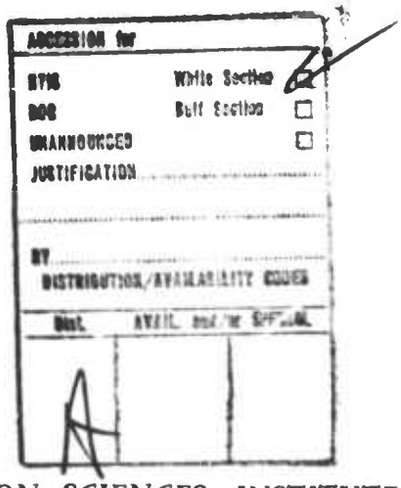
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ISI/RR-75-36
December 1975

Robert M. Balzer

Imprecise Program Specification



INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223, PROGRAM CODE NO. 3D30 AND 3P10.

CONTENTS

FOREWORD

This paper was originally presented at a Meeting on 20 Years of Computer Science, June 16 - 19, 1975, at Pisa, Italy, sponsored by the Instituto di Elaborazione della Informazione under the aegis of the Consiglio Nazionale delle Ricerche. It appeared in the *Proceedings of a Meeting on 20 Years of Computer Science*, Supplement 1 of Volume XII of the journal *Calcolo*

1. INTRODUCTION

It is well known that software is in a desperate state. It is unreliable, delivered late, unresponsive to change, inefficient, and expensive. Furthermore, since it is currently labor-intensive, the situation will further deteriorate as demand increases and labor costs rise. Thus the industry faces one of two choices: either increase the productivity of highly trained, carefully selected specialists or reduce the training requirements through automation, thereby broadening the base of qualified users. Structured programming, built around the concept of discipline, addresses the first path, automatic programming the second. We feel that the first approach will perpetuate the current crisis as systems continue to become more complex. Only automating the process can control the enormous complexity, improve the reliability, modifiability, and efficiency, and reduce the cost. For this approach to be successful, the system must acquire and use a semantic description of a domain--a particular universe of discourse--to understand the user's statements, fill in omitted details, and maintain consistency.

A GLOBAL VIEW OF AUTOMATIC PROGRAMMING

This section presents the author's personal framework* for characterizing automatic programming systems in terms of how a task is communicated to the system, the method by and time at which the system acquires the knowledge to perform the task, and the characteristics of the resulting program.

One goal of any automatic programming system is to allow its users to state their problem and any advice about its solution in terms natural to the problem. We treat both the native terms of the field and terms from other fields which *users* have found useful to describe and conceptualize problems and solutions as the problem domain terms of a given field. With this definition, we conjecture that the solution of every computable problem can be represented entirely in problem domain terms as a sequence which may involve loops and conditionals of actions in that domain which affect a data base of relationships

---

* This view resulted from the author's discussions with and suggestions from numerous colleagues, for which he is deeply indebted. This section is a condensation of part of a larger work (R. M. Balzer, Automatic Programming, USC/Information Sciences Institute, September 1972 (draft)) which attempts to structure the field by means of the conceptualization expressed here. The interested reader should consult this work, which describes the issues in greater detail and presents supporting evidence.

between the entities of the domain. Included either as part of the data base or as a separate part of the model is the history of the model (i.e., the sequence of actions applied to the model). This logically completes the model and makes questions or actions involving historical information possible. In a strong sense, such a solution is a direct simulation of the domain. The system models at each step what would occur in the domain.

The important part of the above conjecture is that any computable problem can be solved, and hence described, in terms of the problem domain. This enables us to divide the solution into two parts: external and internal. The former is the problem specification given by the user in completely domain-specific terms. The requirement for such users is no longer a comprehensive knowledge of computers, but rather the ability to characterize completely the relevant relationships between entities of the problem domain and the actions in that domain. In addition, such users should have a rough awareness of the problem-solving capability of the system so that they can provide additional help where needed in the form of more appropriate macro-actions, recommendations about the use of certain actions, and/or imperative sequences which will solve all or part of the problem in problem-related terms.

The internal part is concerned first with finding a solution in problem-related terms (if this has not already been provided by the user), second with finding efficient solutions given the available computing resources. Such optimizations occur at two levels beyond what is normally considered optimization. First, at the problem level, recognition that certain entities and/or relationships are irrelevant makes it possible to remove them from the model. Second, since only part of the state of the modelled domain is required (and only at certain points in the solution process) rather than a complete simulation of the model at each step, the system can employ alternative representations which require less maintenance and which either directly mirror the required part of the domain state or allow such parts to be computationally inferred. Such representations may also permit a more direct solution. These optimizations form the main distinction between the code generation part of an automatic programming system and current state-of-the-art compilers.

Thus, our definition of an automatic programming system is one which accepts a problem in terms of a model of the domain, obtains a solution in terms of this model, and produces an efficient computer implementation of this solution in the form of a program.

THE FOUR PHASES: AN OVERVIEW

Automatic programming begins with the application of problem solving to problem statements rather than problem solutions, i.e., with the attempt by a computer system to understand the task being specified. Once the task has been understood, if it is not in the form of a process it must be transformed into one. This is the traditional area of Artificial Intelligence and human program design. The resulting process model must be verified as being the one desired by the user and adequate for the user's problem. If not, it must be modified and transformed by the above steps and reverified. It must then be made into an efficiently running program. This involves automating the ad hoc knowledge of computer science.

A complete automatic programming system thus consists of four major phases: problem acquisition, process transformation, model verification, and automatic coding. Problem acquisition is the process by which the system obtains (1) a description of the problem to be solved or task to be performed in a form processable by the system and (2) the knowledge needed to solve the problem. The result of this phase is a well formed problem and knowledge base which can be manipulated by the system and transformed into a high-level process for solving the problem during process transformation. The third phase is used to verify that this process is the one desired and that it is adequate for the problem solution. The fourth phase, automatic coding, fills in the necessary details, optimizes the process, and produces the actual code to solve the problem.

THE AUTOMATIC PROGRAMMING MODEL

One of the most striking and deep-rooted features of the automatic programming model presented here is the interface it creates between a high-level external specification of a problem which omits data structures that are not part of the domain and the internal implementation of that specification in an efficient representation.

This choice of a basic interface has predicated large parts of the entire model. This choice as the basic interface within the automatic programming model can be expected to provide four important gains. First, the complete model conjecture states that such a division is feasible for stating and solving domain-dependent problems. Second, since the choice of a data representation and the maintenance of its consistency occupy such a large portion of current programs, their omission should drastically reduce the size and complexity of the resulting specifications. Third, since so much detail has been removed from the specification, it is easier for the system to understand what the task is rather than to become lost in the details of what is going on. Finally, since the problem has not

been overspecified with a particular choice of representation in order to express it the system is now free to choose a representation that will efficiently solve the problem at hand. The system has been given increased flexibility in its choice and may well outperform humans in correctly making representation choices; this is true not because the system is more intelligent than the user, but because it can cycle through more possibilities and bring to bear a greater level of effort in such optimizations than any user is willing or able to invest in such issues.

Problem Acquisition

The problem acquisition phase is concerned with obtaining an understanding of the user's problem and the domain in which it exists so that the process transformation phase can attempt to find a sequence of transformations or operations in that domain that will obtain the solution required by the user. Thus, the problem acquisition phase is concerned with building a model of the user's domain that represents the interactions between the entities of that domain and the effect on those entities by the allowed transformations or applicable operations. Only by developing such a model of the user's domain can the automatic programming system have any degree of generality in the domains for which it is applicable.

Currently, all such models of user domains have been coded into a system. It is proposed here that such models can be specified to the system by its users and that through these models the system can acquire the knowledge necessary to solve problems within these domains and to understand what is required for such a solution. The two main issues, then, are what constitutes an adequate and appropriate model and how is such a model specified or communicated to the system.

The adequacy of a model is dependent upon its use in solving the problem. Operationally, this requires that the automatic programming system be capable of finding the complete set of applicable transformations on the model and be able to calculate the consequences of each of these actions. The appropriateness of the model is a measure of how well suited the available transformations are to solving the problem at hand, i.e., an adequate model can be made more appropriate by adding to it nonprimitive transformations made up of a sequence of primitive ones, which are suitable building blocks for the problem being posed. The model may also be made more appropriate by including recommendations about the suitability of alternative strategies for sequences of model transformations. Users can significantly reduce the well known problem of building a powerful general-purpose problem solver by tailoring the specified model to make it more appropriate for the problem at hand.

The basic viewpoint, then, is to process the user's natural language communication with the understanding that it is meant to convey to the automatic programming system a model of his problem domain. Towards this end the system can extract entities and the relationships between them from the communication. It can further query the user as to the relationships between entities which have not as yet been explicitly specified but which have been inferred by the previous communication. Such inferences by the system about the completeness of the model require a sophisticated understanding not only of the communication but of the types of models used for problem domain specification. Unfortunately, our sophistication in both these areas is quite limited. In communication we need to be able to understand how information is ordered for presentation, how context is established and utilized, how the capabilities of the recipient affects the communication, and how these capabilities are perceived by the speaker. In modelling we need to have a space of possible models, an understanding of how the parts of a model interact, a means for recognizing incompleteness and inconsistencies in models, a means for obtaining all the allowed operations on the model, and the means for transforming the models with these operations.

Process Transformation

Our contention is that the main activity in programming is not finding a solution but in finding a solution which omits the irrelevancies and abstracts the necessary processing for efficient implementation. This is a strong contention, but for most programming problems a solution is known; the main concern is finding a more efficient one. This is not optimization in the normal sense of the term. The concern, rather, is with finding irrelevancies in the complete model and representational abstractions based on the required processing of that model. Once these logical representations have been found, they must be efficiently implemented.

The above contention, if true, greatly shifts the emphasis within the process transformation phase from that of a general purpose problem solver solving problems in a domain-independent way to modifying a solution so that it does not maintain any irrelevant portions of the complete model and abstracting the relevant portions into a more efficient representation for the processing required. Together with problem acquisition the ability to find representational abstractions and transform complete model solutions into those which utilize these representations represents the main technological drawbacks to obtaining an Automatic Programming system.

Model Verification

Although the Automatic Coding phase will produce only correct code, program testing cannot disappear. This is because problem acquisition and process transformation

will undoubtedly employ a number of heuristics and may very well incorrectly interpret either the problem statement or the allowed transformations that can occur in the user's model. Because of this, the user must verify that the system created is the one that he desired.

The technology for this is at hand. It consists of today's methods wherein a test case is given to the system and its performance is used to validate the model that it constructed. Additionally, the system can aid the process by generating test cases of its own which probe uncertain areas that could have led to either misunderstanding or incompleteness in the original model. One might also expect that program debugging would disappear, but for very similar reasons it too will remain under automatic programming. If there is a disparity between the user's model and the system's model, then the reason for this disparity must be ascertained.

Automatic Coding

Automatic coding is concerned with finding an efficient computer implementation of the process description obtained from the preceding phase. This description does not yet include a choice of data representations, but does specify the major processing elements and sequences. It is intended that this phase will not need any domain-specific knowledge except for input frequency and distribution information. The major logical representation and processing decisions have already been made in process transformation.

Of all the phases in the Automatic Programming system, automatic coding is the one essential component. Without it the system cannot produce programs, and hence, though it may be useful, it cannot be an Automatic Programming system.

Most people are not truly creative when they reorganize sections of their program to increase efficiency. Rather than inventing totally new representations, they appear to select one out of an ill-defined set of such possible representations and adapt or modify it to function in the current situation. This is probably the main challenge to the Automatic Coding phase: the ability not only to cycle through a set of alternative representations, but also to adapt and modify them to the existing situation. Such an ability would vastly increase the applicability of a small set of alternative representations.

From such automatic coding studies, one would expect to see both a set of heuristics and, eventually, a calculus for data representation choices.

2. THE AUTOMATIC PROGRAMMING PROJECT

The goal of ISI's Automatic Programming project is simply to allow experts in an application area (who are not programmers) to functionally specify their application directly to a computer system, with the system transforming this input into a precise operational functional specification of the application. Such an accomplishment represents a testable model of the proposed application which could be used as follows:

- To examine the functional behavior of the application against the requirements and, if necessary, to modify the functional specifications until they satisfy the requirements.
- As the input to an automatic test data generator which would develop test cases to comprehensively exercise the model.
- As a precise specification of the desired application program from which a human programmer could generate the application and against which the implementation could be tested.
- Eventually, as the specification of the desired application program, for an automatic program optimizer--thus eliminating, ultimately, the need for programmers.

Because programming activities are so diverse, such a system must be capable of accepting specifications for a wide variety of applications.

GENERAL APPROACH

Functionally, the two most important characteristics of our proposed system are its independence from any particular problem domain and its attempt to deal directly with nonprofessional computer users without the intervention of computer programmers--choices which have largely dictated the direction of the project. Domain independence requires that the domain "physics"--its objects and their relationships with other objects, its laws, its transformations, and its constraints--be available in a processable form within the system and that the system be general enough to deal effectively with a wide variety of such physics. Direct interaction with nonprofessional computer users means that both the physics and the problem statements will be in

problem-oriented (as opposed to computer-oriented) terms, preferably in natural language, and that they will be "loose" descriptions containing incomplete, inconsistent, and irrelevant statements rather than a precise formal structure. The primary goal of our system is to acquire from a dialogue with the user the physics of the loosely defined domain, structure it, and use it to understand further communication and to write a program accomplishing the user's stated tasks.

The constraints and restrictions of the computer have increasingly been incorporated into programming advances for several years. They are manifest in better languages, automatic storage mechanisms, and optimizations of many forms. On the other hand, the structure, constraints, and limitations of the problem domain have generally not been thus incorporated. A major theme of automatic programming (in fact the characteristic distinction between it and conventional programming) is the use of such knowledge--an issue which raises a number of questions. If the system is to understand something of a domain, how is the knowledge on which this understanding is based to be represented? What procedures can be made available for exploiting this knowledge in guiding the system's interaction with a user and in generating programs? How, in particular, is the essentially nonprocedural information in constraints and limitations to be reflected in a procedural form? What can be done to help identify inconsistencies? How can the system be given a capacity for inference similar to that which forms the mainstay of human communication and which allows obvious details to be left unspecified? Will the system be able to understand its own products well enough to be able to modify them in response to changed requirements? Answers to these questions define the front on which important advances in automatic programming will be made.

RESTRICTIONS

To concentrate on this knowledge extraction and domain structuring activity, we have assumed the existence of a natural language parser which transforms the user's input into a parsed case structure. Such a parser is currently beyond the state of the art, but this goal is actively being pursued by other groups and we expect it to be available by the time our project is ready to assemble a total system. Until then, we are manually transforming the natural language input into the case structured form required. If such a parser does not materialize, we would have to use a more restrictive and formal subset of natural language.

As a second means of limiting the scope of our work, we have decided to omit efficiency concerns for the programs generated; thus we will focus on generating a logically correct program for the user's needs without attempting to optimize it. This greatly simplifies our effort by allowing us to directly model the user's domain in a data-representation-free manner through an associative data base, hence obtaining

running programs modeling the user's conception of the problem. By not having to introduce extraneous details (such as data representation) during the construction phase, we can concentrate on the program's logical behavior. Furthermore, we firmly believe that such representation-free and behaviorly specified programs are the correct way to program--for both man and machine--and that optimization should occur as a separate and later phase (not part of this project). It is clear that with such an approach the maintenance problem would be greatly simplified. The logical-behavior specification would be modified and the program reoptimized.

SPECIFIC APPROACH

We are building a system with two major components--domain acquisition and model completion. The domain acquisition component sequentially processes a set of statements describing the user's problem and the domain in which it exists. This component is responsible for extracting from these statements the description of the object being manipulated, the actions performed on them, the criteria necessary and sufficient to perform these actions, the constraints which must be satisfied, and the rules for inferring information not explicitly stated. This information may be given directly, may be inferred from example usage, or may be assumed in order to make sense of the input. Some of this information may have been previously acquired and saved in a domain description.

This component is implemented through a production system in which each transformation rule has a pattern which, if found in the input, activates the rule. An activated rule will typically assert some extracted knowledge in the associative data base and rewrite the input with the extracted information omitted or transformed. This activation process is continued until no rule matches the (transformed) input. Then the next input is processed.

A production schema was chosen because of its orientation toward case analysis, its facility for expanding as new rules are added, and its ability to accept manual transformations for unimplemented rules.

During these transformations, when an ambiguous interpretation is noted, one of three actions is taken: the problem can be kept for later processing in the hope that new information will resolve the ambiguity; the user can be asked directly to resolve the ambiguity; or the system can establish a backtracking point, assume one interpretation, and be prepared to back up and assume the other. Currently, only the first two options are used, since our system has no backtracking capability.

The model completion component is responsible for all interstatement processing. Its main function is to form a program by organizing the actions referenced in the

individual statements into an appropriate control structure. These actions are organized into sequential segments or asynchronously activated demons in a two-stage process. First, the needs, requirements, and results of each action are analyzed to determine any implicit ordering restrictions. This partial ordering is then merged with any explicit partial ordering specified in the input to produce the final ordering restrictions. The second stage determines which actions should be treated as asynchronous demons and removes them from the ordering. It then attempts to find a total ordering consistent with the restrictions. Finally, all action descriptions, action invocations, and object references are transformed into an executable form.

CURRENT STATUS

We decided to develop our system in the context of a real-world (albeit simplified) problem. Having selected the significant domain of automatic message distribution, we have extracted from an existing functional specifications manual a short, simplified, and very-high-level loose description of a real implemented system.

With the help of some manual transformations this description has been processed and analyzed by the domain acquisition component. The model completion component is largely unimplemented, but one part which takes the requirements and results of the actions described and produces the implicit partial ordering is working. Furthermore, it identifies the inputs and outputs of the system by finding, respectively, the information used but never produced and the information produced but never used.

The task specification, an example of the manually parsed input, and the structured knowledge extracted from the input is given below:

English Description of Message Distribution System

1: MESSAGES RECEIVED FROM THE AUTODIN-ASC ARE ROUTED FOR SERVICE-ACTION IF REQUIRED AND THEN PROCESSED FOR AUTOMATIC DISTRIBUTION ASSIGNMENT

2: IF THIS ASSIGNMENT CANNOT BE PERFORMED AUTOMATICALLY BY THE SYSTEM, THE MESSAGE IS DISPLAYED ON THE CRT FOR AN OPERATOR TO PERFORM THE ASSIGNMENT MANUALLY

3: THE MESSAGE IS THEN DISTRIBUTED TO EACH ASSIGNED OFFICE

4: EACH MESSAGE IS ASSIGNED FOR ACTION TO A SINGLE OFFICE WHICH IS REFERRED TO AS THE ACTION OFFICE

5: THE NUMBER OF COPIES OF A MESSAGE SENT TO AN OFFICE IS A FUNCTION OF WHETHER THE OFFICE IS ASSIGNED FOR ACTION OR INFORMATION

6: MESSAGES THAT ARE CLASSIFIED TOP-SECRET AND THOSE WITH SPECIAL-HANDLING INSTRUCTIONS ARE NOT ALLOWED TO BE ASSIGNED AUTOMATICALLY BUT ARE FORCED TO MANUAL DISTRIBUTION ASSIGNMENT

7: SERVICE MESSAGES ARE IDENTIFIED BY COMMUNICATION-ACTION CODES IN THE CONTENT-INDICATOR CODE FIELD OR BY KEYS IN THE TEXT

8: THESE MESSAGES REQUIRE SOME TYPE OF SERVICE-ACTION AND SHOULD BE PRINTED FOR THE SERVICE-SECTION

9: THE RULES FOR EDITING MESSAGES ARE (1) REPLACE ALL LINE FEEDS WITH SPACES (2) SAVE ONLY ALPHANUMERIC CHARACTERS AND SPACES AND THEN (3) ELIMINATE ALL REDUNDANT SPACES

10: IT IS NECESSARY TO EDIT THE TEXT PORTION OF THE MESSAGE

11: ALL MESSAGES ARE SEARCHED FOR ALL KEYS

12: ASSOCIATED WITH EACH TYPE OF KEY IS AN ACTION TO BE PERFORMED WHEN A KEY OF THAT TYPE IS LOCATED IN A MESSAGE

13: THE ACTION FOR TYPE-0 KEYS IS: IF NO ACTION OFFICE HAS BEEN ASSIGNED TO THE MESSAGE, THE ACTION OFFICE FROM THE KEY IS ASSIGNED AS THE ACTION OFFICE OF THE MESSAGE. IF THERE IS ALREADY AN ACTION OFFICE FOR THE MESSAGE, THE ACTION OFFICE IN THE KEY IS TREATED AS AN INFORMATION OFFICE. ALL INFORMATION OFFICES IN THE KEY ARE ASSIGNED TO THE MESSAGE IF THEY HAVE NOT ALREADY BEEN ASSIGNED AS ACTION OR INFORMATION OFFICES.

14: THE ACTION FOR TYPE-1 KEYS IS: IF ANY TYPE-1 KEY IS FOUND IN THE MESSAGE, THE FIRST ONE FOUND IS USED TO DETERMINE THE ACTION OFFICE AND ALL OTHER KEYS ARE USED ONLY TO ASSIGN INFORMATION OFFICES

Actual Input for Message Distribution Example

```
[RPAQQ
 FS1
 (INPUT-SENTENCE
  (SOURCE\TEXT (MESSAGES RECEIVED FROM THE AUTODIN-ASC ARE ROUTED
                    FOR SERVICE-ACTION IF REQUIRED AND THEN
                    PROCESSED FOR AUTOMATIC DISTRIBUTION
                    ASSIGNMENT))
  (FS-NOTATION
   (CONJOINED
    (CONJUNCTION AND-THEN)
    (CONJ-ARGS
     ([FSIF
       [PRED (NFS (HEAD EVENT*REQ)
                  (ACTION REQUIRE)
                  (OBJECT (NFS (HEAD EVENT*SA)
                               (ACTION SERVICE-ACTION]
       (THEN
        (NFS
         (HEAD EVENT*RT)
         (ACTION ROUTE)
         [OBJECT
          (NFS (HEAD MESSAGE*1)
               (NBR PLURAL)
               (REL (NFS (HEAD EVENT*RCV)
                         (ACTION RECEIVE)
                         (OBJECT MESSAGE*1)
                         (FROM (NFS (HEAD "AUTODIN-ASC")
                                    (DET THE]
         (FOR EVENT*SA]
       (NFS (HEAD EVENT*PRC)
            (ACTION PROCESS)
            (MOOD DCL)
            (OBJECT MESSAGE*1)
            (FOR (NFS (HEAD EVENT*ASG)
                      (ACTION ASSIGN)
                      (MOD AUTOMATIC)
                      (MOD (NFS (HEAD EVENT*DST)
                                (ACTION DISTRIBUTE]
```

Structural Knowledge Extracted From Input

1. NEW TYPES

TYPE	IMMEDIATE SUPERTYPE	INSTANCES, IF ANY
CODE		
CODE-OPERATIONS		"COMMUNICATION-ACTION"
CODE-TYPE		CONTENT-INDICATOR-CODE
CONTENT-INDICATOR CODE	CODE	
INSTRUCTION		
INSTRUCTION-CATEGORY		"SPECIAL-HANDLING"
KEY		
KEY-CLASS		"TYPE-0", "TYPE-1"
KEY-TYPE		"TYPE-0-KEY, TYPE-1-KEY
MESSAGE		
MESSAGE-CLASS-A		"TOP-SECRET"
MESSAGE-CLASS-B		"SECRET"
OFFICE	LOCATION	
SERVICE-MESSAGE	MESSAGE	
TEXT	ORDERED/SET	
TYPE-0-KEY	KEY	
TYPE-1-KEY	KEY	

II. PART-OF RELATIONS

RELATION	DOMAIN-TYPE	RANGE-TYPE
CONT-IND-CODE-PART	MESSAGE	CONTENT-INDICATOR-CODE
CONT-IND-CODE-SUBPART	CONTENT-INDICATOR-CODE	CODE-OPERATION
INSTRUCTION-PART	MESSAGE	INSTRUCTION
KEY-IN-TEXT	TEXT	KEY
TEXT-PART	MESSAGE	TEXT

(NOTE: KEY-IN-TEXT IS THE SAME AS "MATCH-SUBSEQUENCE," BUT THIS CAN ONLY BE DETERMINED BY INFERENCE.)

III. OTHER RELATIONS

RELATION	ARGUMENT-TYPES	
ACTION-FOR-KEY-TYPE	KEY-TYPE	EVENT
KEY-TYPE-FOR	KEY	KEY-TYPE
MESSAGE-CLASS-A-FOR	MESSAGE	MESSAGE-CLASS-A
MESSAGE-CLASS-B-FOR	MESSAGE	MESSAGE-CLASS-B
NUMBER-OF-COPIES-OF-MESSAGES	CONDITION	NUMBER
OFFICE-FOR-MESSAGE	MESSAGE	OFFICE
OFFICE-FOR-KEY	KEY	OFFICE

IV. INFERENCE RULES

(1) (KEY-TYPE-FOR KEY "TYPE-0")
iff
(AIO KEY TYPE-0-KEY)

(2) (KEY-TYPE-FOR KEY "TYPE-1")
iff
(AIO KEY TYPE-1-KEY)

(3) (MESSAGE-CLASS B-FOR MESSAGE "SERVICE")
iff
(AIO MESSAGE SERVICE-MESSAGE)

V. OTHER TUPLES ASSERTED

(AIO "AUTODIN-ASC" LOCATION)

(ACTION-FOR-KEY-TYPE TYPE-0-KEY EVENT)
(ACTION-FOR-KEY-TYPE TYPE-1-KEY EVENT)

VI. NEW ACTIONS

ACTION	ARGUMENT-TYPES
ASSIGN	MESSAGE, OFFICE, RELATION
DETERMINE	KEY OFFICE
DISTRIBUTE	MESSAGE OFFICE
EDIT	MESSAGE
RECEIVE	MESSAGE LOCATION LOCATION
ROUTE	MESSAGE LOCATION LOCATION
TREAT-AS	ENTITY TYPE EVENT

INFORMATION INTERPRETED BUT LEFT ENCODED IN "EVENTS" AND "OBJECT DESCRIPTORS"
(other than case-argument pairings)

"The system	= APSYSTEM
"Manual" X	= (PERFORMED-BY X USER)
"Automatic" X	= (PERFORMED-BY X APSYSTEM)
"The CRT"	= OPERATOR-CRT
"cannot" (in FS2)	= result (of assign by APSYSTEM) = FAILURE
"number of copies"	= x, st.(CARDINALITY Y X),
where Y	= some set of copies of a message (see below).
"copy of a message"	= result of performing action COPY on message
"message to an office"	= some specialization of TRANSMIT is the ACTION of an EVENT in which "message" is the transmitted object and "an office" is the goal location.
"necessary" (FS10)	= (REQUIRED-OF ? edit...)

PLANS

By the end of the year we expect the system to be able to handle this entire example without manual assistance. This will require (1) replacing all the manual transformations in domain acquisition with implemented rules; (2) implementing the model completion component and connecting it to the prior phase; and (3) implementing a module to collect needed input data for the program generated. In addition, we plan to develop an execution monitoring capability to enable a user to watch the generated program operate as a debugging aid.

Our example contains two known errors, one of which could be spotted by a bug apprehension system we have begun to plan. It is caused by producing, under certain circumstances, a data value after it has been used to control program flow. This error and many other common ones can be spotted as potential (data-dependent) problems by a pattern-directed analysis of the program. Their occurrence could then easily be spotted in actual behavior trace.

The second bug concerns an interpretation of the English statement, "X is a function of Y." Does this mean that X is a function only of Y or of Y and some other unnamed things? We have chosen the former meaning, although the latter was intended in the example. This interpretation will cause a bug in the generated program which can be spotted only by observing its behavior.

We then plan to select and present to our system several different real-world domains of approximately the same complexity as the message-distribution domain. Although we have tried to build a domain-independent system, we have been driven by our example in that we have built only those transformations required by the example. Thus, as we address new domains, more transformations will become necessary to handle new situations previously unencountered. The new transformations may interfere with the existing ones. We will have to identify and resolve such conflicts.

The main goal of these studies will be to determine the generality of our system in terms of the amount of overlap, and the amount of conflict, with existing facilities. In some sense, we must develop an estimation of the size of the "vocabulary" (i.e., the facilities) needed to handle domain descriptions. We will also be studying how to specify a domain and program how to represent them in the system.

This understanding of domain and program descriptions will allow us to accept more imprecise and incomplete specifications by resolving or filling in information from information specified elsewhere and through knowledge of domain structures and

interrelationships. We will continue to push on this front until we can handle specifications typically found in functional specification manuals.

If we were totally successful in attaining domain independence, then new domains could be accepted without any modification of the system by merely providing their domain description. We do not expect to achieve such a level of independence. However, our goal is to minimize such modification so that by the end of 1976 we can acquire and handle a new domain of roughly the size and complexity of the message-distribution domain in less than a week.

3. *IMPRECISE SPECIFICATIONS*

There are three main problems in transforming a specification into a program. The first, efficiency, has been explicitly excluded from our consideration. The second, transforming nonprocedural specifications into procedural ones, such as constructing programs so that stated constraints cannot be violated, is in general a very difficult problem and has therefore been postponed for later consideration. That leaves only ill-defined or imprecise specifications. This remaining major problem has become our main focus because of the significant improvements which can thus be realized.

The notion of imprecise specifications is itself imprecise. By imprecise, we mean information which is not explicit in any statement but is implicit in some group of statements and context. We do not have a complete categorization of the ways in which a specification can be deficient, nor do we understand all the boundaries. But our approach is engineering-based rather than mathematical. Rather than attempting to handle *all* cases, we are looking for those which arise frequently in natural language communication between two people. Our assumption is that the user is attempting to be helpful and that something is imprecise only because either it doesn't matter or because for the speaker one, and only one, interpretation is obvious and hence the meaning is unambiguous. Therefore, removing the imprecision should nearly always be simple and involve only shallow reasoning.

We list below the types of imprecisions we currently handle or plan to handle, a specific example drawn from the problem presented in the previous section (the numbers in the square brackets identify the sentence numbers), and a discussion of how such imprecisions can be handled.

1. Complete parameter specifications for events (actions or relations).

   A. Disambiguation by well-formedness criteria of *IF* statement -

      "All information offices in the key are assigned to the message if they have not already been assigned as action or information offices" [S13]

      The second *ASSIGN* in the sentence doesn't specify to what the office is assigned. From previous specializations, we find it could be

to either *KEYS* or *MESSAGES*. During a meta-evaluation phase the program is tested for well-formedness which, among other things, requires that the value of the predicate of an *IF* statement is not determinable from the program structure itself (i.e., without any knowledge of the data). The office being investigated is known from the first part of the sentence to be assigned to a key ("office in the key" see 5B below). Hence, only if *MESSAGE* is filled in as the unspecified parameter is the *IF* well-formed.

B. Dynamic Context -

"The rules for editing messages are: replace all line feeds with spaces" [S9] and "It is necessary to edit the text portion of the message" [S10]

The set in which the replacement is to be performed is not specified. Lexical analysis indicates that *MESSAGE* is a parameter of *EDIT*, but it is not a set. However, it has several components which are sets *ADDRESSEE*, *TITLE*, *TEXT*, etc). Dynamic context (from sentence 10) indicates that the *TEXT* component should be edited and hence it is the unspecified parameter to *REPLACE*.

C. Modification of parameters -

"The message is distributed to each assigned office" [S3]
"The number of copies of a message to an office ..." [S5]

Sentence three indicates that *MESSAGES* are to be *DISTRIBUTED*. Sentence five further specifies this parameter as being those which are the result of *COPYING* the *MESSAGE*. Thus the call to *DISTRIBUTE* must be modified to be the result of the *COPY* action on the *MESSAGE* which was originally thought to be the parameter to *DISTRIBUTE*.

2. Sequencing

A. Loop Formation

"Messages received from the Autodin ASC are routed ..." [S1]

A set (*MESSAGES*) is specified for the direct object parameter of

ROUTE which is expected to be singular. The causes a loop to be formed around the *ROUTE* action with *MESSAGE* as the iteration variable and controlled by the filter "messages recieved from the Autodin-ASC". This loop is then percolated up through the "if required" and "and then processed" statements which surround the *ROUTE* because they are both dependent on the iteration variable. This brings the loop to the outermost level of the sentence.

B. Demons

"Messages received from the Autodin-ASC are routed ..." [S1]

A loop at the top level of a sentence which is not explicitly sequenced relative to other statements is treated as a loop distributed in time--a demon--which is fired whenever its controlling filter is satisfied.

C. Purpose

"Processed for automatic distribution assignment" [S1]

If an action (*PROCESS*) is performed for the purpose of enabling another (*ASSIGNMENT*) which is not explicitly sequenced, then have it precede the enabled action. Thus although it is never explicitly invoked in the specification, we infer that *ASSIGNMENT* should follow *PROCESS* and similarly that *DISTRIBUTION* should follow *ASSIGNMENT*.

D. Explicit Sequencing

"The message is then searched for all keys" [S11]

The *SEARCH* is made to follow the event of the previous sentence (*EDIT*).

E. Remote Loops

"The number of copies of a message to an office ..." [S5]

As mentioned in 1C above, this sentence modifies the invocation of

DISTRIBUTION of *MESSAGES*. It changes the actual parameters from *MESSAGE* to a specified *NUMBER OF COPIES*. This causes a loop to be formed around the *COPY* action, which in turn causes the loop to be percolated up around the *DISTRIBUTE* since it cannot take a set as a parameter.

F. Requirements Analysis

"The message is then searched for all keys" [S11]

This informs us that *SEARCH* follows *EDIT*, but the placement of this pair relative to the other actions is not known. Therefore an analysis of the pre- and post-conditions of each action is undertaken to discover any unstated sequencing rules. This analysis shows that the *ASSIGNMENT* is caused by actions performed only when a *KEY* is *LOCATED*. Since *LOCATE* is a successful *SEARCH*, *SEARCH* must precede *ASSIGNMENT*.

G. *IF-THEN* Sequences

"If no action office has been assigned to the message, ... If there is already an action office for the message, ..." [S13]

This sentence is of the form "if P then X; if not (P) then Y" and should be interpreted as "if P then X; else Y". More generally, several *IF* statements following each other should be treated as a *CASE* statement rather than a sequence of *IF* statements.

3. Time Frame

A. Passive Voice

"Messages received from Autodin-ASC are routed" [S1]

Does this mean that when *RECEIVED MESSAGES* have already been *ROUTED*, or upon *RECEIPT* they should then be *ROUTED*? Such statements are interpreted as either a test or an action invocation. The critical issue is that the interpretation should be the same for all items. The problem is that, in general, this cannot be determined made at specification time. Thus, this imprecision is left until the first usage, which examines the situation existing at that point and determines the interpretation that should be used thereafter.

B. Positive Constraints

"Each message must be assigned to a single office for action" [S4]

Again, the interpretation is not clear. Does it mean that the *ASSIGNMENT* should have already been made or that such an *ASSIGNMENT* must now be made? Our interpretation is "if not (test) then perform". That is, if the condition has not already been met, then meet it (if possible). One further imprecision remains: when should such an interpretation be applied? Whereas negative constraints apply everywhere (they can never be violated), positive constraints apply only at some particular time, normally unspecified. We default such unspecified times to the first unconditional usage of the event or any of its unique side conditions. For the above positive constraint, this is during *DISTRIBUTION*, when the action is performed for each *ASSIGNED MESSAGE*. The earlier usages in sentence 13 are conditional.

4. Irrelevant Information

A. Indeterminate Specializations

"These messages require some type of service action ..." [S8]

Neither the types of *SERVICE-ACTION* nor the method of determining which one applies in a particular situation is given nor is the distinction used, hence the system assumes the distinction is irrelevant.

B. Indoterminate Sets

"Replace all line feeds with spaces" [S9]

The cardinality of the set of *SPACES* is unspecified and hence is assumed not to matter. Two is assumed.

5. Reference

A. Uniformity

"and then processed for automatic distribution assignment" [S1]

PROCESS is clearly a dummy name for some more specific actions which enable *ASSIGNMENT*. Unfortunately this is not specified. However, *EDIT* followed by *SEARCH* perform the function of enabling *ASSIGNMENT* and are not explicitly sequenced but by requirements analysis must precede *ASSIGNMENT*. It should therefore be assumed that the definition of *PROCESS* is *EDIT* followed by *SEARCH*.

B. Generalized Relations

"The action office from the key" [S13]

Prepositions like *FROM*, *IN*, and *OF* often are not part of the case frame for the omitted relation between the entities on either side of the preposition. Instead they imply that the entity on the left is *ASSOCIATED-WITH* the one on the right. The system responds to such generalized relations by searching for a known relation between the two entities (here *ASSIGN* an *OFFICE* to a *KEY* for *ACTION*).

6. Implied Relations

A. Use of Known Attribute Values

"... are not allowed to be assigned automatically" [S6]

AUTOMATIC is known to be an attribute value of the *PERFORMED-BY* relation which specifies who actually performs an action (here the *ASSIGN* is performed by the system).

B. Use of Unknown Attribute Values

"Top-secret messages" [S6]

TOP-SECRET is an unknown attribute. A new named relation is created which links *MESSAGE* with an named range of which *TOP-SECRET* is an element. It is assumed that the attribute values in this range are mutually exclusive, and that other unknown adjective modifiers of this same type of object (*MESSAGE*) also belong to this range.

4. *CONCLUSION*

In this report we have tried to present a particular view of Automatic Programming as a field, examine a single project consistent with this view working on specification acquisition, and discuss several different forms of imprecision and a possible method of coping with them. This approach is based on applying analysis and problem solving techniques to the problem statement, not to solve it, but rather to understand it. Knowledge of the characteristics of well-formed specifications, of how people specify tasks, and a domain description to provide redundancy disambiguates natural communication to a great extent.

Though such an approach is far from producing practical results, it does offer the eventual promise of removing the major remaining barrier to society's effective use of computers, i.e., the ability to specify tasks at a level appropriate for human communication with automated implementations rather than in a highly formalized notation requiring excessive training, attention to details and optimization, and associated high costs. Only then can the promise of computers--the ultimate malleable object--be widely realized.