INFORMALITY IN PROGRAM SPECIFICATIONS

Robert Balzer, Neil Goldman and David Wile
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina Del Rey, California 90291

ABSTRACT

This paper is concerned primarily with (1) the procedure by which process-oriented specifications are obtained from goal-oriented requirement specifications and (2) computer-based tools for their construction. It first determines some attributes of a suitable process-oriented specification language, then examines the reasons why specifications would still be difficult to write in such a language. The key to overcoming these difficulties seems to be the careful introduction of informality based on partial, rather than complete, descriptions and the use of a computer-based tool that uses context extensively to complete these descriptions during the process of constructing a well-formed specification. Some results obtained by a running prototype of such a computer-based tool on a few informal example specifications are presented and, finally, some of the techniques used by this phototype system are discussed.

/. *INTRODUCTION*

A critical step in the development of a software system occurs when its goal-oriented requirements specification is transformed into a process-oriented form that specifies how the requirements are to be achieved. Only after this transformation has occurred can the feasibility of the system be analyzed and the consistency of the process specification with the requirements be verified. The key to this transformation is expressing the process-oriented specification abstractly so that its functionality is completely determined while the class of possible implementations remains broad.

We believe that such abstract process-oriented specifications are the key to rationalizing the software development process. Such specifications are, in reality, programs written in a very high level abstract programming language. As such, they could provide an effective interface between the two major software concerns: functionality and efficiency. These concerns should be decoupled so that the functionality of a system can be addressed before its efficiency has been considered. Once functionality has been accepted, it can be preserved while the system is optimized. Thus, since the abstract process-oriented specification is a program, its consistency with the requirements could be formally verified, informally argued, or tested by actually executing the specification. Furthermore, the end user could be given hands-on experience exercising the specification to see if it behaved as intended. Deviations and/or inconsistencies could be corrected in the specification before any implementation occurred.

Once the system's functionality has been accepted by the user, the efficiency of the system in meeting its performance requirements remains an issue. Such efficiency must be gained without altering the system's accepted functionality. We have argued elsewhere [1] that a computer-based tool can be built which guarantees maintenance of functionality while a program is optimized without sacrificing the programmer's ingenuity or initiative in determining how best to achieve efficiency.

NOTE: This research was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. DAHC15 72 C 0308, ARPA Order No. 2223, Program Codes 3D30 and 3P10.

In this paper we are concerned primarily with the procedure by which such process-oriented specifications are obtained and with computer-based tools for their construction. We will begin by determining some attributes of a suitable process-oriented specification language, then examine why specifications would still be difficult to write in such a language. We will argue that the key to overcoming these difficulties is the careful introduction of informality based on partial, rather than complete, descriptions and the use of a computer-based tool which utilizes context extensively to complete these descriptions during the process of constructing a well formed specification. We will then present some results obtained by a prototype of such a computer-based tool on a few informal example specifications. Finally, we will discuss some of the techniques used by this prototype system.

2. ATTRIBUTES OF SUITABLE PROCESS-ORIENTED SPECIFICATION LANGUAGES

As stated above, a suitable process-oriented specification must completely define functionality, represent a broad class of possible implementations, and be executable.

How can we obtain such a language? We begin by noting that a suitably abstract programming language is a specification language. Several recent languages almost meet the above requirements for an executable specification language. They have arisen from two separate disciplines:

1. *Specification Languages.* Languages, such as RSL[2], PSL[3], etc., designed specifically for specification, describe a system in terms of data flows *and* processing units but do not functionally define the processing. Such languages can provide a simulation of the described system down to some level of detail, but cannot describe or simulate its full functionality.

2. *Abstract programming Languages.* Spawned by Dijkstra's notions of structuring, a generation of programming languages (CLU[4], Alphard[5], Euclid[6], Pearl[7]) has bloomed which isolate the definition of data objects, and the operations allowed on them, from their use and manipulation in the program. The result is the ability to use abstract program entities which model those that occur in the application being programmed. These entities are defined in terms of more computer-science-oriented entities, which are, in turn, defined in terms of more primitive ones, until the primitive objects and operations of the language are reached. Without the successive refinements of the abstract objects and operations, these languages would be suitable for specification, except that they would then lose their property of executability. Their executability has been gained at the expense of complete specification of implementation (down to the base level of the language).

What is clearly needed, then, is a language which can fully specify a system functionally without fully specifying its implementation. What are the required properties of such a language?

First, it must be able to define and manipulate application-oriented objects (as is done by the abstract programming languages). Second, the description of these objects and operations must be in terms of some formalism that does not require successive refinement to gain functionality and that does not overly constrain the

implementation. This is the Key issue that would enable specification and programming languages to be unified.

Three formalisms have been proposed for this role: sets, axiomatic specification, and relational data bases.

One of the earliest efforts is Jack Schwartz's SETL[8] language. Sets are the single abstract type allowed for which multiple implementations exist. All the operations on sets can deal with any of the implementations. Thus, users need not be concerned with any of these implementations while specifying the manipulations to be performed on their sets. Because functionality was completely captured by the SETL definitions of sets, implementation did not have to be considered. However, such implementation-free functionality existed only for sets and was not extensible.

More recently, Guttag, Horowitz, and Musser [9] have discussed an axiomatic specification technique in which the functional behavior of new abstract objects are axiomatically defined by algebraic equations. These algebraic equations act as functional requirements which any implementation of the objects and operations upon them must satisfy. Furthermore, they provide a way of executing programs using the operations directly without providing any implementation. Whenever *an* operation is performed on an object, the "state" of that object is transformed by applying the algebraic equation for that operation to the existing "state." The resulting state is just another expression in the algebra. As more and more operations are performed, these states become more complex. However, the states can be simplified by general rules of the algebra such as AND(A False)->False, or by using the equations for the abstract objects as rewrite rules, such as for a stack, POP(PUSH(A x))=>A. Such equivalence rules are part of the functional definition of the operations on the abstract objects. If the axiomatic functional definitions are complete, then specifications in this language can be directly executed while no implementation need be selected and the choice of possibilities has not been constrained. These axiomatic functional definitions provide a user the capability of adding arbitrary new abstract types to the language that can be manipulated in an implementation independent way. This extensible capability is exactly analogous to SETL's built-in capability to manipulate sets in an implementation-independent way.

Finally, we have languages in which the "state" is represented by a series of assertions in a relational data base, rather than by an expression, and in which the effects of an action are expressed as a series of additions or deletions to the data base rather than as an equation to be applied to the "state." The big difference between these two approaches is that in the axiomatic approach the functional definitions are expressed as interactions between the operations on a data type and hence do not rely on any more primitive notions. In the relational approach, as in SETL, each operation is functionally defined in terms of how it affects a built-in primitive notion, the relational data base.

The self-defining, or closed, property of axiomatic definitions would seem to favor that approach because each abstract object and its operations can be considered in isolation without relying on outside semantics and without specifying any constraints on the implementation. Unfortunately, this property comes at the expense of expressing the behavior of objects entirely in terms of the operations upon them and the need to express this behavior in the form of algebraic equations so that the equivalence of alternative sequences of operations can be formed (e.g., the POP(PUSH(A x))->A equivalence cited earlier for stacks).

In the relational approach, rather than stressing a closely knit set of types and operations on them, objects are perceived entirely in terms of their relationships with each other and a set of primitive operations which allow these relationships to be built and destroyed and to be extracted. Non-primitive operations exist on the objects, but they merely alter the set of relationships that exist between the objects. This view allows incremental elaboration of objects, their relationships with each other, and operations upon them. Most importantly, this approach enables objects and operations to be modeled almost exactly as they are conceived by the user in his application (as measured by how they are expressed in our most unconstrained form of communication-natural language).

This latter property is the reason we have selected the relational approach: We feel it minimizes the difficulty that a user would have in constructing an operational specification.

3. WHY OPERATIONAL SPECIFICATIONS ARE HARD TO CONSTRUCT

Unfortunately, even when the user's difficulties in constructing operational specifications are minimized by the use of the relational approach, the task remains burdensome and error-prone, primarily because although a suitable language has been chosen, it is still formal. Each reference to an object or action must be consistent and complete. The large number of interacting objects, actions, and relationships require the user to do a great deal of (error-prone) clerical bookkeeping which impedes his attention to the specification itself and reduces its reliability.

Suppose we constructed a computer aid which relieved the user of these clerical chores. How would the specification task be altered? We begin by considering how people specify software systems when unconstrained by computer formalisms.

4. SEMANTIC CONSTRUCTS IN NATURAL LANGUAGE SPECIFICATION

We studied many actual natural language software specifications. The main semantic difference between these specifications and their formal equivalent is that partial descriptions instead of complete descriptions are used. When such partial descriptions are understood it is because they can be completed from the surrounding context. The partial descriptions focus both the writer's and the reader's attention on the relevant issues and condense the specification. Furthermore, the extensive use of context almost totally eliminates bookkeeping operations from the natural language specification. These are some of the properties we find so useful in natural language specifications and which we so sorely lack in formal specification languages.

We have evidence [see sections 5 and 6], in the form of a running prototype system that these properties can be added to a previously formal specification language and that a computer tool can complete the partial description from the existing context. Such a capability is not totally new; it already exists in limited form.

Most programming languages use the context provided by declarations to complete partial descriptions of the operations to be performed on those objects (e.g., ADD becomes either INTEGER-ADD or FLOATING-ADD, depending on the declared attributes of its operands). The Codsyl DBDTG report [10] goes further in the use of context by completing partial references to an item by use of the "current" instance of that item as established by some other statement in the program. Data base declarations are also used to determine how various

program variables *are* to be used in completing partial descriptions of data base items.

These uses of context in programming languages have been accepted, and even championed, because for each use, the context-providing mechanisms are well-defined, the completion rules are simple and direct, and only a single interpretation is valid.

The context mechanisms we are proposing here are much more complex, the context generated much more diffuse, and a given partial description may produce zero, one, or several valid interpretations. Zero valid interpretations means that the partial description is inconsistent with the existing context. A single valid interpretation means that the partial description can be unambiguously completed through use of the existing context. Multiple valid interpretations indicate that sufficient context does not exist to complete the description and that interaction with the user is required to resolve the ambiguity.

Our work should be viewed as an effort to provide more general context mechanisms to resolve the ambiguity introduced in the specification by partial descriptions. If, as we believe, such mechanisms can be provided, would they be a beneficial addition to specification languages?

5. *DESIRABILITY OF INFORMALITY*

We recognize that our approach is controversial and apparently opposes the current trend to make program specifications more and more formal and to introduce such formalisms earlier in the development cycle. We believe closer examination will reveal that our approach is not only compatible with the desire for increased formalism, but a necessary adjunct to it.

Attention has been focused on formalisms for program specification to the exclusion of concern with the difficulty and reliability of creating such formal specifications and with maintaining them during the program life-cycle. Our approach specifically addresses these issues.

First, it should be recognized that informality will always exist during the formulation of a specification. The issue is whether the informal form is explicitly entered into the computer and transformed, with the user's help, into the formal specification, or whether it exists only outside the computer system in someone's head or written somewhere in unanalyzable form. We should consider, then, the feasibility and the desirability of a computer-based tool to aid in the transformation of an informal specification into a formal one.

Let us begin with the question of feasibility. While the results presented in the next section are preliminary and the examples chosen far smaller and simpler than real specifications, we are optimistic about continued progress and ultimate practicality of this approach. However, since these results are far from conclusive, we invite the reader to reach his own conclusions after considering the examples of the next section and the description of the prototype system which follows them.

Assuming for the moment that such a system is feasible, we consider its desirability. Informal specifications have three obvious advantages. First, they are more concise than formal specifications and focus both the specifier's and the reader's attention. They are more concise because only part of the specification is explicit; the rest is implicit and must be extracted from context. Attention is focused on the explicit information and, therefore, away from the implicit information, which increases both the readability and the understandability of the specification.

The second advantage is that informal specifications which employ partial rather than complete descriptions are a familiar, in fact normal, mode of communication. This reduces the training requirements of users, permits a wider set of users, and reduces dependence on the judgment and accuracy of intermediaries.

The final advantage deals with the maintainability of the system. Since about 707. of the total life cycle costs of large systems are for maintenance, any improved capabilities in this *area are very* significant. As we have argued elsewhere [1], the main deterrent to maintainability is optimization. Optimization spreads information throughout a program and increases its complexity through increased interactions among the parts. Both of these optimization effects greatly impede the ability to alter the program. An obvious solution is to alter an unoptimized specification and then reoptimize the program. No cost-effective and reliable technology currently exists for such reoptimization, though one has been proposed

en

A similar situation exists between the informal and formal specifications. The creation of a formal specification involves spreading implicitly specified information throughout the specification and increasing the complexity by structuring the specification into parts and establishing the necessary interfaces between them. As before, both of these formalization effects greatly impede the ability to modify the specification. Again, a solution is obvious: modify the informal specification and retransform it into a revised formal specification. Under the assumed feasibility of our approach, this solution would be possible and would greatly simplify maintaining the formal specification of the system.

We now consider three possible disadvantages of a computer-based tool to aid in transforming an informal specification into a formal one. The first possible disadvantage is that the informal constructs will be misunderstood by the computer tool. This is entirely possible, just as it is when a human intermediary interprets an informal specification. While the computer tool cannot match human performance in understanding the informal specification, it operates much more methodically. It can question the user when it detects that there *are* alternative interpretations of some statement. It can record and make explicit all assumptions it makes in transforming the formal specification. It can paraphrase the informal specification to verify that its interpretation is accurate (the current prototype system records its assumptions and interacts with the user to determine the correct interpretation of unresolved ambiguities, but does not yet contain any paraphrase capabilities). Thus, feedback and interaction with the user can eliminate the problem of possible misinterpretation of the informal specification.

The second possible disadvantage is that the computer-based tool will decrease the reliability of the transformation to a formal specification. If the informal specification exists only outside the computer system, then we must rely on the accuracy of the user or, more often, on some trained intermediary to accurately transform it into a formal specification. This transformation depends upon properly understanding the informal specification (see previous paragraph), then restating it in the required formalism. Once the proper understanding has been obtained, the restatement involves moving information from one place to another and changing its form. History would indicate that such clerical bookkeeping transformations are error-prone and can always be done more reliably by a computer tool. Hence, once the

correct interpretation has been obtained through the use of context and interaction with the user, the restatement of the informal specification into the required formalism can be more reliably performed by the computer-based tool than by the user or his intermediary. Therefore, reliability would be improved rather than reduced by such a tool once understanding was obtained.

Understanding, rather than reliability, thus emerges as the key feasibility issue. One way to improve understanding is to increase the interaction with the user. This leads to the third possible disadvantage: that the required volume of interaction will abrogate the advantage of informality. We do not expect this to be an issue with the current system or its successors, since we feel that its current performance level, as evidenced in the following section, indicates that the required interaction rate would be sufficiently small to prevent annoying or sidetracking the user.

Thus, we conclude that the availability of such a computer-based tool would be highly desirable because it would simplify the creation of a formal specification while increasing the reliability of the formulation process; improve the maintainability of the formal specification; reduce special training requirements; and expand the base of potential users. The question of feasibility, which remains as the paramount issue, rests clearly on the ability to correctly interpret an informal specification. We therefore now present some preliminary results obtained by the prototype system and describe its operation so that the reader can observe its performance level and judge for himself the generality of its context resolution mechanisms and therefore its feasibility.

6. *RESULTS*

This section presents two examples successfully handled by the prototype system. The examples were extracted from actual natural language specification manuals, and the results illustrate the power of the system's context mechanisms. However, our system is a prototype and, as such, it is far from complete. New examples currently expose new problems which are resolved by adding new capabilities to the system. Therefore, until some measure of closure is obtained, it should not be assumed that the prototype will correctly process new examples of the same "complexity" as earlier examples. Our goal is to add each new capability in as general a form as possible so that when it is used in new examples it will function correctly. In this way we expect to "grow" the system as more complex and varied examples are tried.

For each of the examples, we present three figures: the actual parenthesized version of the informal input currently used by the system (to avoid syntactic parsing problems)[II], a manually marked version which indicates some of the informalities to be resolved by the system, and a stylized version of the formal output program produced by the system.

The first example is a system which automatically distributes messages to offices on the basis of a keyword search of the text of the message. Figure 1 gives the informal natural language description. Figure 2 indicates some of the imprecisions contained in this example which must be resolved to obtain the system's formalization of this specification as an operational program (Figure 3).

To give some measure of the amount of imprecision in this example and, therefore, the amount of aid provided by the system, we have compiled the following statistics:

Number of missing operands	■	18
Number of incomplete references	■	22
Number of implicit type conversions	•	9
Number of terminology changes	•	3
Number of refinements or elaborations	-	2
Number of implicit sequencing decisions	-	7

ACTUAL INPUT FOR MESSAGE PROCESSING EXAMPLE

♦((MESSAGES ((RECEIVED) FROM (THE "AUTODIN-ASC"))) (ARE PROCESSED) FOR (AUTOMATIC DISTRIBUTION ASSIGNMENT))

♦((THE MESSAGE) (IS DISTRIBUTED) TO (EACH ((ASSIGNED)) OFFICE))

♦((THE NUMBER OF (COPIES OF (A MESSAGE) ((DISTRIBUTED) TO (AN OFFICE)))) (IS) (A FUNCTION OF (WHETHER ((THE OFFICE) (IS ASSIGNED) FOR (("ACTION") OR ("INFORMATION"))))))

♦((THE RULES FOR ((EDITING) (MESSAGES))) (ARE) (: ((REPLACE) (ALL LINE-FEEDS) WITH (SPACES)) ((SAVE) (ONLY (ALPHANUMERIC CHARACTERS) AND (SPACES))) ((ELIMINATE) (ALL REDUNDANT SPACES))))

♦(((TO EDIT) (THE TEXT PORTION OF (THE MESSAGE))) (IS) (NECESSARY))

♦(THEN (THE MESSAGE) (IS SEARCHED) FOR (ALL KEYS))

♦(WHEN ((A KEY) (IS LOCATED) IN (A MESSAGE)) ((PERFORM) (THE ACTION ((ASSOCIATED) WITH (THAT TYPE OF (KEY))))))

♦((THE ACTION FOR (TYPE-0 KEYS)) (IS) (: (IF ((NO OFFICE) (HAS BEEN ASSIGNED) TO (THE MESSAGE) FOR ("ACTION")) ((THE "ACTION" OFFICE FROM (THE KEY)) (IS ASSIGNED) TO (THE MESSAGE) FOR ("ACTION"))) (IF ((THERE IS) ALREADY (AN "ACTION" OFFICE FOR (THE MESSAGE))) ((THE "ACTION" OFFICE FROM (THE KEY)) (IS TREATED) AS (AN "INFORMATION" OFFICE))) (((LABEL 0FFS1 (ALL "INFORMATION" OFFICES FROM (THE KEY))) (ARE ASSIGNED) TO (THE MESSAGE)) IF ((REF 0FFS1 THEY) (HAVE (NOT) (ALREADY) BEEN ASSIGNED) FOR (("ACTION") OR ("INFORMATION"))))))

♦((THE ACTION FOR (TYPE-1 KEYS)) (IS) (: (IF ((THE KEY) (IS) (THE FIRST TYPE-1 KEY ((FOUND) IN (THE MESSAGE)))) THEN ((THE KEY) (IS USED) TO ((DETERMINE) (THE "ACTION" OFFICE)))) (OTHERWISE (THE KEY) (IS USED) TO ((DETERMINE) (ONLY "INFORMATION" OFFICES)))))

Figure 1

To illustrate how context is used to complete the partial descriptions in the example, we consider a few cases:

1. *Partial sequencing.* Distribution is never explicitly invoked in the informal specification. However, the first sentence indicates that Assignment is performed to enable the Distribution. Hence, Distribution should be explicitly invoked after Assignment.

2. *Missing operand.* Sentence two indicates that the message should be distributed to certain offices—those that are "assigned." But, as can be determined from other usages in the informal specifications, offices can be "assigned" to either messages or keys. This missing operand can be resolved by remembering that Assignment

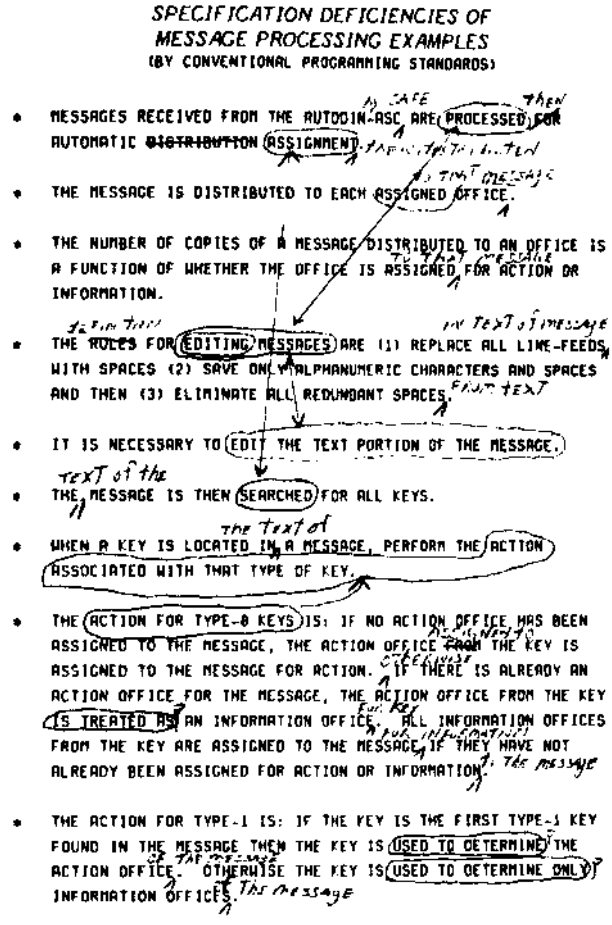

Figure 2

was performed to enable Distribution. Hence, Distribution must use some result of the assignment process. Assignment, from the last two input sentences, assigns offices to the current message. Hence, Distribution must consume offices assigned to that message.

Incomplete reference. Sentence four says to replace all line feeds with spaces. First, replace requires a third operand, some set in which the replacement will occur. Context indicates that this missing operand should be the text of the message parameter of Edit. Second, the use of a plural in the operand of an action which expects a singular operand, indicates an implicit loop. Hence, we have, "for all line feeds, replace the line feed by a space in the text of the message." Now, which line feeds are we concerned with? Only those in the text of the message because they are the only ones which can be replaced. Hence, completing the partial reference, we have "for all line feeds in the text of the message, replace the line feed by a space in the text of the message."

It should be noted that of the approximately 61 decisions which had to be made for this example, all but one were resolved correctly by the prototype system. The message it distributed is the edited one (with all punctuation removed) rather than the original unedited one. The cause of the error is that the system does not understand the difference between an object being changed and its participating in relations with other objects; therefore, it has no concept of the original state of an object and hence does not consider this as a possible completion of any partial reference.

PROGRAM CREATED BY PROTOTYPE SYSTEM

```
(WHENEVER (receive message FROR autodin-asc BY safe)
 DO(odit text OF message)
 (search text OF message FOR (CREATE THE SET OF keys))
 (distribute-processfl message))

(disiribute-process#l (message)
 (FOR ALL (offices assigned TO message FOR ANYTHING)
  (distrtbute-process#2 message office)))

(distribute-process#2 (message office)
 (00 (functional (BOOLEAN (assigned office TO message FOR action))
  (BOOLEAN (assigned office TO message FOR information)))
 TiriES (distribute A copy UH1CH IS A copy OF messago AND located
  AT safe FROM safe TO location OF office)))

(edit (text)
 (FOR ALL line-feeds IN text
  (replace line-feed IN text BY (CREATE SET OF spaces)))
 (keep (union (CREATE THE SET OF alphanumberlc characters IN text)
   (CREATE THE SET OF spaces IN text))
  FROM text)
 (FOR ALL spaces IN text ANO redundant IN text
  (remove space FROM text))

(WHENEVER (locate A key IN text OF message AT POSITION ANYTHING)
 DO (CASE (type OF key)
  (type-8 (type-8-action message key))
   (type-1 (type-1-action message key))))

(type-0-action (message key)
 (IF (NOT (EXISTS action office FOR message))
  THEN (assign THE action officefl FOR key
   TO message FOR action)
 ELSE (treat action office#2 FOR key
   AS information office#2 FOR key
  IN (IF (NOT (assigned office#2 TO message
     FOR action OR information))
   THEN (assign office#2 TO message FOR information))))
 (FOR ALL (offtce#3 assigned TO key FOR information)
  (IF (NOT (assigned office#3 TO message
   FOR action OR information)
   THEN (assign office#3 TO message FOR information))))

(type-1-action (message key)
 (IF key - (key#i UH1CH IS (SEARCH HISTORY FOR FIRST
  (locate type-1 key#I IN text OF message AT position ANY)))
   THEN (determine THE action office FOR message
   BY (type-0-action message key))
  ELSE (determine ONLY THE information office FOR message
   BY (IF (EXISTS action office FOR message)
      THEN (treat action office#I FOR key
         AS information office#I FOR key
         IN (IF (NOT (assigned office #1 TO message
            FOR action OR information))
           THEN (assign office'I TO message FOR information))))
      (FOR ALL office#2 assigned TO key FOR information)
         (IF (NOT (assigned office#2 TO message
           FOR action OR information))
           THEN (assign office#2 TO message
             FOR information))))))
```

Figure 3

This capability can clearly be added to the system, but the important point is that interpretation errors will occur, just as they do when human intermediaries are used to produce the formal specification. It is therefore essential to provide extensive feedback and assumption-testing facilities so that such errors, when made, can be discovered and corrected by the user.

The second example is from a system for scheduling a satellite communication channel by multiplexing it among several users (subscribers). It specifies the component of the system which receives a schedule (SOL) from the controller of the satellite channel and extracts from it the portions of the

next transmission cycle which have been reserved for a particular subscriber and those portions available to any user (RATS). This information is placed in a transmission schedule used by another component to actually utilize the channel during the allowed periods. Figure 4 gives the informal natural language description. Figure 5 indicates some of the imprecisions contained in this example which must be resolved to obtain the system's formalization of the specification as an operational program (Figure 6). In addition to the process description of Figure 4, we have assumed that the formulas referenced and a structural description of the objects of the domain have been separately specified.

The relevant portions of these specifications are that the SOL is an ordered set of subscriber and RATS entries. Each subscriber entry has subscriber identifier and transmission length fields, while a RATS entry has only the latter. The transmission schedule is a set of entries, each of which is composed of an absolute transmission time and a transmission length. One of these entries is the primary entry of the transmission schedule. Finally, formulas 1 and 2 both take an SOL entry as input and produce, respectively, a relative and an absolute transmission time.

Using the same measures of imprecision as in the first example, we find that this example has about half as many imprecisions.

Number of missing operands	"	7
Number of incomplete references	-	12
Number of implicit type conversion	■	3
Number of terminology charges	=	0
Number of refinement or elaboration	-	0
Number of implicit sequencing decisions	=	4

The example is interesting as a test of the generality of the mechanisms which worked on the first example, and because of the new issues it raises. We will examine each of these to illustrate the range of capabilities added to the prototype to enable it to correctly understand this example and produce the operational program of Figure 6.

```
((THE SOL)
 (IS SEARCHED)
 FOR
 (AN ENTRY FOR (THE SUBSCRIBER))))

(IF ((ONE)
    (IS FOUND))
   ((THE SUBSCRIBER'S (RELATIVE TRANSMISSION TIME))
    (IS COMPUTED) ACCORDING-TO ("FORMULA-1"))))

((THE SUBSCRIBER'S (CLOCK TRANSMISSION TIME))
 (IS COMPUTED) ACCORDING-TO ("FORMULA-2")))

[WHEN ((THE (TRANSMISSION TIME))
      (HAS BEEN COMPUTED))
     ((IT)
      (IS INSERTED)
      AS (THE (PRIMARY ENTRY))
      IN (A (TRANSMISSION SCHEDULE])

[FOR (EACH RATS ENTRY)
    (PERFORM)
    (: ((THE RATS'S (RELATIVE TRANSMISSION TIME))
        (IS COMPUTED) ACCORDING-TO ("FORMULA-1"))
       ((THE RATS'S (CLOCK TRANSMISSION TIME))
        (IS COMPUTED) ACCORDING-TO ("FORMULA-2"])

((THE RATS (TRANSMISSION TIMES))
 (ARE ENTERED)
 INTO (THE SCHEDULE)))
```

Figure 4

* THE SOL IS SEARCHED FOR AN ENTRY ~~FOR~~ THE SUBSCRIBER.
* IF (ONE) IS FOUND, THE SUBSCRIBER'S RELATIVE TRANSMISSION TIME IS COMPUTED ~~ACCORDING TO FORMULA-1~~.
* THE SUBSCRIBER'S CLOCK TRANSMISSION TIME IS COMPUTED ACCORDING TO FORMULA-2.
* WHEN THE TRANSMISSION TIME HAS BEEN COMPUTED, IT IS INSERTED AS THE PRIMARY ENTRY IN A TRANSMISSION SCHEDULE.
* FOR EACH RATS ENTRY, THE RATS'S RELATIVE TRANSMISSION TIME IS COMPUTED ACCORDING TO FORMULA-1 AND THE RATS'S CLOCK TRANSMISSION TIME IS COMPUTED ACCORDING TO FORMULA-2.
* ~~THE~~ RATS TRANSMISSION TIMES ARE ENTERED INTO THE SCHEDULE.

Figure 5

```
(build-transmission-schedule (sol subscriber)
   (CREATE transmission-schedule)
   (search sol FOR A subscriber-entry SUCH THAT
    sid OF subscriber EQUALS sid OF subscriber-entry)
   (IF (locate A subscriber-entry SUCH THAT
    sid OF subscriber EQUALS sid
       OF subscriber-entry IN sol)
    THEN
     (MAKE (RESULT-OF (FORMULA-1 subscriber-entry))
      BE THE relative-transmission-time OF subscriber)
     (MAKE (RESULT-OF (FORMULA-2 subscriber-entry))
      BE THE clock-transmission-time OF subscriber))
   (FOR ALL rats WHICH ARE IN sol
       DO (MAKE (RESULT-OF (formula-1 rats))
             BE THE relative-transmission-time OF rats)
          (MAKE (RESULT-OF (formula-2 rats))
             BE THE clock-transmission-time OF rats))
   (FOR ALL clock-transmission-time OF rats
       DO (MAKE clock-transmission-time BE THE
             transmission-time OF (CREATE transmission-entry))
          (ADD transmission-entry TO transmission-schedule)))

(WHENEVER (MAKE time BE THE clock-transmission-time
    OF subscriber)
      DO (MAKE time BE THE transmission-time
            OF (CREATE transmission-entry))
         (ADD transmission-entry TO transmission-schedule)
         (MAKE transmission-entry BE THE primary-entry
            OF transmission-schedule))
```

Figure 6

1. *Scope of conditional.* In natural language communication the end of a conditional is almost never explicit. Instead, context must be used to determine whether subsequent statements are part of the conditional. In sentence three of the example, the input to formula 2 is the SOL entry found in the previous sentence. Thus, sentence three is really part of the conditional statement.

2. *Implicit formation of relations.* In sentence two, the relative transmission time produced by formula 1 is supposed to be associated with the subscriber. Since that association is not established elsewhere, it is implicitly being established here. Hence this passive construct must be treated as an active one.

3. *Implicit creation of outputs.* In a similar fashion, various sentences establish associations with a transmission schedule (the output of this example) but an instance of one is never explicitly created. Such usage indicated that an implicit creation of the output is required.

A. Expectation failure. In addition to process and structural statements, a specification normally contains expectations about the state of the computation at some point which provide context for people to explain why something is being done or some properties of its result. They also provide some redundancy against which an understanding of the specification can be checked. In the example, one of these expectations (that all of the components of the entries of the output have been produced) fails, which indicates either a misunderstanding of the specification or an inconsistency Or incompleteness. In this case, both our example and the actual specification from which it was drawn are incomplete; they fail to describe how the length field of the entries of the transmission schedule are calculated from the inputs.

7. *DESCRIPTION OE THE PROTOTYPE SYSTEM*

The prototype system is structurally quite simple. It has three phases (Linguistic, Planning, and Meta-Evaluation) which are sequentially invoked to process the informal specification. Each phase uses the results of the previous phases, but no capability currently exists to reinvoke an earlier phase if a difficulty is encountered. Hence, either ambiguity must be resolved within a phase or the possibilities passed forward to the next phase for it to resolve.

We will describe the prototype system by working backward from the goal through the phases (in reverse order) toward the input to expose the system design and provide context for understanding the operation of each phase.

The goal of the system is to create a formal operational specification from the informal input, which means that it must complete each of the partial descriptions in the input to produce the output. In general, each partial description has several different possible completions, and a separate decision must be made for each partial description to select the proper completion for it.

Based on the partial description and the context in which it occurs, an a priori ordered set of possible completions is created for each partial description. But one decision cannot be made in isolation from the others; decisions must be consistent with one another and the resulting output specification must make sense as a whole. Since the output is a program in the formal specification language, it must meet all the criteria for program well-formedness. Fortunately, programs are highly constrained objects (one reason they are so hard to write), so there are many well-formedness criteria which must be satisfied.

This provides a classical backtracking situation [12], since there are many interrelated individual decisions that in combination can be either accepted or rejected by some criteria (the well-formedness rules). In such situations, the decisions are made one at a time in some order. After each decision the object (program) formed by the current set of decisions is tested to see if it meets the criteria (well-formedness rules). If it does, then the next decision is made, and so on, until all the decisions have been made and the result accepted. If at any stage the partially formed result is rejected, then the next possibility at the most recent decision point is chosen instead and a new result formed and tested as before. If all possibilities have been tried and rejected for the most recent decision point, then the state of the decision-making process is backed up to that existing at the previous decision point and a new possibility chosen. This process will terminate either by finding an acceptable solution (formal specification) or by determining that none can be found. The resulting object (program) is an acceptable solution (formal specification) f*or* the problem (informal specification).

The order in which decisions (rather than the order of alternatives within a decision) are made should be chosen to maximize early rejection of infeasible combinations of decisions. This requires that the rejection criteria can be applied to partially determined objects. The preferred decision order is clearly dependent on the nature of the acceptance/rejection criteria.

We now let the nature of the well-formedness criteria determine the structure of the prototype system so that the early rejection possibilities inherent in the criteria can be utilized. The criteria fall into three categories: dynamic state-of-computation criteria, global reference criteria, and static flow criteria. Each of these categories must be handled differently.

The dynamic state-of-computation criteria are based only on the current "state" of the program and its data base (e.g., "the constraints of a domain must not be violated" and "it must be possible to execute both branches of a condition"). They require that all decisions that affect the computation to that point (but not beyond) must be made before the criteria can be tested. Thus, if decisions could be made as they are needed by the computation of the program and the program "state" examined at each stage of the computation, then the dynamic state-of-computation criteria could be used to obtain early rejection of infeasible decisions.

This is exactly the strategy adopted in the design of the prototype system. However, since no actual input data is available for the program to be tested, and since the program must be well-formed for a variety inputs, symbolic inputs rather than actual inputs are used. Instead of actual execution, the program is symbolically executed on the inputs, which provides a much stronger test of well-formedness than would execution on any particular set of inputs.

However, completely representing the state of the computation as a program is symbolically executed is very difficult (e.g., determining the state after execution of a loop or a conditional statement) and more detailed than necessary for the well-formedness rules. Therefore, the prototype system uses a weaker form of interpretation, called Meta-Evaluation, which only partially determines the program's state as computation proceeds (e.g., loops are executed only once for

some "generic" element, and the effects of THEN and ELSE clauses *are* marked as POSSIBLE, but are not conditioned by the predicate of the If). This Meta-Evaluation process is much easier to implement and still provides a wealth of run-time context used by the acceptance/rejection criteria to determine program well-formedness.

The global referencing criteria (such as "parameters must be used in the body of a procedure") test the overall use of names within the program and thus cannot be tested until all decisions have been made. They are tested only after the Meta-Evaluation is complete.

The final category of criteria, static flow (e.g., "items must be produced before being consumed" and "outputs must be produced somewhere"), are more complex. The Meta-Evaluation process requires a program on which to operate, which may contain partial descriptions that the Meta-Evaluation process will attempt to complete by backtracking. This program "outline" is created from the informal input for the Meta-Evaluation process by the flow analysis, or Planning, phase, which examines the individual process descriptions and the elaborations, refinements, and modifications of them in the input, then determines which pieces belong together and how the refinements, elaborations, and modifications interact. It performs a producer/consumer analysis of these operations to determine their relative sequencing and where in the sequence any unused and unsequenced operations should occur. This analysis enables the Planning phase to determine the overall operation sequencing for the program outline from the partial sequencing information contained in the input. It uses the data flow well-formedness criteria and the heuristic that each described operation must be invoked somewhere in the resulting program (otherwise, why did the user bother to describe it?) to complete the partial sequence descriptions.

If the criteria are not sufficiently strong to produce a unique program outline, the ambiguity must be resolved either by interacting with the user or by including the alternatives in the program outline for the Meta-Evaluation phase to resolve as part of its decisionmaking process. In the prototype system, the Meta-Evaluation phase is prepared to deal with only minor sequencing alternatives such as the scope of conditional statements (If a statement following a conditional assumes a particular value of the predicate, it must be made part of one of the branches of the conditional.) and demons (Are all situations which match the firing pattern of a demon intended to invoke it or only those which arise in some particular context, and if so what context?). Major sequencing issues—such as whether one statement is a refinement of another or not—that cannot be resolved by the Planning phase must be resolved by the user before the Meta-Evaluation phase.

Both the Planning and Meta-Evaluation phases use a structural description of the application domain to provide context for their program execution, and inference rules which define relation inter-dependencies in the process domain. This structural base is the application-specific foundation upon which the Planning and Meta-Evaluation phases rest, and must be provided before they are invoked. It contains all the application-specific contextual knowledge. It augments the system's built-in knowledge of data flow and program well-formedness and enables the system to be specialized to a particular application and to use this expertise in conjunction with its built-in program formation knowledge to formalize the input specification.

The construction of a suitable application-specific structural base is itself an arduous, error-prone task. Furthermore, our study of actual program specifications indicated that most of the structural information was already informally contained in the program specification. We therefore decided to allow partial descriptions in the specification of the structural base and to permit such descriptions to be intermixed with the program specification.

Since we *are* concerned only with the semantic issues raised by using partial descriptions in the program specification, the system uses a parenthesized version of the natural language specification as its actual input to avoid any syntactic parsing issues. This parenthesized input does not affect the semantic issues we have discussed.

The first tasks, then, of the system are to separate the process descriptions from the structural descriptions, to convert both to internal form, and to complete any partial structural descriptions. These tasks comprise the system's Linguistic phase, which precedes the other two.

If a formal structural base already exists for some application, then, of course, it is loaded first and is augmented by and checked for consistency with any structural statements contained within the program specification.

Thus, in chronological order (rather than the reverse dependence order used above), the system's basic mode of operation consists of reading an input specification, separating it into structural and processing descriptions; completing the structural descriptions and integrating the result into any existing structural base; determining the gross program structure by producer/consumer analysis during the Planning phase; and, finally determining the final program structure through Meta-Evaluation.

REFERENCES

1. Balzer, Robert, Neil Goldman, and David Wile, "On the Transformational Implementation Approach to Programming," *2nd International Conference On Software Engineering,* October 1976, p. 337.

2. Bell, Thomas E. and David Bixler, "An Extendable Approach to Computer-Aided Software Requirements Engineering," *2nd International Conference on Software Engineering,* October 1976, p. 70.

3. Teichroew, Daniel and Ernest Allen Hershey, III, "PSL/PSA A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *2nd International Conference on Software Engineering,* October 1976, p. 2.

4. Jones, Anita K. and Barbara H. Liskov, "A Language for Controlling Access to Shared Data," *SEC Supplement,* 1976, p. 68.

5. Wulf, Wm. A. and Mary Shaw, "An Introduction to the Construction and Verification of Alphard Programs," *2nd International Conference On Software Engineering,* October 1976, p. 390.

6. Lampson, B. W. and J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek, "Report on the Programming Language Euclid," Xerox Research Center, Palo Alto, August 1976.

7. Snowdon, R. A. , *PEARL: An interactive system for the preparation and validation of structured programs.* Computing Laboratory, University of Newcastle Upon Tyne, November 1971.

8. Schwartz, J. T., *On Programming, An Interim Report on the SETL Project,* Computer Science Department, Courant Inst. Math. Sci., New York University, 1973.

9. Guttag, John V., Ellis Horowitz, and David R. Musser, "The Design of Data Type Specifications," *2nd International Conference On Software Engineering,* October 1976, p. 414

10. CODASYL, Data Base Task Group, April 1971 Report, ACM, New York.

11. Elschlager, R., "Overview of a Natural Language Programming System," Unpublished Report, CS Department, Stanford University, February, 1977.

12. Gerhart, Susan L. and Lawrence Yelowitz, "Control Structure Abstractions of the Backtracking Programming Technique," *2nd International Conference On Softu/ar* Engineering,* October 1976, p. 391.