AUTOMATED DERIVATION OF PROGRAM CONTROL STRUCTURE FROM NATURAL LANGUAGE PROGRAM DESCRIPTIONS

David Wile
Robert Balzer
Neil Goldman

USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, Ca. 90291

Abstract

This paper describes a system which organizes a natural language description of a program into a conventional program control structure, as a part of a larger system for converting informal natural language program specifications into running programs. Analysis of the input program fragments using a model of a human "reader" of specifications has been found to be a very successful adjunct to conventional "planning" methodologies.

Natural language descriptions of programs can frequently be characterized as "rubble"--a very loosely organized set of almost independent description fragments [Schwartz]. Such specifications are often quite robust, due to a large degree of redundancy; they are also frequently quite concise, due to reliance on the readers' innate knowledge and their knowledge of the application domain. This paper discusses a paradigm for structuring the portion of "rubble" program descriptions which maps into conventional programming language control constructs and definition facilities.

In order to focus on structuring natural language, it is necessary to indicate where this mapping fits in the broader scheme of "understanding" natural language program descriptions. The research described below is the basis for the design of an intermediate stage in the operation of the SAFE system [Balzer], a system designed and implemented at ISI to produce formal, operational specifications for programs described in natural language. In particular, a (parenthesized) natural language description of a program is given to the system--a description which retains most semantic ambiguities of natural language, but which avoids its syntactic ambiguities. The input first goes through a "domain acquisition" phase [Goldman], which acquires domain knowledge relating the objects and actions of the modelled world. The "planning phase", described herein, is then used to structure the input into a program in conventional terms. Finally, a phase concerned with the resolution of fine details--anaphoric reference, type conversion, and some sequential structure resolution--is used to produce the final program. The respective phases deal in turn with the data and operation structure, the program definition and control structure, and the program variable and parameter structure.

The SAFE system makes operational specifications more precise by filling in those details that were surpressed from the specification because they were deemed inferable by an "intelligent reader". These specifications must be operational, specifying informally and at a high level, how something is to be done, not merely what must be achieved. This requirement enables the corresponding formal program to be constructed without any deep problem solving activity by resolving the ambiguities contained in the specification within the context of program well-formedness rules and the constraints of the application domain. When an ambiguity cannot be resolved by the system, it asks the user which interpretation is intended.

An Example

There appear to be three basic problems when attempting to map natural language, operational descriptions of programs into program *control* constructs: the mapping is generally one-to-many (or even many-to-many); considerable reliance on implicit relationships between application domain primitives is used to disambiguate sequential relationships between events; and all descriptions are subject to contextual refinement and interpretation, a facility almost completely foreign to existing programming languages. To illustrate the problems that arise in converting natural language input into a program control structure, a small example is presented below. The example has application domain nouns and verbs replaced with

This research was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. DAHC15 72 C 0308.

upper case letters so that the reader is not able to use domain specific knowledge unavailable to the system:

> Normally, each *P* must be *X*ed before being *Y*ed. Each *P* must also be *Z*ed. However, *W*ed *P*s need not be *X*ed. It is necessary to *X* each of *P*'s *Q*s.

There are several ambiguities in this example: there are ambiguities of action sequencing--are there any sequential relationships among *X*, *Y*, and *Z* other than the explicit one stated? The mapping of some sentences into program constructs is also ambiguous; e.g., is the first sentence a loop, a conditional (if a *P* is in context) or even a "demon" (if *P*s are created asynchronously)? Notice the contextual modification that the "however" must cause: previous sentences' program images must now be embedded in a conditional which tests for the special case (*W*ed *P*s). Is the last sentence a refinement of the use of *X* in the first sentence, or does it indicate a separate invocation of *X*? Clearly, both the reader and our system would require more information to resolve these ambiguities. All of the decisions needed to determine a correct control structure for the above program rely heavily on relationships among the primitive actions and object types and the context in which the input is interpreted.

Summary of Basic Method

Considerable effort in artificial intelligence has been spent attempting to automatically determine sequential plans to accomplish some goal [Sacerdoti, Fikes]. We have very little to add to that technology. Our approach differs in that we are not concerned with creation of sophisticated plans which entail deep reasoning processes: the user is responsible for formulating the plan--our system "merely" makes it formal. It is fortunate that we can make this assumption, for the sequential ambiguities interact strongly with the contextual and statement form ambiguities: the combination is quite complex.

Our contribution lies in the resolution of the contextual ambiguities--the discovery and incorporation of refinements of, and modifications to, structures produced from the text. We know of no other efforts which attack this problem in program understanding systems, but see analogs in general natural language understanding [Riesbeck] and dialog understanding [Mann] systems. Our general approach is to first postulate a program structure based solely on static evidence. A second pass over the resulting program is used to further resolve--in a more dynamic way--problems left unsolved in the first pass.

The static analysis of the program weakly models a human reader of program specifications. Such a reader tends to *predict* what the next sentence will do to the context he has already obtained. For example, if a particular verb is not used in a manner consistent with the reader's understanding of its usage, (perhaps *X* cannot be used on a *P* in the example above) he may predict that more will be said to *refine* his understanding of the verb. Some linguistic cues explicitly force prediction, such as the use of "normally" above. If a reader has some perception of a goal of a process, and that goal has not been met, he may predict that further sentences will be at the same level as the last--will be successors to it in a control construct sense. In the example above, if *Y*ing did not produce the goal of the process, a successor to *Y* would be predicted. Furthermore, if a perceived goal has been met, the reader may predict that the next sentence will *change contexts*--the current context has been specified fully.

Naturally, the reader does not rely entirely on prediction for his structural understanding, but is quite willing to *adapt* when explicit or implicit cues contradict his predictions. In such cases, he may remember the prediction (or more accurately, the problem which lead to the prediction), but will certainly accept the next sentence for what it is. For example, the reader faced with a new paragraph when the goals of the previous paragraph were not met, will probably change contexts in accord with the first sentence of the new paragraph. In the above example, if *Z* was the sole producer of an input required by *Y*, the reader would adapt and place *Z* as the predecessor of *Y*, despite a prediction that the sentence will be a successor for *Y*.

In order to model human reading skills (at this very shallow level), the planning phase considers each sentence in order, and attempts to predict what the next sentence will do to the program being constructed (its "context"). Then, when considering the next sentence, it examines it for surface and derived features, and, when possible, incorporates it into the program in a manner consistent with one of its predictions. If it cannot incorporate it, it either adapts by acting according to explicit directive information in the sentence (as arises with the "however" in the example above), or it creates a new context in which it interprets subsequent sentences, until the new context's relationship with the previous context can be established.

The program resulting from the application of the reading model is then simulated by the planning phase in order to see if problems discovered in the static model will be resolved dynamically. Such resolution might occur when some calling context produces relationships known to be required by the called program, but not known to have been produced during the reading model analysis of the called program. Some minor modifications to the sequential structure of the program can be made during this phase, but the definition (refinement) structure is considered to be constant at this point. The dynamic modelling mechanism is based on a quite traditional (STRIPS-like) "meta-evaluation" technique.

Central to the paradigm is the information used to relate the primitive objects, actions, and events of the domain. The basic methods for discovering problems and isolating contexts (for modifications and refinements) rely on an abstract representation of the program in terms of sets of relation and type names used by the events. The sets associated with events represent the relations and types consumed by the event, those both produced and consumed by the event, and those just produced by the event. Producer/consumer relationships between events are expressible as set operations (membership, inclusion, intersection) on these. Although some ad hoc measures are required for second order events (events with events as parameters) and some detailed refinement considerations are required, the abstract representation yields a rather clean mechanism for analysis of problems in "rubble" representations.

The SAFE system is an operational system which has been applied to several moderate sized examples (around ten English sentences). Below we outline the environment in which the planning phase runs, explicate the basis it uses for collecting information, and then explain the context model mentioned above in more detail, demonstrating it with an example.

Environment of Planning Phase

The planning phase processes programs after linguistic analysis has determined the data and procedure specification structure and before a "meta-evaluation" process determines anaphoric referents, type conversions, defaults and other details relevant to formalizing the specification as an operational program. It is necessary to understand both the input/output format of the planning phase and the model world in which final programs are run to understand the actions of the planning phase.

The SAFE system attempts to convert parenthesized natural language specifications into a precise program in a high level programming language whose major distinction is that it supports a relational data base with backtrackable pattern matching, automatic inference, constraint and demon invocation features. Output programs are in an event-oriented representation in which actions manipulate objects and their relationships with one another by modifying the contents of the global relational data base. An *event* is an invocation of an action--a procedure call in traditional programming languages. It is the sequencing of events and the definitions of actions as sequences of events which is the prime responsibility of the planning phase.

The input to the planning phase is a fleshed out translation of the natural language input into a form we call *event descriptors*. These descriptors correspond approximately with instantiated *case-frames* for basic actions and relations known to the system either as primitives or domain-specific actions. In the process of building the event descriptors for the input sentences, the linguistic phase builds up a model of the relations, types and actions which will be used by the program being specified. (This model may be augmented by an explicit model given to the system preceding linguistic analysis.) The model contains a strongly-typed specification of each relation and action's parameters. Additionally, some inference rules may have been established for the domain.

The basic event descriptor types into which the input sentences are mapped are: *events* (from verbs), *objects* and *sets* (from nouns and plurals--the arguments to events), *conditionals*, *conjunctions*, *loops* (from verbs with set objects) and *sequences* (from explicit enumerations of steps). The planning phase manipulates the paragraph/sentence/clause structure of the input into a program structure, represented in terms of the same types of event descriptors. A few additional types are needed to represent the programs output by the planning phase: *parallel event* descriptor, *choice of events* descriptor (ambiguous events to be decided by the final phase), and *selector of events* descriptor. With the exception of choice of event descriptors, each descriptor has a natural analog in the programming language into which the final phase ultimately translates the program specification.

Abstract Input Representation

The planning phase has three tasks: to choose programming language control constructs to represent English language syntactic constructs, to determine the sequence in which events must be called, and to determine the definition and calling structure of the input events. In this section, we describe the evidence used by the planning phase to make the decisions required to accomplish these tasks.

A significant portion of the task of choosing control constructs is presently done by the linguistic phase--as is indicated by the event descriptor types which are passed to the planning phase. However, we hope to lessen this reliance on the linguistic phase in the future and consider delaying more control construct choices. Because the linguistic phase never considers intersentential relationships, except to keep the domain model consistent across sentences, all distributed control constructs are missed by the linguistic phase.

These are handled by the planning phase in the "reading model" phase sketched above. These all tend to be canonical mappings based on English usage patterns, often involving explicit syntax indicating cases or exceptions--such as, "normally", "otherwise", "an exception is...", etc. In addition, some second order actions are indicative of distributed control construct usage. Some adverbs indicate explicit demonic application is desired--"whenever x occurs do y".

Although control construct choice is based primarily on syntactic cues, sequential evidence is about equally implicit and explicit. Explicit sequential relationships between events are indicated via conjunctions--e.g. "...and then...", adverbs--e.g. "first", "next", and by explicit statements of expectation or purpose--e.g. "...the purpose of x is to determine y".

As implicit evidence for sequential relationships between event descriptors, a producer/consumer analysis is done by the planning phase. Because the events are incomplete (e.g., missing parameters, have incorrect parameter types, have no variable names bound), the analysis is much more abstract than that done in the precise mathematical proof systems used for goal directed planning schemes like STRIPS.

The method that is used is to define precondition and postcondition abstractions for all primitve events (calls on actions). These conditions are represented as sets of names of relations and types used as parameters to events along with the types and relation names defined for the preconditions to the primitive actions which they invoke.

For example, the event representing "catch the green ball from the red player" would have as abstract precondition the set { color ball player }, plus whatever precondition the action associated with the verb "catch" might require (which would certainly contain some relation like "thrown", "tossed", "in motion", etc.). Notice that color in the above is an implicit relation which was determined by the linguistic phase as the way of relating green with ball and red with player. The postcondition of the same event depends entirely on the postcondition of catch, which would at least include a type such as "location."

Precondition and postcondition abstractions are extended to control constructs containing primitive event descriptors in the obvious way. For example, the precondition for a conditional event consists of the union of the names of relations and types used in the condition with the precondition abstractions of the "then" and "else" parts. The postcondition abstraction is the union of the postcondition abstractions of the "then" and "else" parts. After extending the abstract precondition and postcondition set concept to all event descriptor types, we can make the definition: event *A* ***implicitly precedes*** event *B* if the abstract precondition of event *B* intersected with the abstract postcondition of event *A* is not empty.

Finally, we must consider what evidence may be used to indicate that one event was intended to define or "refine" another. Naturally, we must look to English usage for the clues. Since we know of no previous research attempting to deduce the refinement structure of program descriptions we have had to develop our own simple theory of refinement clues. There are two very common devices used in English to indicate that a particular event actually references another event--i.e., requires refinement. First, there are a large number of "second order" constructs (which take events as parameters), which explicitly require refinement. Verbs such as process, determine, compute, treat and establish become explicit *event variables*; each is handled individually by the planning phase, although a general pre and postcondition analysis can be used for purposes of determining refinements. In addition, some prepositions (e.g., before, after and by) also occasion the creation of event variables.

Other devices used to indicate that an event is indeed an event variable are omission of parameters to an action and type mismatching in which a set is used in place of an element type. These are implicit methods for expressing the need for refinement. It often (usually) turns out that events are "self refining"--the user intended that the system fill in the missing details rather than match a more precisely specified event elsewhere. However, the facility is a very powerful Enlish facility when used as an event reference, and one which deserves special attention in program description understanding. It is absolutely necessary to recognize refinements when a specification is presented to the system in the most ideal way: as a stepwise refinement of previous events.

Recognition that an event is an event variable is the easier half of the refinement problem. Discovering which event refines an event variable--i.e., the event variable's binding--is more difficult. The candidates are drawn from those events with consistent producer/consumer abstractions (for second order verbs) and from those which call the same action (for domain action refinements). The latter analysis is a bit ad hoc--parameters are counted and corresponding parameters of possible refinement-related events are checked for upward type compatability.

To summarize, the information used by the planning phase to determine the sequencing of events, definitions of domain actions and refinements of events, includes both explicit and implicit notions of sequence and event invocation. Implicit sequencing is determined from an abstract representation of the preconditions and postconditions to events and actions. Implicit refinement requirements come about from a subsetting analysis of parameters to events.

The Reading Model

The initial version of the planning phase--completed in October, 1975--treated the problems of action definition (aggregation, refinement) and event sequencing as separable. Candidate refinements were determined, definitions made, and then a STRIPS-like simulation (in the abstract space of sets of relation and type names) was used to incorporate the remaining uncalled events. This worked surprisingly well considering that the order of sentence input was absolutely ignored and that decisions were only made when a

unique producer/consumer relationship could be established. A moderate size message-processing example was successfully run through this system [Balzer].

Further examples quickly established the weaknesses of this approach. It should be mentioned that although we do not eschew user interaction, we would prefer that it be minimized and reserved for rather difficult and important questions. Hence, although almost any scheme that narrows down sequential choices is amenable to a "menu" interaction mechanism, it is now quite clear to us that program understanding systems need some notion of "context" to exhibit intelligence.

Intuitively, we want a context to correspond to a "state of mind" of a reader of a program specification. This translates most naturally into a set of event descriptors. Once it is established that a particular *context*--set of event descriptors--is relevant, all efforts will be directed toward relating successive event descriptors (from the input sequence) to the event descriptors in the given context. The major benefit this has is to make some abstract relationships unique which would otherwise be ambiguous due to possible relationships with events in other contexts. For example, an event which implicitly precedes two events, only one of which is in context, may be put into the program as the predecessor of the event in context. Naturally, since these relationships are used to determine an event's position in the program structure--both sequentially and definitionally--uniqueness avoids user interaction.

We emphasize: a context is a set of event descriptors. When it is established that a new event descriptor should be related to a given context, even the most obscure relationship with event descriptors in that context will be tried before a different context's events are considered.

The number of relationships between event descriptors in any context may be quite large. Also, the events are changed by the planning phase dynamically--precedence and refinement relationships have to be recomputed frequently--for they are often recursive functions of the extant program structure. Hence, it is somewhat important to minimize the number of relations considered when attempting to incorporate a new event into the program. This is done by *predicting* how a successor sentence will be used. This limits the number of relationships with the current context which are necessary to consider. In addition, a predictive mechanism is necessary for some intersentential linguistic (syntactic) forms. The predictions present an interesting dichotomy to the contexts: while contexts limit the *events* considered in relationships with successive input events, predictions limit the *relationships* considered with them.

It is perhaps best now to describe the reading model in more detail, followed quickly with an example of how it works. The overall model is to attempt to incorporate each clause of each sentence of each paragraph (considered in input order) into the program. The initial state is set up to have several (empty) contexts predefined--one for each action to be defined, one for the main program and one for "problems" which arise. The default context is the main program.

With each clause we first establish a context from explicit cues in the sentence, if necessary. These could be occasioned by English statements of the form "during *X* do *Y*" or more obliquely "after *Zing*"--where *Zing* is done only in one particular context. Given a context, we attempt to incorporate the clause into the context and then predict what the next clause will do to the context. To summarize, the top level reading model does the following:

```
Initialize context to main program;
Each paragraph:
   Each sentence:
      Each clause:
         Establish context;
         Incorporate clause into context;
         Predict next clause's effect.
Change context back to main program.
```

As mentioned above, the context is established by considering information explicitly included in the event. In order to incorporate the current clause into the program, the planning phase first determines those predictions which can be satisfied by the current clause in the current context. If those predictions are consistent, actions are then taken based on the predictions. (A set of predictions can simultaneously be satisfied--an action may refine a superior action, and at the same time establish the enabling conditions for a second action--thus preceding it.) If the satisfiable predictions are inconsistent, then not enough information is known to unambiguously determine how to use the event, so a *problem context* is established with the clause as its sole member. (An event which could both precede and follow another event in a context would give rise to inconsistent predictions both being satisfiable.)

If no predictions are satisfied, the clause is examined for explicit directive information to see if some coordinating conjunction was used (e.g., and-then, finally, first, etc.). Lacking these, a "last ditch" attempt is made to see if the new event could possibly parallel some other event in context (consuming and producing approximately the same information). Then (finally) the reading model considers changing back to a previous context to attempt to incorporate the event.

We can summarize the incorporation of the current clause, thus:

```
Determine satisfiable predictions;
If consistent,
      take actions indicated by predictions;
If inconsistent, establish a "problem" context;
If no satisfiable predictions:
   Attempt to use explicit information;
   Attempt to use a parallel construct;
   All else failed: Change contexts.
```

Predictions for the next clause are made based on the effects of the current clause on the context. All newly incorporated clauses are subject to abstract pre and post condition analysis to decide whether to predict that the successive clause will *precede* or *follow* the current clause. Additionally, all event descriptors which are calls on actions are subject to possible *refinement* predictions (based on parameter information as mentioned above).

Conditionals and some second order actions can cause predictions that the next clause will specify *another subcase* of the current conditional, or that it may *subsume* the current clause, by providing a definition for an exceptional condition (the case when "normally" is used).

The actions taken based on satisfied predictions are basically trivial--the event is incorporated as the successor, predecessor, refinement, or subcase in a straight-forward manner. The only interesting situations left to describe involve the consistency of predictions and what goes on when changing contexts--specifically, how do problems ever become resolved, contexts popped, etc.

Consistency of predictions is a difficult problem. Fortunately, the assumption that predictions are inconsistent has the effect of forcing the user to decide which predictions are satisfied (in most cases) because the planning phase changes to a "problem context". An arbitrary design decision has been made to permit only one such problem context to exist at any given time. Hence, if a second problem arises when already in a problem context, the system will be forced to resolve the problem, usually by calling the user.

The main decision in the planning phase is whether or not to "change contexts." It is a subsequent and separate decision whether to push a new context, pop to an old, etc., once a change has been initiated. This decision procedure is quite simple. Contexts are organized into a set, some of whose elements are ordered into a stack. If a problem context exists, it is the top of the context stack, and once the problem is resolved, will most likely be incorporated into the context which is the second element of the stack. If we change contexts to a context which is already on the stack, the stack is popped to that context. Otherwise, the new context becomes the top of the stack. However, a problem context is always resolved before any activity is permitted on the context stack.

The resolution of the problem is conceptually quite neat--the context itself is treated as a clause and the normal routine for incorporating the current clause is called recursively. If it can be incorporated, it is. Otherwise, the (ambiguous) set of satisfiable predictions is presented to the user who decides how the clause should be incorporated. It must be mentioned that while a problem context exists, it acts like any other context: successive sentences are incorporated into it. This incorporation may cause the original problem to go away and allow the mechanism just described to resolve the problem.

An Example

To illustrate the utility of the reading model we will sketch the path of a small example through the program. The example is a modification of a larger example on which the program has run successfully:

> Screen each message before it is output. To screen a message, match each of the message IDs with the guard list. During screening, initially validate the message. If it is valid, determine the guard list by prompting the operator.
>
> After screening, if a match occurs, send the message with the matching ID as key. Otherwise,...

A very stylized version of the event descriptors output by the linguistic phase for the above is:

1. *Each* message:
   screen(message);
   output(message).
2. screen(message) =
   Each ID (message): match (ID, guard-list).
3. *during* screen: *initially*: validate(message).
4. *if* valid(something):
   determine (guard-list, prompt(operator))
5. *after* screen:
   if match(something):
   send(message, match(ID, something))
6. *otherwise*: ...

Assuming now that validate, output, match, prompt and send are all primitive--and that prompt and send are output routines, the reading model would procede as follows.

The first clause will become the main program (default starting point) since it does not explicitly change the context. Predictions will be set up to require a producer of a set of messages (something must precede the main program) and a refinement of the output event--since a device parameter is missing.

The second clause will then be read; as a defining clause of screen, the context consisting of the main program will be pushed, and the loop will become the body of screen. (Immediate actions--like definition-- are actually taken care of before the attempt to incorporate current clause is made.) No producer of a guard-list is known. Let's assume there is an inference rule relating messages to IDs, so the only prediction of a predecessor will be that something will produce a guard-list; i.e., a predecessor to the match will be predicted. In addition, nothing consumes the match result, so a successor to the body of screen will be predicted also.

The third clause will establish that the context should indeed be screen. (It is actually ambiguous after sentence 2 which context is predominant--the main program or screen. We have no good ideas on how to adapt the context mechanism's dominance, yet.) At any rate, if we assume that the validate produces no useful result for match--viz., no guard-list--we have no reason to put it in front of the body of screen from a producer/consumer analysis standpoint. In addition, it is ambiguous whether the validate should precede the loop or the match within the loop. This ambiguity must be discovered by the explicit routine that attempts to incorporate "initially" clauses. (Very similar routines must exist for the other relative conjunctions when they are not incorporated via predictions.) Hence, a problem context is created. The context stack consists of: problem (sentence 3) at the top, screen definition second, and the main program last. Since no one consumes the validate results, the model predicts a successor to validate will be forthcoming. (In this problem context, no one produces a message, so a predecessor will be predicted as well.)

The fourth sentence then comes in, succeeds the validating, thus fulfilling a prediction in the current context, and will be incorporated into the current context. No one consumes the guard-list, so a successor to sentence 4 will be predicted. In addition, no action was specified to be taken when validation failed. Hence, a prediction that this will be specified subsequently is made.

Sentence 5 first attempts to change contexts into one in which screening has occurred--the main program. This forces the problem context to be popped: hence, the problem must be solved. However, the "initially" now makes sense, for the entire problem context now produces a guard-list which can precede the match. Unfortunately, it remains ambiguous as to whether the guard list is determined each time through the loop or only once. (Had the second clause used an ID, for example, this ambiguity would have been resolved.) The user must be prompted to make this determination (whose outcome is irrelevant to this discussion).

Now sentence 5 has reestablished the main program as its context. Because send can be an output routine, either send or the whole conditional can be a refinement of output. Choosing the conditional satisfies the prediction of refinement of output and incorporates the whole clause. The conditional then causes the prediction of extra cases to be made.

Hence, the otherwise in sentence 6 will be incorporated as the case when no match occurs...

It should be clear how the reading model is intended to work at this point. We mentioned earlier that a dynamic model is used after the reading model to resolve problems which depend on a more dynamic context. It can be used to decide that problems predicted by the reading model are not actually problems, by showing that some relations are consumed dynamically. For example, the match produced by screen looks like a problem to the reading model, for no one consumes it inside of screen. However, the dynamic model will be quite happy, for match is consumed by the main program. In addition, the dynamic model can insert some trivial events which it invents when relations are consumed for which there is only one possible producer. What once was the most important part of the planning phase--the dynamic model--is now basically a consistency checker. More experience is needed with the reading model to determine where its presumptiveness requires delaying some decisions to this last dynamic phase.

Conclusions

It is clear to us that natural language program specifications can be automatically understood by machines--given the ability to interact with the specifier. It is now a matter of how large a vocabulary and how sophisticated the reasoning processes need to be to constitute a useful facility. Although it is conceivable that program specifications can be structured from "rubble" without a "reading model", it is almost certain that some of the natural language flexibility of event reference and specification imprecision must be lost if the input organization is ignored.

We are quite pleased that this phase has serendipitously attained an unexpected measure of robustness. Recall that we were forced to abstract our characterization of events to a point far less precise than might be considered desirable--sets of relations and types were used instead of precise predicates with variables, quantifiers, etc. In so doing, the number of relationships between events became too large--more events seemed to be related than were in reality. The reading model has exactly the right property of narrowing down the number of events considered so that the number of relationships between events is actually brought in line with what might be expected from a less abstract model in the first place! The competition and integration of these alternative information sources provides a balanced approach not inherent in either. This ultimately gives the planning phase a robustness unusual in complex systems.

For the immediate future, we intend to run several small examples through the system, probably doing some perturbation analysis on paraphrases of the examples. Within the next few years we intend to attack a much larger example (20 pages) and face the sizing issues necessary for the development of a system of any real utility to specifiers.

REFERENCES

1. Balzer, R. M., Goldman, N. M., Wile, D. S. Informality in program specifications. USC/Information Sciences Institute. (April, 1977) ISI/RR-77-59.

2. Fikes, R. and Nilsson, N. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence.* 2 (1971).

3. Goldman, N. M., Balzer, R. M., Wile, D. S. The inference of domain structure from informal process descriptions. in Proceedings of a Workshop on Pattern-Directed Inference Systems, May 23-27, Honolulu.

4. Levin, J. A. and Moore, J. A. Dialogue games: meta-communication structures for natural language interaction. USC/ Information Sciences Institute (January, 1977) ISI/RR-77-53.

5. Riesbeck, C. and Schank, R. Comprehension by computer: analysis of sentences in context. Yale Univ., Dept. of Comp. Sci., Research Report 78, Oct. 1976.

6. Sacerdoti, E. The nonlinear nature of plans. Stanford Research Institute, AI Group Technical Note 101, Jan. 1975.

7. Schwartz, J. Structureless programming. Courant Institute, SETL Newsletter 135A, Jul. 1974.