An Equipment Model and its Role in the Interpretation of Nominal Compounds

Tomasz Ksiezyk and Ralph Grishman

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer St.
New York, New York 10012
(212) 460-7446, (212) 460-7492

Abstract

For natural language understanding systems designed for domains including relatively complex equipment, it is not sufficient to use general knowledge about this equipment. We show problems which can be solved only if the system has access to a detailed equipment model. We discuss features of such models, in particular, their ability to simulate the equipment's behavior. As an illustration, we describe a simulation model for an air compressor. Finally, we demonstrate how to find referents in this model for nominal compounds.

1. Introduction

The work presented here is part of the PROTEUS (PROtotype TExt Understanding System) system currently under development at the Courant Institute of Mathematical Sciences, New York University.[1] The objective of our research is to understand short natural language texts about equipment. Our texts at present are CASualty REPorts (CASREPs) which describe failures of equipment installed on Navy ships. Our initial domain is the starting air system for propulsion gas turbines. A typical CASREP consists of several sentences, for example:

> Unable to maintain lube oil pressure to SAC [*Starting Air Compressor*]. Disengaged immediately after alarm. Metal particles in oil sample and strainer.

It is widely accepted among researchers that in order to achieve natural language understanding systems robust enough for practical application, it is necessary to provide them with a lot of common-sense and domain-specific knowledge. However, so far, there is no consensus as to what is the best way of choosing, organizing and using such knowledge.

The novelty of the approach presented here is that, besides general knowledge about equipment, we also use a quite extensive simulation model for the specific piece of equipment which the texts deal with. We found that for understanding purposes it is more appropriate to make the simulation qualitative rather than quantative. Thus, for example, we are not interested in the precise value of oil pressure, but only whether it is too low or too high. The model is built from instances of prototypes which contain the bulk of general knowledge. It exists in the system permanently. In this situation the analysis of a piece of text consists of two stages: (1) locating in the model the objects mentioned in text; (2) interpreting the text using both the specific information residing in the model and the general knowledge which is accessible from the model. There is no clear-cut distinction between these two stages (see discussion of the examples in the next section).

[1] An overview of the system is given in [Grishman 1986], submitted to the AAAI-86.

We see the following merits of having a simulation model:

(a) the model provides us with a reliable background against which we can check the correctness of the understanding process on several levels: finding referents of noun phrases, assigning semantic cases to verbs, establishing causal relationships between individual sentences of the text.

(b) the requirements of simulation help us to decide what kind of knowledge about the equipment should be included in the model, how it could best be organized and which inferences it should be possible to make. It appears that the information needed for simulation largely coincides with that necessary for language understanding.

(c) the ability to simulate the behavior of a piece of equipment provides a very nice verification method for the understanding process at the level of interaction with a user - it is relatively straightforward to build a dynamic graphical interface which allows the user to have a friendly insight in the way his input has been understoood by the system.

In the remainder of the paper we will show examples of problems which can be solved only if the system has access to some kind of a simulation model of the domain equipment. Having demonstrated the need for such a model, we will discuss the design decisions which we found important for our domain and which seem to apply generally for complex equipment. How these considerations influenced the model for the SAC may be seen in the next section. Then we present a method of finding referents in the model for nominal compounds describing SAC's components. Finally, we briefly describe our future work.

2. Need for a Model

In most natural language understanding systems the knowledge about the domain of discourse is organized in the form of prototypes for objects and actions, and for the relations between them which are relevant for the domain. The prototypes are repositories for knowledge about the instances they subsume. This knowledge is highly structured - there are many links through which apparently distant concepts may be connected. The text is processed on a sentence by sentence basis. Usually, each sentence is split into linguistic entities with syntactic and semantic information attached. This information is used to determine the prototype for each entity. Through these prototypes there is access to general information about the concepts invoked by the sentence. This information is often necessary for the adequate interpretation (i.e. understanding) of the sentence. To account for the fact that the understanding of an utterance depends sometimes on the context in which the utterance is set, it is necessary to maintain information about the discourse context. One way of organizing this information is by creating and storing instances of prototypes for entities from the text as they come under analysis. The combined information coming from the context and from the processed sentence is used to solve problems like anaphora resolution, connectivity, etc.

Assuming this approach, let's consider the following sentence (let it be the first sentence in the analyzed text):

Starting air regulating valve failed.

Having completed the syntactic and semantic analysis of the sentence, we would recognize *starting air regulating valve* as an example of the prototype *regulating valve*. We would then fetch its description and create an instance of a regulating valve. Next, using the general knowledge about valves (of which regulating valve is a more specific case), and the semantic information about *starting air*, we would modify the just created instance with the fact that the substance the valve regulates is starting air. From the syntactic analysis we would know that *starting air regulating valve* is the subject of verb *fail*. Using the prototype of the action *fail*, we would create its instance and possibly also would further modify the instance of the valve so that the fact about its operational state is recorded. These two instances would now constitute the discourse context so far. Now, suppose the message continues with the sentence:

Unable to consistently start nr 1b turbine.

The processing would be similar to what has been described above for the first sentence. We would create an instance of a gas turbine, would fill its proper name slot with *nr 1b* and finally use the instance as an argument in another instance recording the finding about start problems.

These two sentences come from an actual CASREP. In the starting air system (our initial domain) there are three different valves regulating starting air. Two questions might be posed in connection with this short, two-sentence text: (1) which of the three valves was meant in the first sentence? (2) could the failure of the valve mentioned in the first sentence be the cause of the trouble reported in the second sentence?

The general knowledge of equipment may tell us a lot about failures, such as: if a machinery element fails, then it is inoperative, or if an element is inoperative, then the element of which it is part is probably inoperative as well, etc. Unfortunately, such knowledge is not enough: there is no way to answer these two questions (not only for an artificial understanding system, but even for us, humans) without access to rather detailed knowledge about how various elements of the given piece of equipment are interconnected and how they work as an ensemble. In our case we could hypothesize (using general knowledge about text structures) that there is a causal relationship between the facts stated in the two sentences. To test this, we would have to consider each of the three valves in turn and check how its inoperative state could affect the starting of the specific (i.e. nr 1b) turbine. If one of the three valves, when inoperative, would make the turbine starting unreliable, then we could claim that this valve is the proper referent for the *starting air regulating valve* mentioned in the first sentence. This finding would let us also answer question (2) affirmatively.

The above example, as well as others of similar nature, demonstrate that in cases where the domain is very specialized and complicated (a typical situation for real-life equipment), language understanding systems should be provided not only with general knowledge about the equipment but also have access to its model.

With an equipment model available, the processing of the two sentences would change: for the first sentence, instead of building a new description for the *starting air regulating valve*, we would rather try to find an object / objects in the model which could be described by this noun phrase. We would treat *nr 1b gas turbine* similarly. The semantics of *start* would be a kind of simulation procedure defined for the model. Now, let's consider problems (1) and (2) again. Viewing *nr 1b* as a proper name, we should easily find the object in the model which corresponds to the referred turbine. The analysis of *starting air regulating valve* would leave us with three pointers to the three objects in the model corresponding to the three starting air regulating valves in the equipment. In order to resolve this ambiguity we could make the following assumption, which seems very reasonable:

> Suppose first, that the valve's failure has indeed caused problems for the turbine. Now, if we confirm that at least one among the three valves, if inoperative, has this effect, then our assumption was correct and we found the right referent(s); if none of the three valves has any impact on the turbine, then our assumption was wrong: it answers question (2) negatively and leaves (1) still open.

Then we would proceed with the confirmation phase, considering each of the three candidates separately. We would temporarily set its operational state to INOPERATIVE, initiate the START procedure, and then check whether the functional state of the *nr 1b gas turbine* in the model has been set to RUNNING (for simplicity reasons let's assume that there is no *consistently* adverb in the second sentence). If for all three simulation experiments we wind up with the value RUNNING for the turbine, then we must conclude that there is no causal relationship between the sentences. Otherwise, we would claim to have found the right referent for the valve. Having unambiguously located the object referred to in the first sentence, we would modify its operational state accordingly.

3. Characteristics of an Equipment Model

In the preceding section we tried to show that general knowledge about equipment is by itself not enough to solve some problems of understanding. The decision to provide PROTEUS with an equipment model confronted us with a new question. Where and how to draw a division line between the knowledge about equipment in general and a model of a specific piece of equipment? The ultimate objective of our research is to design PROTEUS in such a way that it may be adapted easily to new equipment. Clearly, the model has to be built anew each time we want to use PROTEUS for a new piece of equipment. The general knowledge, on the other hand, should undergo, in such cases, only a slight extension due to the new types of components in the new equipment. For example, moving from the starting air system to the main reduction gear, we would have to build a new model for the gear, but while doing this, we should be able to use many of the structures designed for modelling components which also occured in the starting air system, like bearings, lubrication system elements, etc. This goal can be achieved using prototypes and their instances: the model would be built of instances of prototypes. The prototypes would constitute part of the general knowledge data base. In the instances we would store only the information which is specific to the object described by the instance. For example, in case of a gearbox, the information about its function (i.e. speed change) should be stored in the prototype, and only the ratio of this change should reside in the instance of a specific gearbox. Also the information about how a specific gearbox is used in the domain equipment must be kept in the instance. Of course, the prototype-instance scheme ensures that all the general knowledge connected with the prototype is also accessible from instances of this prototype. We found the rich repertoire of programming tools constituting the flavor system in *Symbolics-Lisp* a very convenient vehicle for implementing this strategy.

On the level of prototypes we should apply the principle of generality as well. Hence, for example, we should consider the prototype of a regulating valve as a special case of a valve and have the knowledge characteristic for all possible types of valves connected with the valve prototype. This knowledge could then be propagated down in the hierarchy if necessary. Because the problems of structuring knowledge in the form of prototypes have been extensively investigated (research on frames, scripts, semantic nets, etc.), we won't elaborate on this here. We will comment on only one aspect of the hierarchy of prototypes. It seems to us that, for purposes of equipment modelling, this hierarchy should have the structure of a graph rather than of a tree: its nodes should be allowed to have more than just one immediate parent. We mentioned already that there are regulating valves in our equipment. These are valves whose function is to regulate the medium in some manner, usually changing one of its parameters, like pressure or temperature. We also have other valves whose function is different, for example relief or shut-off valves. Thus, is it conceivable to divide valves into classes according to their function. However, this is not the only dimension along which classification is possible. Valves may be also categorized according to their operating principle as electric, hydraulic or pneumatic valves. Now, the problem with a tree-like taxonomy is that we have to arrange the dimensions linearly: if we decide to consider the functional aspect first, we will have to repeat the division according to the operational aspect at each node of the functional level of the hierarchy tree. With the reversed order of dimensions the problem remains the same. It would be therefore much better to allow a node in the hierarchy to inherit properties from more than one immediately preceding node. The flavor system, with its mechanism allowing flavors to be mixed, proved to be very helpful here.

It's obvious that any real-life equipment deserving a natural language front end is big and complex. For example, the starting air system (our initial domain) consists of several hundred elements each of which may be referred to by its descriptive name and be mentioned in a casaulty report. A good measure of the system's complexity is the size of its description in the ship's manual: 28 pages of text, figures and tables. What is the best way of organizing this vast amount of data into a managable model? Clearly, some simplification is unavoidable. How much? Let us address the former problem first. A salient feature of a piece of equipment is its task, i.e. what it should do. Generally speaking, all complex equipment may be viewed as processors of something - if this something is changed qualitatively into something else (e.g. fuel into rotary movement) we may speak of generators; if only some parameters of this something are changed (e.g. low-pressure air into high-pressure air) we may speak of transformers. Usually only part of the equipment's components are directly involved in this primary task. The rest are there to ensure that special conditions are created at certain points in the equipment. This observation provides us with an important structural hint: we can treat a piece of equipment as a functional system consisting of component systems among which one is responsible for the primary function (the equipment task) and the others fulfill auxiliary functions. If necessary, we may apply the same approach recursively to any of the lower level systems. Systems of this kind may be viewed as chains of components linked together in such a way that, at each node of the chain, the processed substance changes slightly, becoming thus more similar to its desired form at the end of the chain. Many of these components work properly only if special conditions are created. Hence the need for auxiliary systems. Another, more conventional way of structuring the model is in the form of a part/whole hierarchy. A natural question arises: where one should stop with these two types of refinements (in system/basic-part and part/whole hierarchies)? This is a more specific version of the question we posed above: how much to simplify? A possible answer is to refine the hierarchy far enough so that everything which potentially may be referred to in the reports would have a description in the model. This, however, seems impractical. Consider, for example, the following sentence:

Borescope investigation revealed a broken tooth on the hub ring gear.

Considering that there are several different gears in our starting air system and each of them has many teeth which are very much alike, it's obvious that creating a separate description for each of them wouldn't be reasonable. The same remark is true for balls in bearings or for connecting elements like screws, bolts or pins. On the other hand, information about the tooth conveyed in the above sentence cannot go unnoticed. The solution we accepted for such elements is not to include their descriptions in the model on a permanent basis but to keep the possibility open to create and to implant into the model their descriptions if such a need arises during the analysis. A rule of thumb for deciding whether a particular element deserves a permanent place in the model can be formulated in the form of the question: how much information specific to this element is necessary to solve understanding problems, like finding referents (see the section on nominal compounds) or making inferences? As an example of the latter, let's consider a specific gear. We would like to know, among other things, what is this gear's role and place in the modelled equipment so that, in case of its damage, we could determine the impact of this on the equipment. Information of this type can be deduced neither from the analyzed text nor from general knowledge about gears. It must be known in advance. Our way to achieve this is to keep the gear's description permanently in the model.

There are, however, elements like teeth which have so little relevant structure that they are always referred to as ***tooth, teeth*** together with the element higher up in the part/whole hierarchy (let's call such an element a host). Thus, it is not necessary to maintain any specific information about them in the model. It is enough, if we are able to create their descriptions only when they occur in the text. All the possible information we will ever need to include into such descriptions will come from the text. The information relating such elements with other parts of the equipment will come from their hosts. For example, the impact of a tooth's damage on the equipment may be derived from the functional information connected with its host.

It is important to notice that there is nothing absolute in distinctions such as the one made above. It is conceivable to have a piece of equipment of a larger scale than the SAC, where elements like gears are not essential enough for us to be bothered with their shapes or locations; if broken they probably would be referred to by giving the higher-level element of which they are part. In such cases we would rather treat gears like we treat teeth here.

It is desirable to be able to use the model on several levels of abstraction. For some purposes it is enough to treat, say, a speed increasing gearbox as a system for which we only know its outside behavior; in other cases, we would like to use information about its internal structure as well. It should, of course, be possible to deduce the external behavior of an object by analyzing its parts; however, it wouldn't be practical to go down to the level of basic components each time we need to know something about the behavior of the equipment on the intermediate level. Our approach of gradually refined levels of functional systems described above fulfills this desideratum. It seems inevitable that any division into levels will always be artificial and therefore, whatever structure of the model we could design, we always will find sentences which mention objects from different levels. Consider for example:

Believe the coupling from diesel to SAC lube oil pump to be sheared.

In our model for the starting air system the diesel and SAC are at the same level of abstraction. The lube oil pump is two levels below the SAC in the hierarchy. How we solve the problem of determining the referent for the above coupling is described in the section on nominal compounds (see below). Here we want only to point out that for any multi-level model, there must be mechanisms available for moving between abstraction levels flexibly.

In the preceding section we discussed two understanding problems. The solution we proposed there relied heavily on the ability to simulate certain actions and processes of the domain equipment. We have mentioned already in the introduction that it is sufficient to simulate equipment behavior qualitatively. It is clear that the solution to the simulation problem depends a lot on the structure of the model. Therefore, the simulation requirement should be one of the important design criteria for the model. Dividing the equipment into functional subsystems and modelling them as chains of components (comp. above) facilitates the simulation task considerably.

There is another aspect of natural language understanding systems whose satisfactory treatment depends a lot on an effective solution to the simulation problem. We may expect that in real-life cases, the output of such systems is either fed into some expert system or communicated to a human user. In both cases important decisions are presumably made, based on this output - otherwise, why to spend money for building them. It is therefore very important for such systems to provide users with means to check the quality of their understanding. In the case of equipment, one quick and user-friendly way of verifying the analysis is through graphics (we elaborate on this a little more in the section describing future work, below). Because equipment is very dynamic, most texts about them involve actions, events, procedures occuring in a certain time sequence. In order to show this graphically, it is necessary to simulate the essential aspects of this on the screen.

The simulation should be designed in such a way that its two independent applications in the system (i.e. text understanding and communication with users) wouldn't require two seperate simulation systems.

4. The Starting Air System Model

As mentioned above, the equipment we have chosen as our initial domain is the starting air system on Navy ships. Its function is to supply a ship's propulsion gas turbines with the high-pressure air necessary to start the turbines. The main part of the starting air system is its compressor (SAC - Starting Air Compressor). It is by far the most complicated element and therefore is prone to various kinds of damage and malfunction. Because of its importance, we started our efforts by building a model of the SAC. So far we have implemented parts of it on a *Symbolics Lisp* machine using *Zeta-Lisp*.

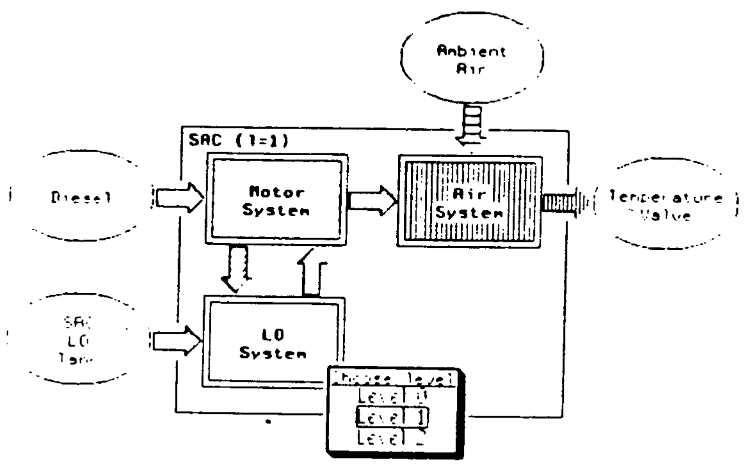

Figure 1. Division of the SAC into subsystems.

Following the guidelines for equipment models given in the preceding section, we divided the SAC into its three functional subsystems (comp. Fig. 1):

(a) Air System - this is the system partially responsible for the SAC's primary task: it takes ambient air, compresses it to the desired pressure and outputs the flow to a system of temperature and pressure regulating valves which precede the turbine starter;

(b) Motor System (auxiliary) - its function is to transmit mechanical rotation from the diesel motor to the compressor blade assembly and lubrication oil pump;

(c) Lubrication Oil (LO) System (auxiliary) - it distributes the oil throughout the SAC and supplies it under pressure to such elements as bearings and some couplings.

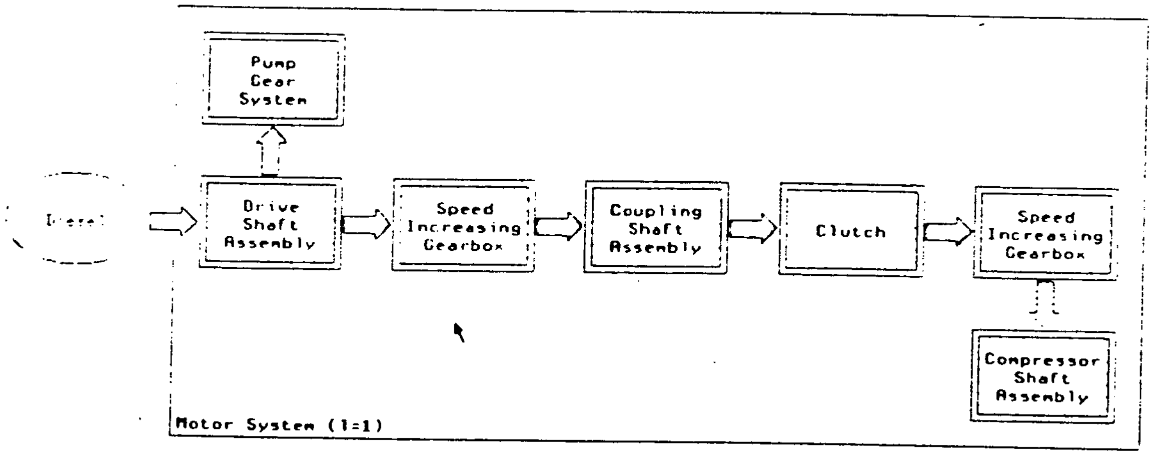

Figure 2. Division of the SAC Motor System into subsystems on level 1.

Each of these three systems may be split into further systems. For example, we view the Motor System as consisting of subsystems shown in Fig. 2. Each of these constituents is again a system consisting of more basic elements. So, for example, one of the two speed increasing gearboxes consists of a hub, a ring gear, an arrangement of three star gears, and a pinion mounted on a shaft.

Every system may be viewed on several levels of abstraction. For example, Fig. 2 shows level 1 of the Motor System. Fig. 3 and 4 show the same system on level 0 and level 2, respectively.

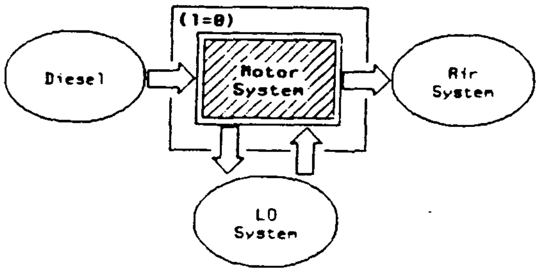

Figure 3. The SAC Motor System on level 0.

All the figures presented here are *Symbolics* screen images generated by PROTEUS from descriptions of the model's elements used for the understanding process. As a matter of fact, we have provided dynamic displays reflecting some of the simulation possibilities of the model. Consider, for example, Fig. 4. It is possible, using the mouse, to position the cursor on, say, the DIESEL ON switch and click on it causing the diesel to be turned on. The compressor starts to run: the small globes inside each of the square elements (from diesel shaft to the clutch) start to rotate in circles with different speeds depending on their place in the system (before or after the speed increasing gearbox); furthermore, all the elements which should be lubricated (those which have in- and outlets in the form of arrows) get oil influx (depicted as dots appearing inside the elements). This follows from the way the SAC operates: the Motor System transmits the rotary movement to the lube oil pump, which starts to work and to supply oil via the LO System (not shown here). Similarly, when we set the clutch to the IN position, the other elements (following the clutch in the chain) will start to rotate. Again, all this is achieved as a side effect of the simulation used for understanding purposes. We want to stress that the "movie" is not the point here. We have to know how the rotary movement propagates in the system, if we want to conduct tests like the one described in section 2, above. Such tests are the primary reason why we equipped our model with a simulation capability.

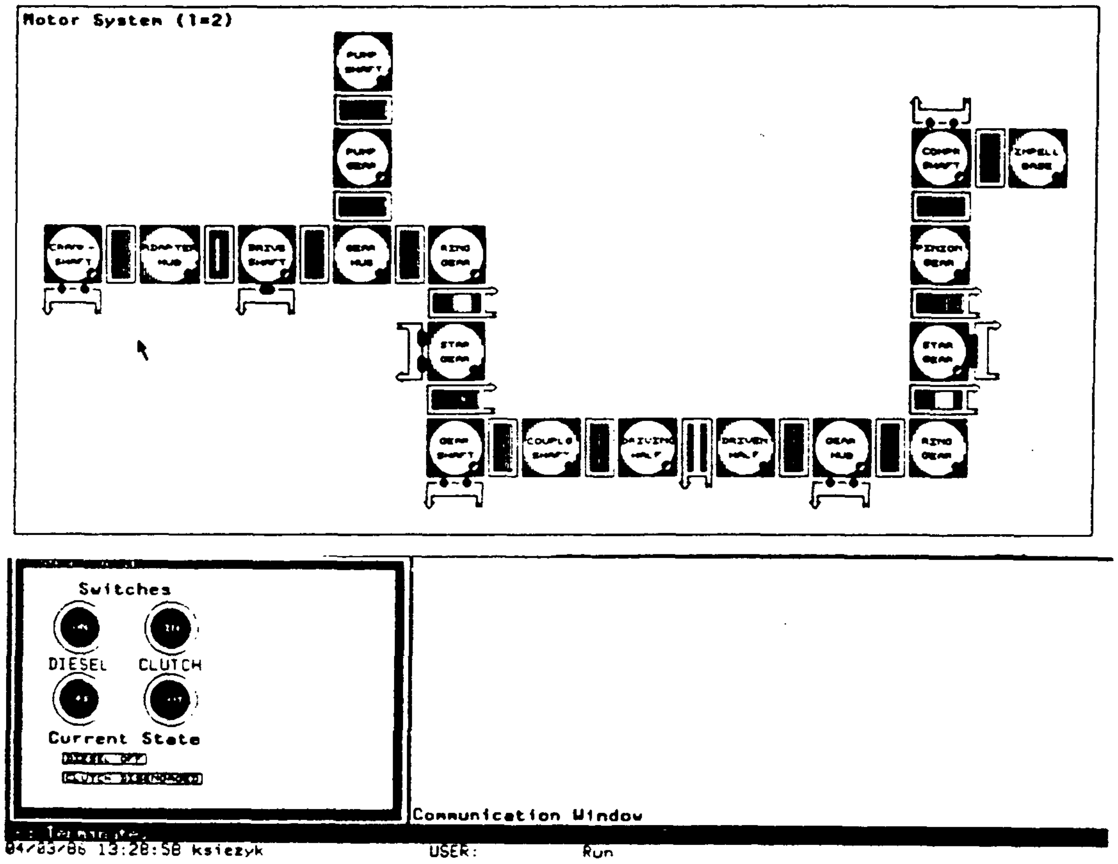

Figure 4. The SAC Motor System on level 2.

Let's turn now to the internal structure of our model for SAC. The structure of the model is based on the *Symbolics-Lisp* flavor system. The prototypes of elements of which the model is built are represented as flavors. The specific elements of the model are encoded as instances of their prototype flavors. The general knowledge about elements is stored in the prototype flavors and can be divided into two parts: (1) declarative knowledge expressed in the form of defaults and restrictions on instance variables; (2) procedural knowledge in the form of methods defined for the flavors. The flavor instances contain only declarative knowledge comprised of instance-variable -- value pairs (we will use more traditional names here: slot -- slot-filler). The prototype flavors are built as mixtures of component flavors, each of which captures a certain aspect(s) of the prototype. The component flavors, which form a graph-like hierarchy, may be viewed as sets of isolated features common to several different prototypes. The sophisticated inheritance mechanism of the flavor system, which works on the level of instance variables (slots) and on the level of methods, allows us to design this hierarchy of flavors in a consise manner. We illustrate these points below with a couple of examples.

Every element which is represented permanently in the model is an instance of a flavor which has the *%building-block* flavor as one of its components flavors (directly or indirectly through intermediate flavors). This reflects the observation that certain facts about model elements will have to be recorded for any kind of element. For example, for every element we want to know its operational state (remember that the texts we are dealing with are about equipment failures) or the system of which it is a part. So, we define:

```
(defflavor %building-block
  (location operational-state part-of screen-location caption)
  ()
  (:settable-instance-variables :screen-location :operational-state)
  :gettable-instance-variables
  (:initable-instance-variables :function :location :part-of)
  (:default-init-plist :operational-state 'OK))
```

In the above definition the first element is the flavor's name, the second is a list of instance variables, the third is a list of component flavors (empty here), and the rest of the definition describes various aspects of instance variables, such as their defaults, how they can be initialized, accessed, etc. (we have omitted this part from flavor definitions given below).

The permanent elements in the model fall into two categories: systems and basic parts. *systems* are those *building blocks* which have structural information. They are chains of elements united by a working substance which they process (for example, the lube oil system). Systems are described at several levels of abstraction. The filler of the *structure* slot is a list of descriptions of the system on different levels - each element in this list specifies, among others, the start and end nodes of the chain of components on this level:

```
(defflavor %system
  (working-substance structure)
  (%building-block))
```

basic parts are those *building blocks* which are at the bottom of the part/whole hierarchy. The *components* slot is initially set to an empty list. It is provided as a destination for those equipment parts which were not included into the model *a priori* but have to be recorded if they occur in the analyzed text (see section 3 for our discussion on this issue).

```
(defflavor %basic-part
  (components)
  (%building-block))
```

Another very common flavor describes the aspects of a building block which capture its role as a component (a node) in some system. It is used as a mixin flavor for building blocks which are systems or basic parts. Its slots record how it is incorporated in the system (*from*, *to* slots) and what its function is with respect to the working substance (i.e. how the substance changes while passing this element). The filler of the *function* slot is a formula interpreted by a method defined for the prototype flavor of the element. This method accesses values of several slots of the instance to which it is applied, for example *input* or *operational-state*. The latter is important because of the potential of failure or damage of the element (see our discussion in section 2, above):

```
(defflavor %system-node
  (from to input output function)
  ())
```

Complex equipment is usually controlled from outside automatically or manually by service personnel. There are, therefore, elements whose operational modes may be changed. Examples of such elements in the SAC are the diesel and clutch. To account for such elements, we defined a flavor *%multiplexer*, which may be mixed with other component flavors to form a prototype. The filler of *switch-locations* is a list of all places from which switching is possible (in our case these are the local and remote control consoles). *switch-actions* specifies for each possible switch position a procedure which has to be run in case the element is set into this position.

```
(defflavor %multiplexer
  (switch-locations switch-actions actual-switch-position)
```

```
    ())
```

None of the above are prototype flavors. They are component flavors which we can use to define prototypes. Let us consider the prototype for diesel motors. It is a piece of equipment complicated enough to treat as a system. Diesels generate rotary movement which is then used to run other pieces of equipment. Thus they are parts of larger systems. Because they run only if a need arises, there must be ways to influence their operational modes. All these facts justify the following definition of a flavor which can serve as a prototype for diesel motors. The point of this definition is to mix together several component flavors corresponding to the just mentioned features.

```
(defflavor %diesel
  ()
  (%system %system-node %multiplexer))
```

Now we are ready to introduce the instance of a specific diesel motor which is part of the starting air system.

```
(setq @diesel-2 (make-instance '%diesel
    ':part-of
          '@ssdg-2
    ':working-substance
          '(ROTATION)
    ':caption
          '("Diesel")
    ':to
          '((ROTATION @sac-2 RIGHT))
    ':from
          '((OIL @container-2 LEFT)
           (AIR @container-1 UP))
    ':function
          '((ROTATION ((OIL . LOW) (AIR . LOW)) . (ROTATION . LOW)))
    ':structure
          '((0 . (DOWN . ((ROTATION OIL AIR) @diesel-2 (@diesel-2) (2 . 2))))
           (1 . (...)))))
```

This instance, its prototype, and the component flavors we showed, are in fact simplified versions of the structures we use in our model. We have included here only these parts which we considered helpful to convey the basic ideas of our prototype-instance scheme used for building the model.

The examples discussed so far demonstrate only the declarative aspect (i.e. the inheritance of instance variables) of the hierarchy we may build using flavors. We also define with each flavor a set of methods which, when combined, provide each instance with a lot of procedural knowledge. It is more difficult to show examples of this because methods are typically long procedures. Describing the *%system-node* flavor above, we mentioned one such method. Similarly, for *%multiplexer* we define a method which, using the data stored in instance's slots, simulates the switching action. Still another example of a method is a drawing procedure which we define for prototypes whose instances may be displayed on the screen. The flavor system supports object-oriented programming. This is reflected in the way methods are invoked - by sending messages (method names) to instances. This allow us to use identically named methods to invoke quite different procedures. For example, it's obvious from looking at the pictures that we use several different drawing procedures. However, we may use the same name, say, *:draw* for all of them. Suppose we have identified an element by locating its instance in the model and want to draw it. We don't have to bother about its prototype in order to know how to draw it - it's enough to send the *:draw* message to this instance. The right method will be chosen automatically. This situation is advantageous for the language understanding process as well. The first thing we do during clause analysis is to find referents in the model (i.e. instances) for linguistic entities occuring in the sentence. The semantics of the verb or predicate adjective is typically expressed in the form of a method. The interpretation of the clause with respect to the model consists then in sending this method to one of the arguments. The part of PROTEUS which deals with the interpretation of clauses hasn't been implemented yet, so we won't go deeper into this subject here.

5. Finding Referents for Nominal Compounds

One notable feature of technical texts is the heavy use of nominal compounds. It seems that their average length is proportional to the complexity of the discourse domain. In the domain of the starting air system, examples like

stripped lube oil pump drive gear and hub ring gear,

are, by no means, seldom occurences.

The problem with nominal compounds is their ambiguity. The syntactic analysis is of almost no help here. Semantically they are also very difficult to deal with [Finin 1986]. The problem may be metaphorically described as a jigsaw puzzle: given several pieces (compound descriptions) put them together to build a sensible picture (nominal compound description). The task becomes somewhat easier in cases when we know that nominal compounds refer to objects existing in the system. In terms of our metaphor it translates into a hint: a set of pictures is given with the assumption that the solution is one of these pictures.

The above observation is the next argument for maintaining an equipment model. Not all nominal compounds fall into this category (a notable class here are verb nomalizations, like *borescope investigation*). However, most of them (especially the longest ones) refer to objects maintained in the model.

PROTEUS processes sentences sequentially (first syntax, then semantics, finally discourse). Both the syntactic and semantic analyzers have been implemented already. [Grishman 1986] describes the overall organization of PROTEUS in some detail. The syntactic component delimits the noun phrases, but does not assign any structure to the pre-nominal modifiers. The interpreter of nominal compounds takes as input an ordered list of words of which the nominal compound consists, and tries to achieve two goals: (1) to determine the structure of the pre-nominal modifiers; (2) to locate the instance(s) in the equipment model referred to by the nominal compound.

The parsing of the nominal compound proceeds bottom-up without backtracking. The words are analyzed from right to left. The parser maintains a **Parse Stack** where all possible partial parses are kept. The information about each partial parse (State Vector) consists of three lists: (1) the **Word List**: the unparsed part of the nominal compound; initially contains the whole compound; (2) the **Forest**: list of partial parse trees for the part of the compound which has been analyzed so far: initially empty; (3) the **List of Referents**: for each partial parse tree in the **Forest**, a list of the model instances which may be named by the words in that partial parse tree.

The condition for a successful parse is twofold: (1) the **Word List** is empty; (2) the **Forest** contains one tree (in such a case the **List of Referents** will, necessarily, also have one list of instances - they will be considered the referents of the compound nominal). The parser works as the following coroutine:

```
LOOP WHILE Parse Stack not empty
  State-Vect = Pop (Parse Stack);
  Word = next word from the Word List of State-Vect;
  Dict-Entry = dictionary entry for Word;
  FOR each reduction rule applicable to State-Vect and Dict-Entry
    Create New-State-Vect;
    IF (termination conditions fulfilled for New-State-Vect)
      THEN return (New-State-Vect)
      ELSE push (Parse Stack New-State-Vect)
```

Each word in the dictionary is assigned two properties: its model class (MOD-C) and its semantic class (SEM-C). We use five different model classes:

Instance **- a word of this class names a set of instances in the model; this set is part of the dictionary entry (in Fig. 5 the word *pump* is an example; *(p1 p2 p3)* are instances of pumps which occur in the model),**

Slot-Filler **- a word of this class can carry information used as slot fillers in some instances; taken alone it doesn't name any model instance (in Fig. 5 the word *lube* is an example),**

Slot-Name **- a word of this class indicates how to interpret some other adajcent words in the compound; an example is *speed* - it tells how to treat *low* in the nominal compound *low speed gearbox*,**

Procedure **- each word of this class is assigned a procedure which, when called with arguments coming from other parts of the noun phrase, returns a referent(s); an example is *coupling*, as in *coupling from diesel to sac lube oil pump* - the coupling meant here is not a single coupling, but a whole sequence of them on the path between diesel and lube oil pump; this sequence has to be evaluated using the model,**

Component **- a word of this class names a set of objects in the domain equipment which are not permanently present in the model (for examples and discussion of this issue see section 3).**

```
        DICTIONARY

(lube   (MOD-C Slot-Filler)            (SEM-C Function))
(oil    (MOD-C Instance (o1 o2 o3))    (SEM-C Working-Substance))
(pump   (MOD-C Instance (p1 p2 p3))    (SEM-C Machinery))
(SAC    (MOD-C Instance (s1))          (SEM-C Machinery))

        SEM-C --> SLOT-NAME TABLE

(Function              :function)
(Machinery             :part-of :components :location)
(Working-Substance     :working-substance)

        INSTANCES

;;; SAC lube oil pump
(setq p3 (make-instance %pump
     ...
  ':part-of 'los2
  ':working-substance '(OIL . o3)))

;;; SAC lube oil
(setq o3 (make-instance %working-substance
     ...
  ':function 'LUBE))

;;; SAC lube oil system
(setq los2 (make-instance %system
     ...
  ':part-of 's1))

;;; SAC
(setq s1 (make-instance %system
     ...
```

Figure 5. Fragments of data used by the parser of nominal compounds.

The two most often used reduction rules are:

(1) *instance + instance --> instance*
(2) *slot-filler + instance --> instance*

In (1), the set of model instances for the result consists of those instances of the second constituent which can be linked through some path in the model to some instance of the first constituent. In (2), the resulting instances are those instances of the second constituent which have a slot whose filler may be matched with the first constituent. The types of links traversed in the search (in the first case) or the checked slots (in the second case) are a function of the semantic class (SEM-C) of the first constituent. This function assigns to each semantic class a set of slot names (see **SEM-C --> SLOT-NAME TABLE** in Fig. 5).

Let us illustrate the way the interpreter works with an example. Fig. 6 shows the trace of parsing *SAC lube oil pump*. We enclosed State Vectors in square brackets; the lists delimited by curled brackets represent (from left to right): the **Word List**, the **Forest**, and the **List of Referents**. The words are represented by numbers; the names (*p1, p2, p3, o1*, ...) are model instances taken from the dictionary (comp. Fig. 5). We analyze the words from right to left. We start with *pump*. We remove it from the **Word List**, find its definition in the dictionary (Fig. 5), and applying a rule not shown above, create the new State Vector (Fig. 6, first vector above the compound). The next word is *oil*. Now, two reduction rules are applicable: the same one we used for *pump* - resulting in the left branch on Fig. 6 and rule (1) above. To apply rule (1), we first find in the dictionary that *oil* is of class *Instance* and names the instances *(o1 o2 o3)*. Next, we try to find out whether any of these instances may be linked to any of the *(p1 p2 p3)*. To do this we take the semantic class of *oil* from the dictionary (Fig. 5): *Working-Substance*. Then we check in the **SEM-C --> SLOT-NAME TABLE** (Fig. 5) which slot names we should consider - the only candidate in this case is *:working-substance*. Finally, we consider each of the instances *(p1 p2 p3)* and check the fillers of their *:working-substance* slots. In Fig. 5 we show only the instance *p3* (the instances *p1* and *p2* are similar). For *p3* we indeed find that it can be linked with one of the considered candidates (namely with *o3*) through the *:working-substance* link. Thus, we include *p3* into the resulting set. A similar analysis for *p1* and *p2* would result in including them into the resulting set as well. Hence, the State Vector in the right branch in Fig. 6 has *(3 4)* as a partial parse tree whose leaves, when combined into one constituent, refer to the set *(p1 p2 p3)*. The analysis at the other points of the trace is similar.

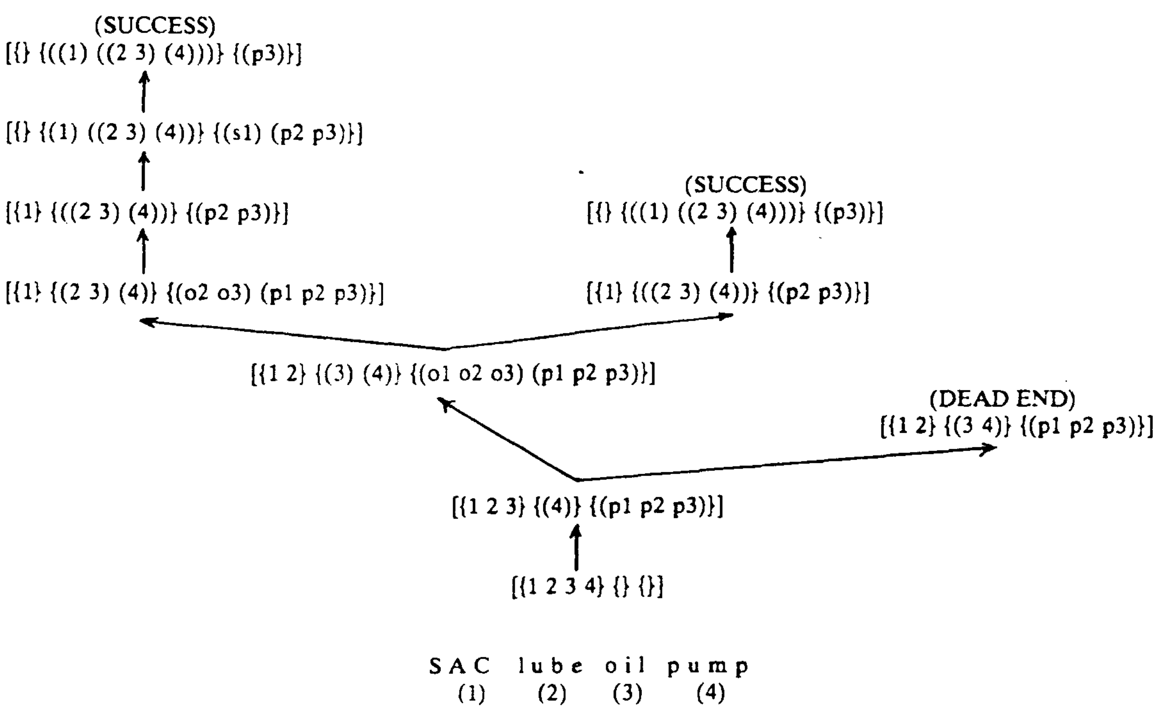
(SUCCESS)
[{} {((1) ((2 3) (4)))} {(p3)}]
[{} {(1) ((2 3) (4))} {(s1) (p2 p3)}]
[{1} {((2 3) (4))} {(p2 p3)}]
[{1} {(2 3) (4)} {(o2 o3) (p1 p2 p3)}]
(SUCCESS)
[{} {((1) ((2 3) (4)))} {(p3)}]
[{1} {((2 3) (4))} {(p2 p3)}]
[{1 2} {(3) (4)} {(o1 o2 o3) (p1 p2 p3)}]
(DEAD END)
[{1 2} {(3 4)} {(p1 p2 p3)}]
[{1 2 3} {(4)} {(p1 p2 p3)}]
[{1 2 3 4} {} {}]
SAC lube oil pump
(1) (2) (3) (4)

Figure 6. The parsing trace for the nominal compound *SAC lube oil pump*.

[Grishman 1986] discusses how to treat modifiers describing the state of a part, such as *cracked* or *sheared*, and also how to handle some ambiguities in conjoined noun phrases (for an example see the beginning of this section).

6. Future Work

The immediate next step in the development of our system is to extend the coverage of the interpreter of nominal compounds to full-fledged noun phrases (including relative clauses, prepositional phrases and conjunctions). Then we plan to work on the interpretation of clauses. It should be possible to define the semantics of most verbs from the domain as operations on the equipment model. Finally, to obtain a robust system, it will be necessary to develop components for finding temporal and causal links between sentences in the text. As is known from previous research (e.g. [Charniak 1977]), success in this area depends mainly on the quality of solutions to the knowledge representation and inference problems. As we indicated in section 2 of this paper, one of the possible approaches to inference mechanism involves the use of a simulation model.

The initial motivation for the system has been the conversion of a stream of messages to a data for subsequent querying, summarization, and trend analysis. However, the use of a detailed equipment model, similar to that employed in simulation systems (e.g. STEAMER [Hollan 1984]), suggests that it may be equally useful as an interface for such systems.

Acknowledgement

This research was supported in part by the Defense Advanced Research Projects Agency under contract N00014-85-K-0163 from the Office of Naval Research.

References

[Bobrow 1977] Bobrow, D. and Winograd, T. An overview of KRL - a knowledge representation language. *Cognitive Science*, 1977, 3-46

[Charniak 1977] Charniak, E. Inference and knowledge in language comprehension. In *Machine Intelligence 8*, D. Michie, Ed. American Elsevier, New York, 541-574

[Finin 1986] Finnin, T. Nominal compounds in a limited context. In *Analyzing Language in Restricted Domains*, R. Grishman and R. Kittredge, Eds. Lawrence Erlbaum Assoc., Hillsdale, NJ

[Grishman 1986] Grishman, R., Ksiezyk, T., and Nhan, N.T. Model-based analysis of messages about equipment. Submitted to the *AAAI-86*

[Hollan 1984] Hollan, J., Hutchins, E., and Weitzman, L. STEAMER: an interactive inspectable simulation-based training system. *AI Magazine*, Summer 1984, 15-27

Reference Guide to Symbolics-Lisp, *Symbolics*, Cambridge, MA, 1984