24

Purposive Understanding

R. C. Schank and G. DeJong

Computer Science Department
Yale University, USA

1. INTRODUCTION

For the past ten years we have been working on the problem of getting a computer to understand natural language. In the beginning, work centred on the problem of parsing. We built an early version of a parser that mapped from English into a language-free representation of the meaning of input sentences (Schank and Tesler, 1969). Simultaneously we worked on the meaning representation itself. We developed Conceptual Dependency which represents meaning as a network of concepts independent of the actual words that might be used to express those concepts (Schank, 1969).

Over the years the parser and the representation evolved as we began to understand the complexity of the problem with which we were dealing. Conceptual Dependency began to rely more on underlying primitives for the representation of the similarities in meaning that transcend the particular words of a language (Schank, 1975). Similarly, our parser developed into a program that relied less on syntactic information to aid it than on predictions that could be obtained by exploiting the properties of the conceptual representation into which we were mapping (Riesbeck, 1975).

We began at this point to worry about the possible use of the conceptual parse that we were producing. We built an inference program (Schank and Rieger, 1974) that exploited the properties of the primitive concepts uncovered by the parser and derived new information from them. These inferences then added information to the conceptual content extracted by the parser. One problem with the inferencer was that it was hard to control. It made inferences without regard for their need. This was part of Rieger's (1975) theory of inferences, but its effect on our programs was to make them rather purposeless and slow.

At this point we developed a theory of understanding of connected text. Part of the problem that we had with controlling inference was due to the fact that we had been working on sentences taken out of context. Sentences con-

sidered in the context of the rest of the text in which they were contained in a sense pointed the way towards the appropriate and necessary inferences. Thus, an inference was considered to be crucial if it helped to tie together the sentences of a text. The end product of such an inference procedure was a connected causal chain of events that represented the implicit and explicit information in a text (Schank, 1974).

At this point we began to program a computer understanding system that would attempt to process input texts. An item crucial to our ability to accomplish this task was what we called a script. A script is a frequently repeated causal chain of events that describes a standard situation. In understanding, when it is possible to notice that one of these standard event chains has been initiated, then it is possible to understand predictively. That is, if we know we are in a restaurant then we can understand where an "order" fits with what we just heard, who might be ordering what from whom, what preconditions (menu, sitting down) might have preceded the "order", and what is likely to happen next. All this information comes from the restaurant script.

The method of processing text outlined above is analgous to that used in parsing. Once we have a well-defined idea of what belongs in the conceptual representation we can determine what is missing and go back into the text to look for it. Thus in both parsing and script application, processing is bottom up until enough information is available to allow the switch to top down.

The program we built to understand texts by the use of scripts is called SAM and is described in Schank et al. (1975), Schank and Abelson (1977) and Cullingford (1977). The program works on the domain of newspaper stories. It does a tremendous amount of detailed processing including inferencing, reference specification, disambiguation, and memory simulation. The program can produce summaries, long paraphrases, translations into other languages as well as answer questions about the input story.

SAM is a very complicated program. It attempts to understand exhaustively and completely. Because of this SAM has two major problems: First, it is rather slow. Although there are ways in which it could be speeded up, it is not at this point the kind of program one would want to have running all the time on one's system. Second, it is a bad simulation of how people actually read newspapers. We built SAM to test out our ideas about how people read as well as to attempt to get a program to read. Obviously there are many kinds of texts besides newspapers, but there is a common basis to reading that transcends what you read. However, we began to wonder if there wasn't something special about newspaper reading that we could exploit so as to build a faster and thus more useful program.

One obvious thing is that people tend to read newspapers with a purpose. They do not read every article nor do they read every word of the articles that they choose to read. People skim until they decide to read in detail.

It seemed reasonable to us to attempt to model this skimming process, using what we had learned about the process of understanding in general. In other words we set out to build a program that would be an extremely top down

system. It would know what it wanted to know or what it was interested in and go out and look for it. Finding what it wanted would cause it to generate new goals to find related information that would amplify what it had found. This purposive understanding program we named FRUMP for Fast Reading and Understanding Memory program.

UNDERSTANDING WITH A PURPOSE

Although there undoubtedly are people who read every word in their daily newspaper, most people do not. The usual procedure is to scan the paper for items of interest, perhaps by starting in a relevant section of the paper, or else by starting randomly. When an interesting headline is found then the reader begins to read. He may read in several modes:

(1) If the story is a completely new one, he may read every word until he becomes tired or the article is finished.
(2) If the story is an update of a currently running story, he may scan for the new information present in the story.
(3) If the story is of a generally continuing nature, he may go quickly through it, looking for items relevant to his particular interests.
(4) If the story is in the reader's domain of special interest, he may read every word and make all possible inferences.

These four modes of reading exemplify four very different processes. Completely new stories occur somewhat less frequently than the others. Examples might be earthquake reports, plane crashes, assassinations, the introduction of a new bill in the legislature, a special announcement by a political figure. Mode 1 stories have usually a "one-time-only" nature, or else they are the beginning of a continuing story.

Mode 2 stories usually contain little if any new information for someone who has kept up to date on the story. Newspapers are written in such a fashion that people who have not kept up can still figure out what was going on. Thus, for example, recent long-running stories in the news such as the Patty Hearst kidnapping or the Philadelphia Legionaires' disease, continually retell the initial setup of the story in each subsequent report. Readers who already know these facts skim them to search for new developments. Often these developments are told in the headline and lead paragraph, and readers will stop there.

Mode 3 stories are probably the most common type of newspaper story. Continuing situations such as the Middle East situation or the energy shortage are problems about which there are usually one or two new developments per week in peak periods, drifting down to a few items a month in slower periods. Some of these kinds of updates occur only once or twice a year in a situation that is very long-term, for example, stories about high taxes and socialism in Sweden.

To read a mode 3 story, a reader must have an interest in the situation in general, along with one or more specific interests related to the situation. Thus

a businessman might be interested in the possibilities for trade with Israel and nothing else about the Middle East. Similarly, a socialist might want to read only about the successes of the government in Sweden and care little about party infighting, for example. Such a goal-oriented reader is easier to simulate than a more passive reader. If we know what we are looking for, we can go in and find it. We need not be disturbed by the complications involved in reading where we have no interest. An obvious advantage for computers here is that we also need not put in all possible knowledge for a situation, where situation is defined very broadly. Rather, by limiting our domains of reading ability to those that coincide with what we are interested in, we solve both the problem of too broad domains causing unbounded amounts of knowledge to be needed and the problem of having to do a lot of useless work to read stories or parts of stories in which we have no interest.

Mode 4 corresponds most to what we normally assume to be the process of reading. However, as we have been arguing, this "normal" reading mode may not occur all that often in newspaper reading. It might occur for reading columnists, or reports of a game in which a favourite team is playing, or other special interest kinds of things.

The four modes correspond to our programs as follows:

mode 1 – this is what SAM does
mode 2 – this is what FRUMP does in update mode
mode 3 – this is what we are designing FRUMP to do
mode 4 – this is what the ultimate reading program would do.

An important point here is that there is a class of stories that we described as being mode 1 that are best read by a FRUMP-type program rather than by a SAM-type program. That is, new events such as car accidents or earthquakes need not always be read for all the detail they contain. Thus, we decided that FRUMP's first task would be in the area of mode 1 stories. Next we worked on updating those stories by quick skimming (mode 2). We are aiming eventually for mode 3 ability in FRUMP.

THE GENERAL STRATEGY

We will assume that a reader should approach a newspaper story as follows:

1. Reading headlines

We have four options here:

(a) quit and forget
(b) quit and remember
(c) go on in a careful detailed manner
(d) go on and skim.

2. Skimming

(a) old story recognized: fill in and update information

(b) new story: scan for concepts predicted by the domain of the story
(c) any story: scan for key concepts of special interest.

3. Updates

(a) look for the continuation of a previously instantiated script
(b) look for unfulfilled expectations from the last reading
(c) look for update-specific problems (for example, the death toll mounts in an earthquake each day).

4. Special interest concepts

(a) instantiate scripts specific to special interest
(b) begin reading carefully when a key concept is found.

A FRUMP RUN

With the above in mind we began to build FRUMP. FRUMP was designed to work on mode 1 and mode 2 kinds of news stories. In theory, when it becomes interested is something it could send control to SAM; however, we have not actually tried to implement this.

To demonstrate its understanding, FRUMP produces summaries of what it has read. Since its basis is language-free Conceptual Dependency type scripts, its summaries can come out in any language, thus Russian and Spanish summaries are produced in addition to English ones. FRUMP's scanning ability is based on abstracting out the principles behind SAM. It is by no means a key-word parser. Its parser is connected to the scripts that it has which describe news situations. It only looks at what it is interested in, ignoring the rest. It is thus a very fast program. FRUMP can understand and produce a brief summary of a 150-word news article taken directly from a newspaper in about 5 seconds of CPU time on a DEC KA10 processor.

To give an idea of FRUMP's capabilities we will now show an example of a run of FRUMP:

Sample run of FRUMP

INPUT:

10-11 CHIHUAHUA, MEXICO, – A PASSENGER TRAIN CARRYING TOURISTS, INCLUDING SOME AMERICANS, COLLIDED WITH A FREIGHT TRAIN IN THE RUGGED SIERRA MADRE OF NORTHERN MEXICO, KILLING AT LEAST SEVENTEEN PERSONS AND INJURING 45, THE POLICE REPORTED TODAY.

THEY SAID THAT AT LEAST FIVE OF THE INJURED WERE AMERICANS, AND THERE WERE UNOFFICIAL REPORTS THAT ONE OF THE DEAD WAS FROM NEW YORK CITY.

SOME OF THE PASSENGERS WERE TRAVEL AGENTS, MOST FROM MEXICO CITY, MAKING THE TRIP AS PART OF A TOURISM PROMOTION, THE POLICE SAID.

THE AMERICAN SOCIETY OF TRAVEL AGENTS HAD BEEN MEETING IN GUADALAJARA, THOUGH IT WAS NOT KNOWN WHETHER ANY OF THE GROUP WERE ABOARD THE TRAIN.

ONE OBSERVATION CAR ON THE RAILROAD TO THE PACIFIC TUMBLED INTO A 45 FOOT CANYON WHEN THE PASSENGER TRAIN SMASHED INTO THE FREIGHT YESTERDAY AFTERNOON NEAR THE VILLAGE OF PITTORREAL ABOUT 20 MILES WEST OF CHIHUAHUA CITY AND 200 MILES SOUTH OF THE UNITED STATES BORDER, THE POLICE SAID.

THEY SAID THAT RESCUE WORKERS WERE STILL TRYING TO PRY APART THE CAR'S WRECKAGE TO REACH PASSENGERS TRAPPED INSIDE. THE RESCUE SQUADS COULD NOT USE CUTTING TORCHES ON THE WRECKAGE BECAUSE SPILLED DIESEL FUEL MIGHT IGNITE, THE POLICE REPORTED.

SELECTED SKETCHY SCRIPT $VEHACCIDENT

DONE PROCESSING
SATISFIED REQUESTS:

```
((<=> ($DATELINE LOC &DLOC MONTH &MON DAY &DAY)))
&DLOC
        CLASS       (#LOCATION)
        LOCALE      (*MEXICO*)
        SATISFIED   ((10) (11))
&MON
        NUMBER      (10)
        SATISFIED   (NIL (11))
        CLASS       (#NUMBER)
&DAY
        NUMBER      (11)
        SATISFIED   (NIL NIL)
        CLASS       (#NUMBER)

((<=> ($VEHACCIDENT VEH &VEH OBJ &OBJ LOC &LOC)))
&VEH
        CLASS       (#PHYSOBJ)
        TYPE        (*VEHICLE*)
        SROLE       (&TRAIN)
        SCRIPT      ($TRAIN)
        SATISFIED   ((10) (11))
&OBJ
        CLASS       (#PHYSOBJ)
        TYPE        (*VEHICLE*)
        SROLE       (&TRAIN)
        SCRIPT      ($TRAIN)
        SATISFIED   ((10) (11))
```

```
&LOC
        CLASS       (#LOCATION)
        LOCALE      (*MEXICO*)
        SATISFIED   ((10) (11))

((ACTOR &DEADGRP TOWARD (*HEALTH* VAL (-10))))
&DEADGRP
        NUMBER      (17)
        SATISFIED   ((10) (11))
        CLASS       (#PERSON)

((ACTOR &HURTGRP TOWARD (*HEALTH* VAL (-LT10))))
&HURTGRP
        NUMBER      (45)
        SATISFIED   ((10) (11))
        CLASS       (#PERSON)

CPU TIME = 7.522 SECONDS

SUMMARY:
17 PEOPLE WERE KILLED AND 45 WERE INJURED WHEN A TRAIN
CRASHED INTO A TRAIN IN MEXICO.
```

HOW FRUMP WORKS

The basis of FRUMP is the script. However, rather than using a script like SAM's, FRUMP uses what we call a sketchy script. The crucial difference is that sketchy scripts have far fewer conceptual dependency representation (only those corresponding to the most important events in SAM's scripts) and more often than not, the causal connections between conceptualizations are not included. The result is that FRUMP understands most of what is important to understand in news articles and works very much faster than SAM. The article of several paragraphs that takes SAM a quarter of an hour to understand can be processed by FRUMP in under ten CPU seconds.

When FRUMP begins to read a newspaper story, it already knows what facts it wants to find. For each type of newspaper story, FRUMP has a list of expected facts that it wants to see. These expectations are called "requests". The collection of all the requests for one type of story makes up the "sketchy script" for that story type. In the remainder of the paper, when we refer to a script we will mean FRUMP's sketchy scripts, not SAM's scripts unless otherwise noted.

In understanding an article, FRUMP must select a script and then try to find occurrences in the article of the facts represented by the requests. Requests are in Conceptual Dependency format and contain unfilled slots. These slots are called "script variables". Understanding an article consists of finding the information corresponding to a request in the text and filling in the slots (binding the script variables) in that request. When an instance of one of the requests is

found in the text and the script variables have been bound, that request is said to be satisfied. The process of satisfying the requests of a script is called instantiating that script. The number of requests in each script is small. The requests correspond to the most important information in a particular type of story. For example, the vehicle accident script used in the sample run above contains four requests. The first request in the vehicle accident script will be satisfied when FRUMP finds the type of vehicle in the accident, the object that the vehicle collided with, and the location of the accident. FRUMP can satisfy the second request by finding the number of people killed; the third by the number injured, and the fourth by who was at fault in the accident. When all of these requests are satisfied by a story. FRUMP knows all that it wants to know about that news event. The rest of the article will be ignored.

When FRUMP is given a new article to understand it skims the first paragraph for identifying information. This information is used to find the appropriate script to use to understand the article. Once the script is identified, FRUMP begins skimming the article.

FRUMP is composed of two conceptually different parts: a parser and a script applier. The parser FRUMP employs was inspired by Becker's phrasal lexicon (Becker, 1975) which was presented at the TINLAP conference in 1975 but unfortunately was not actively pursued by him after that. The parser parses phrases from the text into Conceptual Dependency representations. The script applier then matches these conceptual representations against the requests in the script. When a match is found, the fillers in the parser representation are used to bind the script variables occurring in the request.

FRUMP uses the same language-free system of representation as is used by Riesbeck's parser (Riesbeck, 1974) which is used in SAM. Yet FRUMP's parser is very different from the parser in the SAM system. The Riesbeck parser parses an entire sentence at a time. There is very little communication allowed with the script applier of SAM during parsing; FRUMP's parser is concerned only with parsing parts of a sentence. In SAM the parser and script applier are very distinct. As a result, each does its thing with little influence over the other. FRUMP, however, is much more integrated. The parser and script applier do different tasks, but the precise division between the two is fuzzy. The advantage of the fuzziness is that the two modules can communicate with each other freely. This allows the script applier to control parsing to an extent not possible in SAM. FRUMP's script applier can tell the parser which of several interpretations is correct or even to stop trying to parse the current input text.

THE CONCEPTUAL FRAGMENT PARSER

FRUMP's parser is very top down. It is driven by the high-level requests of the sketchy scripts and works very closely with the script applier. The parsing strategy is to find conceptual fragments from the input text being skimmed that will satisfy all or part of some script request. Important properties of the actual conceptual referents in the text are then copied to variables in the conceptual

representation from FRUMP's dictionary. Finally, this conceptual representation of the meaning of the input text is returned as the parse. Dictionary entries for each word are made up of a list of meaning fragments in conceptual dependency format. For each different meaning there is a series of context tests that must be satisfied before this meaning can be realized as the parse. There are also instructions on how to bind variable role fillers in the conceptual dependency representation. FRUMP has two dictionaries: one is the conceptual fragment dictionary described below; the other contains information about objects and forms and tenses of entries in the other dictionary.

The following is the conceptual fragment dictionary entry for "strike".

```
( (BKWRDS (M1 (TYPE (*VEHICLE*))))
  (FRWRDS (M2 (CLASS (#PHYSOBJ))))
  ((MERGE P1 M1) (MERGE P2 M2))
  ((<=> ($VEHACCIDENT VEH P1 OBJ P2))) )
```

The context tests are arranged in two lists which search backward and forward respectively from the entry word for words that will satisfy the context tests. If a context test is a single word, that word must be present in the input text. If the context test is a list, it is satisfied by finding a word in the input text that has all the properties specified, and copying them to a temporary variable. For example, the first line in the above dictionary entry searches backward from a form of the word "strike" for a vehicle. If it finds one, all the properties of that vehicle are copied to the temporary variable M1, and the forward tests are evaluated. The forward tests require that the word "strike" be followed by a physical object. If all the context tests are satisfied, the properties of the temporary variables are copied to the role fillers in the conceptual representation. This representation is then returned as the parse.

The output of FRUMP's parser is not modified English as it is in Colby's system (Parkinson, Colby, and Faught, 1976), but a language-free conceptual representation. The advantage of a language-free representation is that different phrases with the same meaning will be parsed into the same representation. This in turn means that the test to see if a request is satisfied by a parse is very efficient. It also makes generating summaries in different languages no harder than generating the summary in English.

EXAMPLE OF A SKETCHY SCRIPT

One of the scripts that FRUMP has is a vehicle accident script. All vehicle accidents, whether they are train wrecks, boat collisions, plane crashes etc., have many things in common. In a vehicle accident story there is always a vehicle and there is always an object that it collides with. There is always the possibility that a number of people are killed or injured. In addition, in newspaper stories, the cause of the collision is often reported. These are the important points of a vehicle accident, and these are what FRUMP tries to find out when reading a vehicle accident article. There are, of course, many other things that can happen

in a collision. For example, the injured people are often taken to a hospital, for auto crashes there is often a policeman called to the scene etc. These are less important facts, and unless there is some special reason for noticing them, a human skimming the article will usually miss them. FRUMP also ignores these lesser points. The vehicle accident script currently consists of four requests at three important levels.

```
Request R1
  value:  (((<=> ($VEHACCIDENT VEH &VEH OBJ &OBJ LOC &LOC)))
           ((EQU (<=>) $VEHACCIDENT))
           (PROP (<=> VEH) *VEHICLE* TYPE)
           (PROP (<=> OBJ) #PHYSOBJ CLASS)
           (PROP (<=> LOC) #LOCATION CLASS))
  importance: 0

Request R2
  value:  (((ACTOR &DEADGRP TOWARD (*HEALTH* VAL (-10))))
           ((EQU (TOWARD) *HEALTH*)
            (EQU (TOWARD VAL) -10))
           (PROP (ACTOR) #PERSON CLASS))
  importance: 1

Request R3
  value:  (((ACTOR &HURTGRP TOWARD (*HEALTH* VAL (-LT10))))
           ((EQU (TOWARD) *HEALTH*)
            (EQU (TOWARD VAL) -LT10))
           (PROP (ACTOR) #PERSON CLASS))
  importance: 1

Request R4
  value:  (((<=> ($FAULT ACTOR &ACTOR)))
           ((EQU (<=>) $FAULT))
           (PROP (<=> ACTOR) #PERSON CLASS))
  importance: 2
```

Before FRUMP's processing can be understood in detail, one must understand the requests of its sketchy scripts. As a typical example of FRUMP's requests consider request R2 above. The first line of the request is the Conceptual Dependency representation of the request. &DEADGRP is one of variables of the vehicle accident script. (*HEALTH* VAL (-10)) is the Conceptual Dependency representation for dead. When the variable &DEADGRP is bound to something, the meaning of this Conceptual Dependency representation is that the something it gets bound to is dead. The next three lines are constraint tests that are applied to the output of FRUMP's parser. If the parser yields a representation that passes all of these tests, the script variables contained in the request are bound to the corresponding conceptual role fillers in the parser representation, and the request is satisfied. The first test in the

example request checks that the filler of the TOWARD role in the parser representation is *HEALTH*. The second test checks that the filler of the VAL slot in the filler of the TOWARD slot is −10. The third test checks that the filler of the ACTOR slot (that is the thing that the parser proposes should be bound to &DEADGRP in the request) is of conceptual type PERSON. This means that the only thing that can be bound to &DEADGRP is a person or group of people. In fact the parser is set up so that only groups will be generated in this slot. One of the important properties of groups is the number of things in them. When &DEADGRP is bound to something, all of its properties are copied over to the script variable &DEADGRP. Therefore one of the pieces of information available from this request after it is satisfied (and indeed the most important datum of this request) is the number of people who were killed. The observant reader will have realized that the first two tests in the request are set off from the third one by parentheses. This is to group the tests by whether or not they involve one of the script variables. The first two tests in the example do; the third does not. The grouping makes the understanding process more efficient. This will become clear in the next section.

HOW FRUMP UNDERSTANDS

Once FRUMP has the correct script to use, it starts to scan the article, looking for conceptual fragment words. When it finds one, it retrieves the dictionary entry for that word. Recall that the dictionary entry consists of a list, each element of which contains context tests and a representation that corresponds to one meaning. FRUMP tries to realize each meaning one at a time until all the context tests are satisfied or the dictionary list is exhausted.

The processing of each dictionary word sense or possible meaning consists of first making a list of all the outstanding requests the conceptual representation of this meaning might satisfy. This is done to avoid evaluating all of the context tests of a word sense that has no chance of satisfying a request, and to limit the number of requests that the parse needs to be matched against. A request is included in the list only if all of the first group of role-filler tests of this request are satisfied. Remember that the first group role-filler tests are all tests that do not reference one of the script variables. Therefore these tests can be made before the script variables or the variables in the context tests are bound. For example, if while processing a vehicle accident story, the parser found a word that indicated ((ACTOR P1 TOWARD (*HEALTH* VAL (−10)))) might be a parse, the list of possible requests would be just (R2). The tests that do not look at script variables require that the <=> filler for R1 be $VEHACCIDENT, the filler of the VAL slot in the TOWARD slot must be −LT10 for R3, and the <=> slot in R4 must be filled with $FAULT. Thus the only request that might be satisfied is R2. If the list is empty, there is no need to evaluate any of the context tests; it cannot satisfy a high-level request. At this point, none of the context tests have been evaluated, so that this word sense might not be correct. However, the list is very cheap to create and usually

cuts down on processing later. This is the reason for separating the request tests that reference script variables from those that do not. If the list contains at least one element, the context tests for this meaning are evaluated one at a time. If there is at least one context test that fails, this sense is not a proper reading of the text. If all the word sense fail, the word that was found was a false alarm, and FRUMP reverts to the original scan. If all the context tests are satisfied, the conceptual representation is filled out with the variables bound while evaluating the context tests. This representation is then matched against elements of the list of potentially satisfiable requests made earlier. If one matches, the script variables in that request are bound to the role fillers of the representation, the request is marked with the date of the article, and it is marked as satisfied. At this point, FRUMP can dynamically modify the sketchy script. Associated with each script variable can be a set of demons which are checked when the script variable is satisfied. They can make arbitrary tests and load or delete requests. Thus, it is possible, for example, to have FRUMP look for aid to a country if it is hit by a severe earthquake but not if it was a mild quake. This amounts to high-level inferencing and makes FRUMP much more efficient by eliminating the need to process large numbers of very specialized requests, the specialized requests are not loaded until they are needed. After processing the satisfied request, FRUMP continues its scan for conceptual fragment words. If no request fulfils the detailed match, FRUMP reverts to the original text scan. Notice that there were three ways in which parsing can be discontinued; in these cases FRUMP does just enough work to realize it is working on a bad parse. This is in a large way responsible for FRUMP's efficiency in processing and is directly attributable to the broad communication between the parser and the control structure that makes up the script applier.

When FRUMP has finished processing an article, some requests will have been satisfied but, very likely, others will not. The script has been partly instantiated. This partly instantiated script is then stored on a disk file. If FRUMP should later come across an article updating this news event, it can then retrieve this partly instantiated script and continue satisfying requests where it left off.

DECIDING ON A SCRIPT

Presented with an article FRUMP chooses one of three following ways to process it. First, it can decide that the article is an update of a news event that it has previously processed and select the partly instantiated script from that article to understand with. Second, it can decide that it is the first article of a news event and select the appropriate virgin script. Third, it can fail to recognize the article as one of the types of events for which there exists a script, in which case it will ignore the entire article. The choice is made from information gleaned from a preliminary scan of the article's first paragraph. This scan is made with a special set of active requests.

There is for each script one key request which, if satisfied in the text,

strongly indicates that its script is appropriate to understand the article. For example, the key request for the vehicle accident script is ((<=> ($VEHACCIDENT VEH &VEH OBJ &OBJ LOC &LOC))): here &VEH, &OBJ, and &LOC are script variables which get bound to the vehicle, the object collided with, and the location of the accident respectively. Furthermore, owing to the style of newspaper writers, this request seems always to be satisfied in the first paragraph (and usually the first sentence). The special set of requests is therefore composed of the key request from each virgin script that FRUMP has.

The first paragraph is skimmed until one key request is satisfied. FRUMP now knows which script type the story is and also some information about the story. In the case of the vehicle accident it knows what the vehicle and object are and where the accident occurred. This information is used to decide if the current article refers to a previous news event or a new one. After a sketchy script is partly instantiated by the first article of an event, it is stored away. The type of script it is and the key information about the event are stored specially. After a new article's first paragraph is skimmed, the key information gained is matched against all stored scripts of the same type. If a stored, partly-instantiated script is found that matches, it is brought into core and used to understand the new article. If no previous script is matched, a virgin script with no requests satisfied is used to understand the story. When it is finished, FRUMP writes this partly instantiated script out on the disk file so that any update articles that it finds will have access to it.

UPDATING STORIES

There are three main types of update that FRUMP must handle. These types correspond to pieces of information and not to articles, so that an update article can cause more than one type of update to be made. The update types differ from one another by how the new information is added to the partly instantiated sketchy script.

In the first kind of update, information is only added to a sketchy script. That is, a new article is found to refer to the same news event as previous articles and it supplies information that satisfies a request that was never before satisfied. This is the simplest type of update and is handled as follows: After FRUMP finishes an article, the sketchy script is written out to a file with the key identifying information discussed above. Some of the requests will have been satisfied and some will not. All the requests, whether satisfied or not, are written on the file. When this partly instantiated script is read back into core, the unsatisfied requests are, of course, still active. On reading the update article then, this type of update is treated exactly as if it were a virgin script. When a previously unsatisfied request is satisfied, it is marked as satisfied and tagged with the date of the newspaper.

In the second type of update, the information in the update article replaces

the information in an already satisfied request. In this type, generally only one request is changed at a time, and the change is a direct modification of one or more role-fillers of the request. All requests, whether satisfied or not, are processed during understanding as if they are not. When the role-fillers of a request are to be bound, the date that they were last bound is compared to the date of the current article. If the current article is later, the fillers are updated. If not, the information from the current article is thrown away.

The third type of update is the most complicated. There are many news event types where an arbitrary number of similar sub-events can occur. These sub-events themselves may be rather complex. For example, an earthquake may be followed by any number of aftershocks. Each aftershock may cause death and injury. The recent fighting in Lebanon was made up of a number of individual clashes. Oil from a leaking tanker can wash ashore in several places at different times, each causing different kinds and varying degrees of damage to the shoreline. There are three things to notice about such updates. First, they add new rather than replace old information. Therefore, they must be processed by as yet unsatisfied requests. Second, the structure of each sub-event can be complex so that it cannot be represented by one request alone. Third, since the initial requests for each sub-event must be the same, there is the possibility for any number of copies of the same request to exist in a script each satisfied by a different sub-event. For example, an earthquake and two of its aftershocks may all cause people to be killed. In this earthquake script, then, there will be three copies of the request ((ACTOR &DEADGRP <=> (*HEALTH* VAL (−10)))) each satisfied with a different &DEADGRP.

The solution to these problems of the third update type is to organize the requests corresponding to sub-events into bundles. The script will always have one fresh copy of the bundle active and completely uninstantiated. When a bundle is about to be partly instantiated (that is, when at least one of its requests is to be satisfied) a copy of the uninstantiated bundle is made and these new requests are added to the script. Then FRUMP continues instantiating the bundle. This enables FRUMP to understand any number of the sub-events. Of course, an article could update an update article (for example, revising the death toll caused by an aftershock of an earthquake). If one of the requests in a particular bundle has to be changed, FRUMP must first identify the bundle. After that the update can be handled exactly as the type two updates above.

Bundles are subsections or scenes of scripts. They are very similar to scripts in many ways. In particular, they can always be differentiated by key information. This key information is often simply the date or location of the sub-event. For example, a newspaper report might update the death toll from last Thursday's aftershock of the earthquake that struck Eastern Turkey five days ago. Five days ago, Eastern Turkey, and the fact that it is an earthquake, are used to find the original script. Within this script, FRUMP then finds the bundle of requests for the proper aftershock by matching the date of each to the date last Thursday. When it finds the correct bundle, FRUMP finds the

request corresponding to the number of people killed and updates the proper script variable.

More FRUMP output

The following news stories were taken directly from the *New York Times* and the *New Haven Register.* The Spanish summarizer was written by Jaime Carbonell and the Russian summarizer by Anatole Gershman.

```
INPUT:
    2 - 4 ITALY - - A SEVERE EARTHQUAKE STRUCK
NORTHEASTERN ITALY LAST NIGHT, COLLAPSING ENTIRE
SECTIONS OF TOWNS NORTHEAST OF VENICE NEAR THE
YUGOSLAV BORDER, KILLING AT LEAST 95 PERSONS AND
INJURING AT LEAST 1000, THE ITALIAN INTERIOR MINISTRY
REPORTED.
    IN THE CITY OF UDINE ALONE, A GOVERNMENT SPOKESMAN
SAID THEY FEARED AT LEAST 200 DEAD UNDER THE DEBRIS.
THE CITY, ON THE MAIN RAILROAD BETWEEN ROME AND VIENNA,
HAS A POPULATION OF ABOUT 90000.
    THE SPOKESMAN FOR THE CARIBINIERI, THE PARAMILITARY
NATIONAL POLICE FORCE, SAID THERE HAD BEEN REPORTS OF
SEVERE DAMAGE FROM HALF A DOZEN TOWNS IN THE
FOOTHILLS OF THE ALPS, WITH WHOLE FAMILIES BURIED IN
BUILDING COLLAPSES. COMMUNICATIONS WITH A NUMBER OF
POINTS IN THE AREA WERE STILL OUT.
    THE EARTHQUAKE WAS RECORDED AT 6.3 ON THE RICHTER
SCALE, WHICH MEASURES GROUND MOTION. IN POPULATED
AREAS, A QUAKE REGISTERING 4 ON THAT SCALE CAN CAUSE
MODERATE DAMAGE, A READING OF 6 CAN BE SEVERE AND A
READING OF 7 INDICATES A MAJOR EARTHQUAKE.

SELECTED SKETCHY SCRIPT $EARTHQUAKE

DONE PROCESSING
SATISFIED REQUESTS:

((<=> ($DATELINE LOC &DLOC MONTH &MON DAY &DAY)))
&DLOC
        CLASS       (#LOCATION)
        LOCALE      (*ITALY*)
        SATISFIED   ((2) (4))
&MON
        NUMBER      (2)
        SATISFIED   (NIL (4))
        CLASS       (#NUMBER)
&DAY
        NUMBER      (4)
        SATISFIED   (NIL NIL)
        CLASS       (#NUMBER)
```

NATURAL LANGUAGE

```
((<=> ($EARTHQUAKE LOC & LOC SEVERITY &RIC)))
&LOC
        CLASS       (#LOCATION)
        LOCALE      (*ITALY*)
        SATISFIED   ((2) (4))
&RIC
        NUMBER      (6.3)
        SATISFIED   ((2) (4))
        CLASS       (#NUMBER)

((ACTOR &DEADGRP TOWARD (*HEALTH* VAL (-10))))
&DEADGRP
        NUMBER      (95)
        SATISFIED   ((2) (4))
        CLASS       (#PERSON)

((ACTOR &HURTGRP TOWARD (*HEALTH*) VAL (-LT10))))
&HURTGRP
        NUMBER      (1000)
        SATISFIED   ((2) (4))
        CLASS       (#PERSON)

CPU TIME = 9.440 SECONDS
```

RUSSIAN SUMMARY:
ZEMLETRYASENIE SREDNEI SILY PROIZOSHLO V ITALII. CILA ZEMLETRYASENIYA OPREDELENA V 6.3 BALLA PO SHKALE RIKHTERA. PRI ZEMLETRYASENII 95 CHELOVEK BYLO UBITO I 1000 RANENO.

SPANISH SUMMARY:
HUBO 95 MUERTOS Y 1000 HERIDOS EN UN TERREMOTO FUERTE EN ITALIA. EL TERREMOTO MIDIO 6.3 EN LA ESCALA RICHTER.

ENGLISH SUMMARY:
95 PEOPLE WERE KILLED AND 1000 WERE INJURED IN A SEVERE EARTHQUAKE THAT STRUCK ITALY. THE QUAKE REGISTERED 6.3 ON THE RICHTER SCALE.

INPUT:

11 - 29 KATHEKANI, KENYA, - AT LEAST 12 PEOPLE WERE REPORTED KILLED EARLY TODAY WHEN AN EXPRESS TRAIN RAN ONTO A FLOODED BRIDGE WHOSE RAILS HAD BEEN SWEPT AWAY, CRASHED THROUGH IT AND PLUNGED INTO A RIVER IN KENYA.

THE OFFICIAL PRESS AGENCY REPORTED THAT THE DEATH TOLL WAS AT LEAST 12 AND THAT 70 WERE INJURED IN WHAT RAILROAD OFFICIALS CALLED THE WORST PASSENGER TRAIN DISASTER IN EAST AFRICAN HISTORY.

```
SELECTED SKETCHY SCRIPT $VEHACCIDENT

DONE PROCESSING
SATISFIED REQUESTS:

((<=> ($DATELINE LOC &DLOC MONTH &MON DAY &DAY)))
&DLOC
        CLASS       (#LOCATION)
        LOCALE      (*KENYA*)
        SATISFIED   ((11) (29))
&MON
        NUMBER      (11)
        SATISFIED   (NIL (29))
        CLASS       (#NUMBER)
&DAY
        NUMBER      (29)
        SATISFIED   (NIL NIL)
        CLASS       (#NUMBER)

((<=> ($VEHACCIDENT VEH &VEH OBJ &OBJ LOC &LOC)))
&VEH
        CLASS       (#PHYSOBJ)
        TYPE        (*VEHICLE*)
        SROLE       (&TRAIN)
        SCRIPT      ($TRAIN)
        SATISFIED   ((11) (29))
&OBJ
        CLASS       (#PHYSOBJ)
        CONENT      (*RIVER*)
        CPRPS       (*WATER*)
        SATISFIED   ((11) (29))
&LOC
        CLASS       (#LOCATION)
        LOCALE      (*KENYA*)
        SATISFIED   ((11) (29))

((ACTOR &DEADGRP TOWARD (*HEALTH* VAL (-10))))
&DEADGRP
        NUMBER      (12)
        SATISFIED   ((11) (29))
        CLASS       (#PERSON)

((ACTOR &HURTGRP TOWARD (*HEALTH* VAL (-LT10))))
&HURTGRP
        NUMBER      (70)
        SATISFIED   ((11) (29))
        CLASS       (#PERSON)
```

NATURAL LANGUAGE

CPU TIME = 9.539 SECONDS

RUSSIAN SUMMARY:
V ZHELEZNODOROZHNOI KATASTROFE V KENII 12 CHELOVEK BYLO UBITO I 70 RANENO.

SPANISH SUMMARY:
HUBO UN ACCIDENTE DE FERROCARRIL EN KENYA QUE RESULTO EN 12 MUERTOS Y 70 HERIDOS.

ENGLISH SUMMARY:
A TRAIN CRASH CLAIMED 12 LIVES AND INJURED 70 IN KENYA.

INPUT:

3 - 4 PISA, ITALY -- OFFICIALS TODAY SEARCHED FOR THE BLACK BOX FLIGHT RECORDER ABOARD AN ITALIAN AIR FORCE TRANSPORT PLANE TO DETERMINE WHY THE CRAFT CRASHED INTO A MOUNTAINSIDE KILLING 44 PERSONS.

THEY SAID THE WEATHER WAS CALM AND CLEAR, EXCEPT FOR SOME GROUND LEVEL FOG, WHEN THE U'S MADE HERCULES C130 TRANSPORT PLANE HIT MT. SERRA MOMENTS AFTER TAKEOFF THURSDAY.

THE PILOT, DESCRIBED AS ONE OF THE COUNTRY'S MOST EXPERIENCED, DID NOT REPORT ANY TROUBLE IN A BRIEF RADIO CONVERSATION BEFORE THE CRASH.

SELECTED SKETCHY SCRIPT $VEHACCIDENT

DONE PROCESSING
SATISFIED REQUESTS:

```
((<=> ($DATELINE LOC &DLOC MONTH &MON DAY &DAY)))
&DLOC
        CLASS       (#LOCATION)
        LOCALE      (*ITALY*)
        SATISFIED   ((3) (4))
&MON
        NUMBER      (3)
        SATISFIED   (NIL (4))
        CLASS       (#NUMBER)
&DAY
        NUMBER      (4)
        SATISFIED   (NIL NIL)
        CLASS       (#NUMBER)

((<=> ($VEHACCIDENT VEH &VEH OBJ &OBJ LOC &LOC)))
```

```
&VEH
        CLASS          (#PHYSOBJ)
        TYPE           (*VEHICLE*)
        SROLE          (&AIRPLANE)
        SCRIPT         ($AIRPLANE)
        SATISFIED      ((3) (4))
&OBJ
        CLASS          (#PHYSOBJ)
        CONENT         (*MOUNTAIN*)
        SATISFIED      ((3) (4))
&LOC
        SATISFIED      ((3) (4))
        CLASS          #LOCATION
        LOCALE         (*ITALY*)

((ACTOR &DEADGRP TOWARD (*HEALTH*) VAL (-10))))
&DEADGRP
        NUMBER         (44)
        SATISFIED      ((3) (4))
        CLASS          (#PERSON)

CPU TIME = 6.778 SECONDS

RUSSIAN SUMMARY:
V AVIATSIONNOI KATASTROFE V ITALII 44 CHELOVEK BYLO
UBITO.

SPANISH SUMMARY:
HUBO 44 MUERTOS CUANDO UN AVION CHOCO CONTRA UN
MONTANA EN ITALIA.

ENGLISH SUMMARY:
44 PEOPLE WERE KILLED WHEN AN AIRPLANE CRASHED INTO A
MOUNTAIN IN ITALY.
```

CONCLUSION

We are now hooking up FRUMP to the United Press International wire service. We intend to produce a system that will know about the interests of the users logged in to it and will provide them with summaries of events that they care about as soon as they happen.

Our intention is to produce a practical working Artificial Intelligence program. We do not see FRUMP as the solution to all the complexities of language understanding. It is certainly not a replacement for SAM in anything except an immediate practical sense. However, it does have some theoretical validity of its own. When people skim they use some but not all of the reading techniques available to them. FRUMP, in a sense, has abstracted out the essence

of SAM. Viewed in that way it is analogous to how skimming abstracts out the essence of reading. That is, FRUMP both works and tests out a theory. We view it as a success.

Acknowledgements

The research described in this paper was supported by the Advanced Projects Agency of the Department of Defence and monitored by the Office of Naval Research under Contract N0014-75-C-1111.

REFERENCES

Becker, J. (1975). The phrasal lexicon. *Proc. Theoretical Issues in Natural Language Processing Workshop*, pp. 70–73. Cambridge, Mass.: Bolt, Beranek and Newman.

Cullingford, R. E. (1977). Organizing World Knowledge for Story Understanding by Computer. Ph.D. thesis. Department of Engineering and Applied Science, Yale University.

Parkinson, R., Colby, M., and Faught, W. (1976). Conversational Language Comprehension Using Integrated Pattern-Matching and Limited Parsing. *Technical Report*, UCLA Psychiatry Department, Los Angles, California.

Rieger, C. J. (1975). Conceptual memory and inference. *Conceptual Information Processing*, pp.157–268 (ed. R. C. Schank). Amsterdam: North Holland Publishing Company.

Rieger, C. K. (1975). Conceptual analysis. In *Conceptual Information Processing*, pp. 83–155 (ed. Schank, R. C.). Amsterdam: North Holland Publishing Company.

Riesbeck, C. K. (1974). Computational Understanding: Analysis of Sentences and Context Ph.D. thesis. Stanford University. Also *AI Memo AIM-238*. Stanford: Artificial Intelligence Laboratory, Stanford University.

Schank, R. C. and Abelson, R. P. (1977). *Scripts, Plans, Goals and Understanding: An Inquiry into Human Knowledge Structures.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Schank, R. C. and Yale, A. I. (1975). Project. SAM – A Story Understander. *Research Report 43*, Computer Science Department, Yale University.

Schank, R. C. (1975). *Conceptual Information Processing.* Amsterdam: North Holland Publishing Company.

Schank, R. C. (1974). Understanding Paragraphs. Technical Report. Castagnola, Switzerland: Instituto per gli studi Semantici e Cognitivi.

Schank, R. C. and Tesler, L. (1969). A conceptual parser for natural language. *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington D.C., pp. 569–578 (eds. D. E. Walker and L. M. Norton). Bedford, Mass.: The Mitre Corporation.

Schank, R. C. (1969). A Conceptual Dependency Representation for a Computer-oriented Semantics. Ph.D. thesis. University of Texas.

Schank, R. C. and Rieger, C. K. (1974). Inference and the computer understanding of natural language. *Artificial Intelligence*, Vol. 5, pp. 373–412.