DTIC FILE COPY

(4)

AD-A202 300

Technical Report 1243
July 1988

Knowledge Acquisition Tools and Knowledge Representation Strategies Development for a Naval Expert System

Lessons Learned

C. D. Haupt

DTIC
ELECTE
13 JAN 1989
S D
E

Approved for public release; distribution is unlimited.

89 1 12 040

NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152-5000

E. G. SCHWEIZER, CAPT, USN
Commander

R. M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

This report was prepared by Code 444 of Naval Ocean Systems Center.

Released by
D. C. Eddington, Head
Artificial Intelligence
Technology Branch

Under authority of
W. T. Rasmussen, Head
Advanced C^2 Technologies
Division

JJ

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NOSC TR 1243	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Ocean Systems Center 6b. OFFICE SYMBOL (if applicable) NOSC	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) San Diego, CA 92152-5000	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Joint Tactical Program Management Office 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code) 1500 Planning Research Drive McLean, VA 22102	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	AGENCY ACCESSION NO.
63745A	CD70	RDA	DN307 493

11. TITLE (include Security Classification)
KNOWLEDGE ACQUISITION TOOLS AND KNOWLEDGE REPRESENTATION STRATEGIES DEVELOPMENT FOR A NAVAL EXPERT SYSTEM
Lessons Learned

12. PERSONAL AUTHOR(S)
C. D. Haupt

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Mar 1986 TO Nov 1986	July 1988	29

16. SUPPLEMENTARY NOTATION

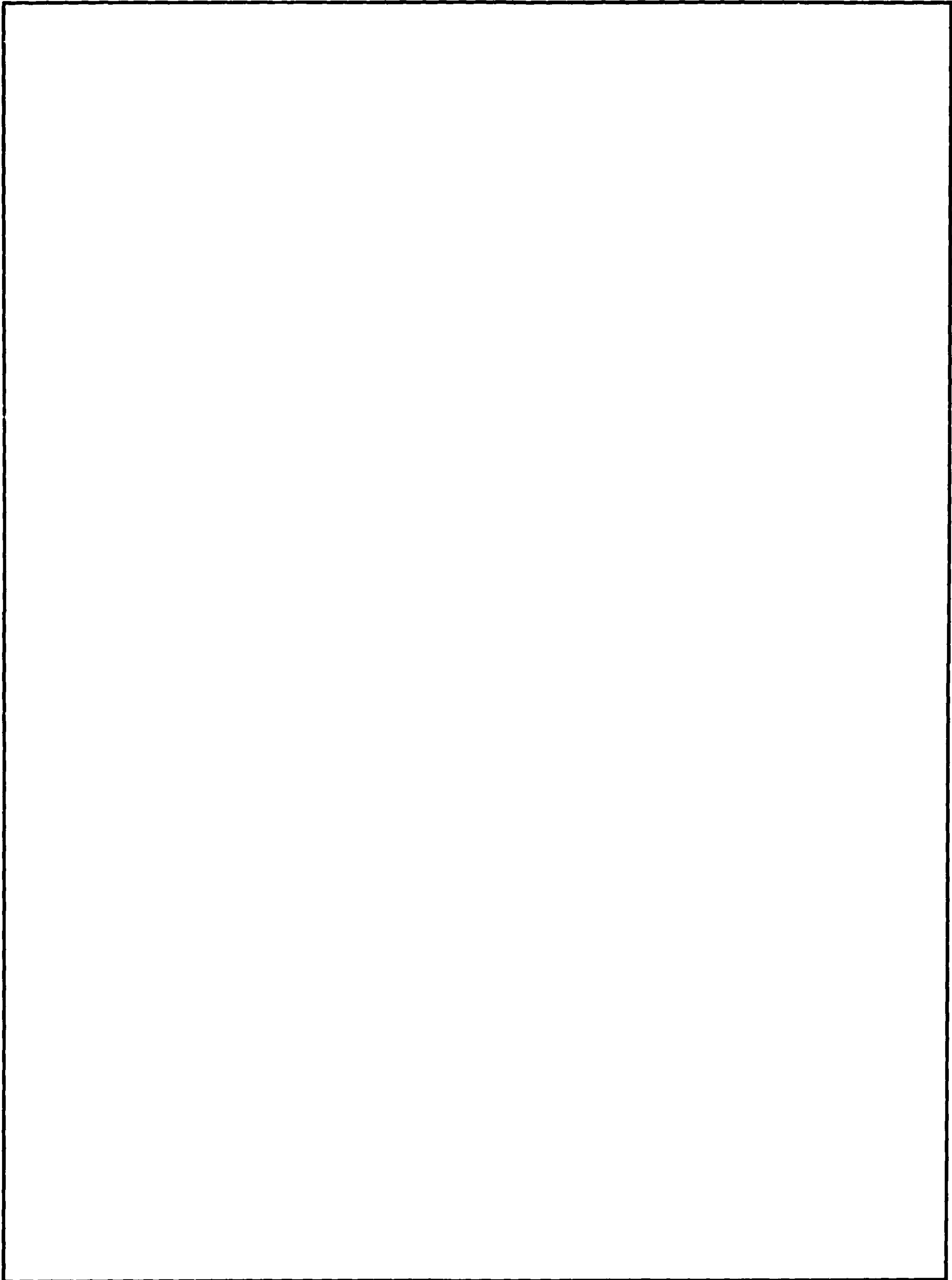
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
			artificial intelligence

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

Research performed on the Command Action Team (CAT) system and the Smart Knowledge Acquisition Tool (SKAT) is described. CAT is a rule-based expert system to perform threat assessment in the naval domain; SKAT is an automated knowledge acquisition tool for CAT. These systems were developed jointly by Naval Ocean Systems Center (NOSC) and Carnegie Mellon University (CMU). The research explored an alternative knowledge representation scheme for CAT that reduced the number of working memory elements. A knowledge representation mechanism was developed to investigate the feasibility of incorporating higher-level control on rule-firings based on the idea of an expert's problem-solving method. A system is described for automatically generating OPS83 rules from an external database, and an analysis is performed describing SKAT's ability to generate the CAT system.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE PERSON C. D. Haupt	22b. TELEPHONE (include Area Code) (619) 553-5302 22c. OFFICE SYMBOL Code 444

DD FORM 1473, 84 JAN



DD FORM 1473, 84 JAN

CONTENTS

INTRODUCTION 1

BACKGROUND 1

GROUPING CAT'S ASSERTIONS BY OBJECT 2
- Objective of Research 2
- Approach 3
- Results 5
- Possible Future Research 8

SKAT-GENERATED DATABASE RULES 9
- Objective of Research 9
- Approach 9
- Results 12
- Possible Future Research 13

AN ANALYSIS OF SKAT'S USEFULNESS FOR NOSC 13
- Objective of Research 13
- Approach 13
- Results 19
- Possible Future Research 22

Accession For
NTIS GRA&I ☒
DTIC TAB ☐
Unannounced ☐
Justification

By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

INTRODUCTION

This report describes a major portion of the work performed by the author while on a developmental training program (DTP) at Carnegie Mellon University (CMU) from March 1986 to November 1986. The research concentrated on the Command Action Team (CAT) program and the Smart Knowledge Acquisition Tool (SKAT) developed by Naval Ocean Systems Center (NOSC) and CMU.

The CAT project at CMU consists of two major portions: the CAT rule-based expert system and the Smart Knowledge Acquisition Tool (SKAT). The CAT system is designed to perform threat assessment for a battlegroup flag watch officer. The system is written in the rule-based system shell language called OPS83. SKAT is the knowledge acquisition tool designed for the CAT system. SKAT was designed to elicit tactical knowledge from an expert flag watch officer, to store that knowledge in an intelligent manner on the computer, to integrate that knowledge into the existing CAT knowledge base, and to translate that knowledge into a useful form in the target language of OPS83.

Three main areas of research are described here. The first area was the development and analysis of a version of CAT that stores assertions in the OBJECT working memory element. The second area was the development of SKAT code to provide database rules for CAT from information contained in the SKAT ontology. The third area of research consisted of a study of SKAT to determine its usefulness for the CAT system residing at NOSC.

BACKGROUND

The Command Action Team (CAT) is a system developed by Naval Ocean Systems Center Code 444 through interaction with researchers at Carnegie Mellon University (CMU) and the operational personnel of Command Carrier Group 3 (CCG 3). CAT was successfully deployed on the USS *Carl Vinson* during its deployment in 1986. CAT is an expert system designed to support the flag watch officer by continually monitoring real-time reports of enemy activity, capability assessments, and intentions. In particular, CAT provides an assessment of the ability of the threat to target the carrier. This tactical situation (TACSIT) assessment considers, among other things, factors such as the ability of the enemy to coordinate with the Soviet Ocean Surveillance System (SOSS), own-force Emission Control (EMCON), national Defense Condition (DEFCON), and mission of the possible detecting platforms. In addition, CAT allows the user to define alert/warning conditions of interest within the particular operating environment.

GROUPING CAT'S ASSERTIONS BY OBJECT

OBJECTIVE OF RESEARCH

The main reason for the training at CMU was to gain knowledge and experience with methods of acquiring knowledge for a rule-based expert system. In order to provide the necessary framework for understanding the problems and concerns associated with building a knowledge acquisition tool for a rule-based expert system it was important to understand the issues surrounding the building of that system. For this reason one of the main areas of research was on the CAT system itself.

The objective was to explore the benefits and drawbacks associated with making the CAT system more object oriented. The primary focus of the system up until this point had been at the assertion level. This research would examine how performance of the system is affected by this representational change. The research would also describe any representational implications of such a change.

One of the main motivations for trying this experiment, besides to explore the representational ramifications, was to see if the performance of the system could be enhanced. Under the OPS philosophy, the original version of CAT is taxing on the matching process due to the large amount of ASSERTION working memory elements that are created during execution of the system. Putting the assertions into an OBJECT element could reduce the amount of overhead time that OPS spends matching because the first thing the OPS matcher matches on the left-hand side is the name of the element. When a rule is checking several attributes for a specific object, and assertions are used, another check has to be performed to be sure all the assertions are referring to the same object. Such matching can result in a large cross-product of interassertion matches that have to be performed.

This research resulted in the development of a prototype system that used the principle that all assertions about a single object should be stored within a single working memory element. This prototype system, called "OBJCAT," also included the implementation of a somewhat independent mechanism which divided the program into a set of problem-solving steps including report pruning, report elaboration, tactical inferencing, and report propagation.

APPROACH

Knowledge Representation

The principal data structure in both CAT and OBJCAT is a network of dependencies among assertions (or propositions). In OBJCAT, this network is represented by working memory elements of type OBJECT and REASON. The OBJECT element contains an array of assertions, each bearing information for the attribute name, confidence factor, observation time, value, and occupant, if any, for the role of the patient (now, somewhat confusingly, called "object" as in the original CAT). Slots in the array are not reserved for particular attributes; assertions are assigned to array elements as the former attributes arrive. There is a set of access functions which extract values, assign values, and test for matches using the OBJECT elements.

REASON elements specify the sources of support for an assertion, and each assertion is associated with at least one unique REASON element. The association is performed by matching. The REASON element contains the name of the attribute and of the object that identifies the assertion. The REASON element contains an array of object/attribute pairs which specify the antecedent conditions for inferred assertions. Additional information includes the portion of the confidence factor attributable to the source of support and some control information.

Other working memory elements used to hold tactical data include a REPORT element which holds information being delivered to OBJCAT, and a NEW_ASSERTION element which holds information from a report being processed or from a newly performed inference.

Control

Contexts in OBJCAT are of two kinds: steps in a problem-solving method and procedure calls. The problem-solving method is the set of high level steps performed by the expert system (and the expert as well). A step within the method contains a set of rules that have no conflict resolution strategy (other than arbitrary selection of an instantiated rule) to decide which rules to fire. Rules within that step will fire regardless of the status or of subsequent firings of other rules within that step. That is, there is no control within a step of the problem-solving method. If the programmer is trying to instill some control in a step, this implies that another step, or substep should be created.

There is a working memory element called METHOD which contains an array of symbols which designate steps in the OBJCAT problem-solving method. One of these symbols is extracted from the array and stored in a constant location designating which step is currently active. Every rule in OBJCAT which is not in a procedure has a test for the METHOD element as its first condition. One distinguished rule does not test for the current step: its purpose is to advance from an exhausted step to the next step.

Control within the "integrate" step (for propagating the effects of updates) is effected by means of a working memory element of type NEW. NEW contains an object and attribute designator, making it refer to a single assertion. When this

assertion is first inserted into the OBJECT element, and each time this assertion is modified, a NEW element is created for the assertion. These elements are tested in all tactical inference rules in such a way that the rule will be instantiated only when there is a NEW element corresponding to at least one of the elements matched in the rule's conditions. These elements are removed when no tactical inference rules remain instantiated.

The Problem-Solving Method

The problem-solving method determined for OBJCAT was

1. Select a report.
2. Elaborate on the report.
3. Make resulting tactical inferences.
4. Integrate resulting conclusions into inference network.

Step Select_Report

As a simulation of report acquisition, a rule in this step selects the oldest REPORT element, transfers its information to working memory elements of type NEW_ASSERTION and REASON, and then discards the REPORT element. If there are no reports, this step is exited as soon as it is entered. Thus, when the reports are exhausted, the system will loop through the steps forever.

Step Elaborate_Report

Rules in this step employ information from the new report, stored in a NEW_ASSERTION element, and in the inference network, to generate new reports. An example is a position report, combined with the former position from the inference net, yields a speed report. Handling this kind of elaboration at the report level may solve some of the problems of control when trajectory-plotting occurs as tactical inference rules.

Step Infer

Tactical inference rules fire only when the step is "infer." These rules create NEW_ASSERTION and REASON elements. At least one of the elements which match the conditions must have an associated NEW element.

Step Integrate

NEW_ASSERTION elements, whether arising from reports or from inferences, are integrated into the inference net in this step. The assertion can be inserted into the OBJECT element for the first time, can replace an existing assertion, can be fused (by evidence combination) into an existing assertion, or can be discarded. Which action is taken is domain dependent, and is sensitive to the existing inference net and to the nature of the attribute.

RESULTS

The system described above was implemented and tested. SKAT was modified to write the 50 tactical inference rules in the system, as well as the explanation rules. The rest of the system was translated by hand and it involved changing every rule in the CAT system into the new representation. The other sets of rules (which included inference net maintenance, and the rules to perform the explanation algorithm) were written by hand because they are so idiosyncratic it would be unreasonable to have SKAT generate them.

Knowledge Representation Results

There were several problems that were identified with this representation. Most of them were a result of deficiencies in OPS83 and not necessarily a generic problem associated with the object-based representation. The five major representational problems identified in OBJCAT are described in the following sections

- The lack of a general matching mechanism.
- The inability to reference previously matched assertions.
- The difficulties of storing multiple valid values for certain types of attributes.
- The inability to store multiple competing values for an attribute.
- The excessive iteration through assertion arrays.

General Matching Problem

In CAT each assertion about an object is stored as separate working memory elements (wme), so the normal OPS matcher finds all of the appropriate matches. In OBJCAT, assertions about an object are all stored in arrays within an OBJECT wme. Normal means of interassertion matching using the OPS matcher are not sufficient in this representation for finding all of the assertions that match the appropriate constraints. This matching is because OBJCAT stores the three most current assertions for each attribute of an object in an array. Thus there is an array for each type of assertion, and there are potentially three assertions that could satisfy a constraint on an object's attribute. The problem is only one of those three satisfied assertions can be used for any one instantiation. Once the first instantiation has been made, somehow the functions doing the matching need to know not to use that assertion again. Instead they should use one of the other two assertions.

The current matching functions in OBJCAT handle this problem by looking only at the most current assertion of each type. It is as if in the original version of CAT a flag is put in the assertion wme that states whether or not that assertion is the most current of that type. Then all of the rules only consider an assertion if this flag is set to "most recent." This method ignores the possible valid cross-matching that could occur.

Referencing Problem

The referencing problem is a theoretical problem; it assumes that the matching problem described above has been solved. The referencing problem has to

do with knowing which assertions matched in previous condition elements. For example, consider the case in which the first condition element of a rule is matching on speed assertions with observation_time less than 100. The second condition element at some point wishes to check the value of any speed assertions that matched in the first condition. There is no way for the second condition element to know which assertions matched in the first condition unless the second is somehow passed a pointer to the matched elements, or is able to perform all of the same tests itself. OPS83 does not allow for variables (either local or global) to be bound or accessed on the left-hand side of a rule, thus precluding one condition element passing the bindings found to another condition element further down on the left-hand side.

Multiple Valid Value Storage Problem

The fact that only three assertions are stored for each attribute of an object causes a problem in the case of assertions that can have many valid values. An example of such an attribute is "has_missile." A object can have many possible missiles. Thus only keeping three has_missile assertions around results in throwing away potentially good information.

One alternative would be to match assertions by attribute and object instead of just attribute as it is done now. This way the has_missile assertions could store the missiles in the object field, and then each has_missile assertion with a different missile would be assigned a set of three assertions. Since has_missile is static information, most likely, there would only be one assertion for each missile laying around. Thus this representation would be wasteful with storage space. In addition, this representation would be afflicted with a version of the general match and referencing problems similar to those described above.

Multiple Competing Values

There are attributes in CAT for which there can be only one valid value. In the old version of CAT, however, it was legal for such attributes to have several assertions with different values. An example of this would be to have two assertions in the system for the nationality of a platform. If one assertion claims that the nationality of Track_1 is US and the other assertion claims it is UR then this is a case in which there are two competing values. Only one of the assertions can be correct because a platform has only one nationality. Often CAT is unable to discern immediately which of the two assertions is the correct one. Both assertions are allowed to remain in the system until it can be shown that one or both should be removed. This could occur through the intercession of a rule which knows how to decide between the two based on other knowledge in the system, the level of the confidence factors, or other pertinent information. The two assertions allow rules that use their information to fire and create higher inference chains. Thus, by leaving in the conflicting assertions, CAT is allowed to look at the possibilities that the assertions present. Once it is determined that one of them should be deleted, CAT can clean up its inference network by removing all of the assertions that resulted from the use of the erroneous assertion.

In OBJCAT, this mechanism for employing multiple competing values is precluded by the storing of assertions in the array. Since three assertions are stored

for an attribute, it is possible to have the assertions around. Yet, because of the general matching problem described above, it is currently impossible to match in a general way on all of the assertions. Consequently, the corresponding inference chains for the competing assertions cannot be created.

Excessive Iteration Problem

This section describes the inefficiencies resulting from having matching functions that have to search through an array of values. In OBJCAT, each constraint on a value requires a function call to determine if the constraint holds. These functions require several arguments:

- The object for which these assertions should be checked.
- The attribute of interest (i.e., "predicate").
- The value field to be checked.
- The relational operator with which to check.
- The slot of the assertion to be checked.

For example, to determine if &object_a has a "speed" assertion such that the value of the speed is greater than 20 knots, the function call would look like this

```
match_object_and_number(&object_a, speed, 1 |>|, 20).
```

(When SKAT generates this code, it realizes that the value of speed is stored in the first value slot of that type of assertion. That is the reason for the one in the third parameter position.) This function then looks through the array of histories for &object_a to find the attribute "speed." If SKAT finds the "speed," then it returns the value of the test to determine if field one of that assertion is greater than 20.

The problem becomes apparent when one compares the way the old version of CAT would do a simple test with the way that OBJCAT would have to perform that same test. Take the simple case of checking for one of a set of possible values, as in "infer_frigate_if_within." A simple atomic check of "(value.name = CV \/ value.name = CVN)" gets transformed in OBJCAT into two function calls:

```
(match_object_and_symbol(&obj_1, type, 1, |=|, CV) \/
 match_object_and_symbol(&obj_1, type, 1, |=|, CVN)).
```

Because this is searching down the array of assertions, this indicates that the time to do matching for condition elements will be greater in OBJCAT than in the old version of CAT.

Performance Results

OBJCAT was tested on a scenario on which CAT had previously been tested. In that scenario, 33 of CAT's 50 tactical inference rules fired as compared with 30 in OBJCAT. The three rules that never fired in OBJCAT were rules that formed the inferences that are highest in the inference network. The reason they did not fire was a result of problems in rules that integrate new assertions into the inference net. This result is also due to a minor representation problem of having multiple instantiations

of the same rule firing due to multiple NEW working memory elements being created on previous cycles of the problem-solving method. Those problems could be fixed, but testing of OBJCAT was ended before they were.

The behavioral characteristics of OBJCAT were unlike those of CAT, but that was mainly due to the three tactical inference rules that did not fire. With those three rules firing, some of the conclusions that were made in CAT would also have been made in OBJCAT. It was inconclusive as to how similar the two systems would have been ultimately had OBJCAT been completely debugged and working, but because of the manifestation of the representation problems described above, it was apparent that only a subset of the conclusions in CAT would have been made in OBJCAT. The percentage of conclusions that would not be made in OBJCAT that would have in CAT is not known.

As far as time and memory characteristics are concerned, however, OBJCAT fared better than CAT. Although the number of rule firings in OBJCAT was twice that of CAT, the actual user time to run the scenario was about 40 percent less. The representation had a dramatic impact on the size of working memory, reducing the mean number of working memory elements by 19 times. The mean sizes of the conflict set, conflict resolution time, and rule fire time were all significantly reduced.

POSSIBLE FUTURE RESEARCH

There are several areas in which this research could be extended. One of the areas would be to try and minimize the cost resulting from the matching functions. This could be done by changing the attribute access functions from searching through an array to using a hashing function. Another way would be to have the object contain a large list of all the attribute names that are of interest and slots to store the values of those attributes. This would provide less flexibility and generality than the present design which allows for any attributes to be used without changing the representation of the object. The cost to access the attributes could be less, however, than the current method of stepping through a large array and looking for the appropriate attribute.

Another area for research would be to explore the idea of a hybrid representation. The original version of CAT uses the ASSERTION element as the basic representational unit. OBJCAT uses the OBJECT element as the basic unit. There are several attributes that change frequently in CAT (such as position, speed, course, etc.). It would be interesting to put these more expensive attributes into the OBJECT element and to put the higher level conclusions that do not get made as often into ASSERTION elements. Storing some attributes at the OBJECT level would reduce some of the interassertion matching liability, whereas storing other attributes in ASSERTIONS would still allow the flexibility that representation provides. The difficulty in implementing such a system, even with SKAT, however, would be significant as there could not be a general way to deal with network maintenance, confidence factor manipulations, explanation, as well as the other pieces of functionality that belong in the system.

A third area in which follow-on work should be done is to see how much of the performance improvements were caused by the object-based representation and how much was a result of the implementation of the problem-solving method. The working

memory reduction is a result of the storing of the assertions in objects; there are just fewer little pieces of information independently lying around. They are bundled up in the OBJECT element. It is not so easy, however, to credit the decreases in conflict set size, conflict resolution time, and rule firing time.

SKAT-GENERATED DATABASE RULES

OBJECTIVE OF RESEARCH

SKAT has stored a lot of information on ships and weapons that was not being used in CAT. One objective was to provide CAT with the capability of using that information.

APPROACH

The Ontology

SKAT's declarative knowledge about domain concepts and their relationships is stored in a Common Lisp data structure called the *ontology*. The ontology resembles a sematic network. The information is stored in the ontology in a very general form since it is used for numerous different purposes.

The ontology contains information on objects that are important for the domain. Those objects may be higher level concepts or groupings, like *fleet* or *aircraft_carrier*, or they may be specific instances of real world objects, like the carrier *kiev* or *Carl Vinson*. The information in the ontology consists of two types:

1. Characteristics of platforms and military installations
2. Characteristics of weapons and sensors

The first type is concerned with the characteristics of platforms and military installations. Ships, aircraft, subsurface vessels, land bases, and satellites all fall under that category. Characteristics such as the weapons and sensors a platform owns, the maximum speed of the platform, and other pertinent data, are all stored in the ontology.

The second type of information in the ontology is concerned with the characteristics of weapons and sensors that can be owned and used by the platforms and military installations. This type would include information on missiles, guns, radar, sonar, torpedoes, electronic countermeasures (ecm) gear, and would include characteristics that are pertinent to each type of weapon and sensor.

All of this information was retrieved from *Jane's Fighting Ships*, so it is unclassified.

Static Database Rules – Initial Design

The initial design of the static database rules centered on the division of the static database information into the two categories described above. Essentially two types of rules were envisioned.

1. Platform type rule.
2. Weapon or sensor type rule.

Platform Type Rule

This type rule included surface and subsurface platforms, aircraft, satellites, and landbases. This rule consisted mainly of a left-hand side that would check for the existence of an assertion with predicate equal to "name" and with value equal to the name of the platform for which that rule was made. Thus, there would be a rule in the system for every platform that was named in the ontology. The right-hand side would consist of a set of assertions being made about the characteristics of the platform. In other words, as soon as a new platform is discovered by the CAT system, all of the static data about that platform is put into the system. In this way, other rules that may depend on this data for matching on the left-hand side will have the data present only when the object is in the system. Essentially working memory does not know about an object and its characteristics unless that object is in the system.

Consider an example database rule of this type. If SHIP_A is known to own MISSILE_1 and RADAR_1, and to have a maximum speed of 30 knots, then an example rule for that platform would look like this:

If there is an assertion for an object in
working memory with
predicate = "name" and
value = "SHIP_A"
then make the following assertions
The object has_missile with name = "MISSILE_1."
The object has_radar with name = "RADAR_1."
The object has maximum speed = 30.

Thus, if there is an assertion that TRACK_1 has a "name" assertion with value SHIP_A (i.e., TRACK_1 is the ship SHIP_A), then the three assertions listed on the right-hand side of the above rule would be made with the subject field equal to "TRACK_1."

Weapon/Sensor Type Rule

The weapon or sensor type rule included missiles, guns, radar, sonar, torpedoes, and electronic countermeasures (ecm) gear. Instead of checking for "name" type assertions, as was done for the platform type rules, the weapon/sensor type rules consisted of a left-hand side that would check for the existence of an assertion with an ownership type predicate and with value equal to the name of the weapon/sensor type for which that rule was made. As with the platform rules, the right-hand side of each of the weapon/sensor rules would consist of a set of assertions being made about the

characteristics of the sensor/weapon for which that rule was made. An ownership type predicate is one that declares that an object owns a certain piece of equipment.

Inheritance of Database Information

The ontology not only has static database information, but also contains links showing the subclass and superclass relationships between the nodes in the ontology. For example, it would represent the fact that the carrier *Carl Vinson* is a member of the Nimitz class, as well as the fact that the Nimitz class is a member of the set of surface combatants. This knowledge of the hierarchical nature of the ontology, as well as the fact that static data is stored not only at the "name" level, but at higher levels of abstraction (such as "class," and "type") provided a means for asserting more information on the right-hand side than that provided by asserting only the information at the "name" node. Thus, if the name of a ship is known, all of the static data that is known for that ship, as well as the data that can be inherited from ancestor nodes of that ship in the ontology, is asserted for that object.

Problem With Initial Design

The main problem with the initial design had to do with the level at which information is stored in the ontology for platforms as well as the number of individual platforms. There is a large number of platforms that SKAT knows about. Writing a rule for each individual platform would require an enormous amount of memory and is unfeasible. In addition, most of the static information about a platform is determined by the class of that object. The initial design did not take advantage of that fact to reduce the number of static database rules that would have to be created. This feature was addressed in the subsequent redesign of the system.

Static Database Rules - Subsequent Design

The main goal of the subsequent design was to take advantage of the fact that most of the features of a platform are determined by the class of that object. In the ontology, with only a few exceptions, the platforms within a class all possessed the same weapons and sensors as well as identical values for the other physical attributes of the object. In fact, the data was stored at the class level in the ontology for the platforms. Individual platforms that possessed more equipment than was shown at the class level had this extra information stored at the individual node in the ontology, with the rest of the attributes being stored at the class level. No provision was set up in the ontology to handle exceptions at the subclass level, but such problems were never encountered anyway.

The subsequent design was geared towards taking advantage of this fact that most of the static information concerning platforms is stored at the class level. This design resulted in four types of static database rules:

1. Platform name to platform class type rule.
2. Individual platform database type rule.
3. Platform class database type rule.
4. Weapon or sensor type rule.

The weapon/sensor type rule was described in a previous section.

Platform Class Database Type Rule

One type of rule that resulted from the design is a rule that asserts the information associated with platforms at the class level. A rule of this type has a left-hand side that checks for the existence of an assertion in working memory with the attribute "class." Thus, if the class of a ship is known, all of the static data that is known for that class, as well as the data that can be inherited from ancestor nodes of that class are asserted for that ship.

Individual Platform Database Type Rule

Some platforms have attributes that are specific to just that one platform and thus cannot be inherited from the class node for that platform or at a higher level in the inheritance network. For such cases, a rule is required that is similar to the rule described in a previous section. The only difference is that the right-hand side of the new rule would assert only the data found at the "name" level, and not any of the inherited information. This rule would also assert the class of the platform so that the platform class database rule described above will be instantiated and the inherited information for that platform will be asserted into working memory.

Platform Name to Platform Class Type Rule

Most of the information for a platform is stored in the platform class database rule, which is instantiated by a "class" assertion for a platform. Thus, if the name of the platform is known, there needs to be some rule that maps the name into the class. For objects that have information stored at the "name" level the name-to-class mapping is done in the same rule that asserts the static information, as described above. For platforms that do not have static information stored at the "name" level, however, the rule needs to be created that asserts the platform class based on the platform name. There does not have to be one of these rules for each platform, however. Since a class normally contains several different platforms, only one rule needs to be created for all of the platforms with that class. The left-hand side of this rule would merely check for the existence of a "name" assertion whose value is the name of one of the members of that class. If it is found, then the right-hand side makes the assertion that that platform has the class defined in the ontology for that object.

RESULTS

The second design described was implemented. Instead of writing the rules directly in OPS83, the database rules were written in the SKAT command language. This was done in order to take advantage of the fact that SKAT is essentially a smart rule editor for CAT rules. SKAT understands what assertions in CAT are and how rules need to be written to support the inference network. When new knowledge representations are being experimented with, SKAT is modified to write the rules in the correct manner to support that new representation. If the static database rules were written directly into OPS83, then every time the knowledge representation changed, the static database rule writer would have to be modified along with SKAT. Since the static database rules are being written in SKAT's command language format, they only have to be written once. This language can then be fed into SKAT whenever the representation is changed and SKAT (when it has been modified for that representation) will write the rules in the appropriate OPS format for that representation.

POSSIBLE FUTURE RESEARCH

One area of possible extension to this work is in the area of making CAT a little smarter about what information it asserts for an object. Working memory is at a premium. The number of assertions that are in the system has a dramatic impact on performance. If the number of static database assertions could be limited to only those that are actually used, this could lessen the impact of loading all of the static data into working memory. There would have to be some rule analysis tools written for SKAT, which could go through the CAT rules and determine what information in the ontology could actually be used by the tactical rules in the system.

Another area for research would be to explore alternate methods of representing the static information. Instead of using assertions, as is done now, alternate representations that group the data into one or several blocks might incur less impact on the inference network and working memory.

It would be useful to change the level at which the static database information is stored. Currently, every platform that comes in causes the class database rule to fire and assert the static information about that class for that object. Thus, two ships that are the same class have the same information asserted for each. An economical approach would be to assert the class information once per class. The left-hand sides of the tactical rules that use that information would have to be made "smarter" to take that into account, but SKAT should be able to do the necessary modifications to allow for that representation.

AN ANALYSIS OF SKAT'S USEFULNESS FOR NOSC

OBJECTIVE OF RESEARCH

This portion of the report will discuss the Smart Knowledge Acquisition Tool (SKAT) that was under development at Carnegie Mellon University (CMU) in the summer of 1986. SKAT is a knowledge acquisition system that was developed for the Command Action Team (CAT) expert system project. This section will describe the structure and function of SKAT. This portion will report on the analysis that was done to determine to what extent SKAT was capable of generating the rules in NOSC's version of CAT. A discussion will follow regarding the extent that SKAT currently is useful for the development of CAT at NOSC, to what extent SKAT could be useful in the near-term with some modifications, and the direction of the research at CMU for SKAT that would impact usefulness at NOSC in the long-term.

Since the SKAT system and the CAT system are constantly being changed and developed, this report reflects the systems that were running up through October 1986.

APPROACH

This section will first describe the SKAT system developed at CMU. Then a description of the CAT system that was analyzed. Because the CAT system at NOSC differed from CMU's version, this description of the CAT system that was used will provide some framework for the analysis.

SKAT

SKAT is a system that understands to a certain extent the template of a tactical inference rule of the CAT expert system. The template that SKAT follows for the tactical inference rules developed at CMU can be described in English in the following manner:

> *A tactical inference rule consists of a left-hand side that matches on the key assertions about objects in the system, checks the appropriate interassertion constraints, and if all the conditions and constraints are satisfied, the rule creates a new assertion that is the conclusion that the rule reaches. The rule also creates the data structures that are necessary to support the function of the inference network.*

In CAT, assertions are essentially the informational primitives. Assertions are the basic pieces of information that the system handles in making inferences. Assertions consist of a "predicate" which is the name of the type of assertion being made (e.g., "has_missile," "speed," etc.). and slots that specify the value of that assertion. Each predicate has a specific set of slots that it needs to have filled in order to specify a value.

Thus, based on this information about assertions, and based on the template described above, SKAT needs to understand several things:

- Assertions and how they are put together.
- Predicates and the slots they contain.
- The legal values for the different slots of each predicate.
- Interassertion comparisons.
- The data structures that the inference net requires in order to maintain a complete and consistent picture.

The knowledge that SKAT has about the rules is spread out over several different parts of the system. These different components, which will be discussed below, include

- The Ontology
- The User Interface Command Language
- The Tactical Inference Rule Generator
- The Assertion Paraphrase Rule Generator.

Ontology

SKAT's declarative knowledge about domain concepts and their relationships is stored in a Common Lisp data structure called the "ontology" which resembles a semantic network. The information is stored in the ontology in a very general form since it's used for numerous different purposes.

The ontology contains information on objects that are important for the domain. Those objects may be higher level concepts or groupings, like "fleet" or "aircraft_carrier," or they may be specific instances of real world objects, like the carrier "kiev" or "Carl_Vinson." Residing with each object node is information about that node. This information includes attribute-value pairs describing different characteristics of the object, as well as information about the supersets and subsets of each node.

The ontology also contains information about the assertions that are possible in the CAT system. The ontology stores information on all of the predicates that CAT employs. Stored with the predicate are the slots for that predicate and the type of values the slots expect. Also stored with each predicate is an English paraphrase of that predicate.

The User Interface Command Language - SCL

The user interface for SKAT is a set of commands that serve to specify different portions of the tactical inference rule. For ease of reference this command language will be called SCL (for Skat Command Language). There are three modes in which the user can work:

- Edit
- Generate
- Teach.

The "edit" mode is currently unimplemented. Upon implementation the mode would allow the user to edit a rule that was previously entered into SKAT. The generate mode is currently a command contained within the "teach" mode. The user may specify that the rule that is being constructed should be generated into the target language and placed into the named file. Using the "teach" mode is the means by which new tactical inference rules are created. The major commands that are used in "teach" mode are

- Constrain
- Conclude
- Refine
- Write
- Literal.

There are other commands that deal with confidence factors, observation times of assertions, histories of commands, and others. Those listed above are important in that they are the tools that the user employs to create the bulk of the rule, in other words, to fill in the variables of the template.

The command *constrain* is used to specify left-hand-side condition elements. The first argument to "constrain" is a reference name for that condition (for example "speed3," "lat_lon-5") or the atom "not." If the word "not" is present this condition is a check in the target language for the nonexistence in working memory of the pattern that is specified for the condition. In OPS83 terminology this is defining a negative condition element. If the "not" is present then the second argument is the reference name. SKAT is able to determine the predicate of the constraint by stripping the number at the end of the reference name. In summary "constrain" is used to check for the existence or nonexistence of assertions with the predicate specified by the reference name.

The command *conclude* is used to specify the right-hand-side conclusion that is to be drawn by this rule. It takes one argument that is the reference name for the assertion. The reference name performs the same function as in the "constrain" command.

The command *refine* is used to further specify the slots of an assertion. This command can be used with both the "constrain" and "conclude" command. The first argument is the reference name for the assertion that is to be refined. The second argument is the name of the slot field to be specified. In the case of a refine statement on a "constrain" predicate, the subsequent arguments are the values against which that field of that predicate are to be tested. In the case of a refine statement on a "conclude" predicate, the subsequent argument is the value that is to be placed in that field of that predicate.

The command *write* prompts SKAT to generate the rule into the target language and place the generated code into a file.

The command *literal* is the escape hatch. *Literal* basically allows the user to put text "as is" into the rule in the event that none of the other SCL commands provide the needed function.

SKAT also allows the user to create and reference variable bindings. These can then be used by condition elements to do intercondition-element comparisons, or by the conclusion to access values contained within assertions that have been matched on the left-hand side.

SKAT provides the capability of reading the input from a file that contains SCL command. Thus the user can prepare the rules in a file and then run them through SKAT. SKAT reads the commands and responds just as if the user was typing at the keyboard, the only difference being the user does not regain control until the whole file has been processed.

The Tactical Inference Rule Generator

The third major component of the SKAT system is the tactical inference rule generator. This portion of SKAT knows how to use the information specified by the user in the SCL to generate a rule in the syntax of the target system, which in the case of CAT is OPS83. This portion of SKAT knows how the major portions of the rule template should look and also knows where to insert the specific information that has been declared by the user for that rule.

The Assertion Paraphrase Rule Generator

SKAT also is able to generate another set of rules. The explanation procedure in CAT employs a set of rules that know how to describe an assertion in a manner that is more English-like than merely displaying the fields of the assertion data structure. These rules use the fact that, knowing the predicate of the assertion, one knows the fields in which the data for that assertion is stored. Since these rules all have a standard template and the ontology stores all of this knowledge about the predicates, there is a section of SKAT that is able to generate the assertion paraphrase rules.

The CAT System

In order to discuss the usefulness of SKAT for NOSC's CAT system, it is necessary to pin-point exactly which system at NOSC was employed in that experiment. The system that was analyzed in this context is the sanitized version of CAT produced by NOSC in July 1986.

CMU has created its own version of CAT that is different from the version at NOSC. Most of the tactical inference rules at CMU are different from those that are contained in NOSC's system. One of the main reasons for this is CMU's inability to have access to the classified information in NOSC's system. The tactical inference rules at CMU have been written with a certain template in mind and this is the template that SKAT employs. Consequently all of the tactical rules at CMU are capable of and are currently being generated by SKAT.

In July of 1986, a sanitized version of the CAT system at NOSC was made so that CMU would at least have access to the structure of the NOSC system if not the actual content. This sanitized version has been analyzed to determine to what extent SKAT is able to generate this version. The sanitized version was chosen for comparison because this version was representative of the system that was being used at NOSC.

The following is a breakdown of CAT. The first column contains the general category into which the rules can be placed. The second column is the number of rules in that category. The third column is the name of the module or file in which the rule is contained.

TYPE OF RULE	#	CONTAINED IN MODULES
database rules	879	(db files, catdb)
database explanation	760	(db files)
other explanation	256	(catpara, catpara 2, catparr, catexp, other files)
alert processing	96	(catalert, catalert2, catalt)
miscellaneous inference	90	(catbrf, catgrp, catrul, cattw, catwarning, evcomb)
miscellaneous control	60	(catcon, catdec)
report processing	55	(catrpt, tckrul)
movement calculations	54	(catmov, catmotion)
tacsit processing	43	(catts, catemt)
logbook processing	34	(logbook)
truth maintenance	20	(cattms, evcomb)
Total (all)	2347	
Total (non-db)	1468	
Total (non-db, non-exp)	452	

Analysis Method

This section will describe the manner in which the analysis was done. First the "literal" command will be described as well as its importance in determining to what extent a rule is capable of being generated by SKAT. Then a disclaimer will be forwarded stating the difficulty of the task of analyzing the rules. Following that, the method of analysis will be described.

The main means of deciding whether or not a rule could be generated was based on the use of the "literal" command. The "literal" command allows the user to name specific text that should be placed into the rule, either on the right- or left-hand side. The rule allows variable references to be included in the text, and the binding of the variables are replaced by the proper pointers in the target code. The "literal" command basically allows the user to create rules that do not fall into SKAT's understanding of a rule template. The amount of use of the "literal" command provides a strong indication of how much or how little the rule is covered by SKAT's idea of a tactical inference rule template, and is the main means of analysis employed here.

This paragraph is more or less a disclaimer for the analysis procedure. Analyzing the rules is a difficult task. First, it is not always clear if a rule falls into the category of tactical inference. In the event that it wasn't clear whether or not a rule was a tactical inference rule it was included as such and analyzed. Second, sometimes a rule is very close to being "generated," but it may have one statement in it that has to be interpreted (such as using a START working memory element as a condition element in order to have that rule dominate in the conflict set by specificity). This prompted the categories of "generatability" which are described in the next section. Even with the different categories, however, it was difficult to know where to place rules. Only a few rules were actually run through SKAT. The process of rewriting the OPS code into SCL code turned out to be a slow and tedious process that was abandoned in favor of looking at the rules and noting the areas of potential problems.

Because the rules were not actually run through SKAT, some problems may have escaped notice. On the whole, however, the analysis was done carefully, and most of the problems were identified.

Due to these difficulties, it was decided that a distinction between completely generated and ungenerated rules was undesirable. Thus, the capability of generating the rule was broken into four categories. The categories delineate the degree to which the "literal" command would have to be used in order to generate the CAT rule from SKAT. The breakdown is as follows:

A = Can be generated without any literal statements in SCL.
B = Can be generated with a few (less than 5) basic literal statements.
C = Can only be generated with complicated literal statements.
D = Cannot be generated.

After most of the analysis was completed it became obvious that the category **C** really had no meaning in that it became hard to distinguish from category **D**. When a rule was complicated it was easier to say that it couldn't be generated. The rule would mainly consist of "literal" statements anyway.

RESULTS

This section presents the results of the analysis. First, the category of rules actually analyzed will be described. The results of the analysis of the current version of SKAT will be presented. An analysis of a version of SKAT with some simple modifications will be presented.

The Rules That Were Closely Examined

Several categories of the rules were eliminated from consideration because they are different from the template of SKAT. Obviously they could not be generated. The categories that meet that criterion are

- alert processing
- miscellaneous control
- logbook processing
- truth maintenance
- part of the explanation rules
- part of miscellaneous inference – (catbrf, catwarning).

This means the rules that were examined are as follows:

- database rules
- database explanation rules
- part of miscellaneous inference rules – (catgrp, catrul, cattw)
- report processing
- movement calculations
- tacsit processing.

Since the database rules all have essentially the same format with only a few changes depending on the database key, they are considered a group. If one can be

generated, they all can be generated. Except for only a handful of rules, the same is true for the explanation rules.

Results of Analysis of SKAT

Following are the number of rules for each category of "generatability."

****

Analysis on current version of SKAT

Generation Category	Number of Rules
A	38
B	15
C	0
D	152

Database rules cannot be generated.
Explanation rules can be generated.

****

The numbers for category **A** are a little misleading in that 32 of the 38 rules fall in the module "catgrp." Each of the 32 rules in that module is very similar to the rest of the rules in that module. They have a certain feature on the left-hand-side (disjunction of predicates and values) that could be fixed by hand such that the rules could be generated by SKAT. The only prerequisite would be that each rule would have to be broken into several rules in order to handle the disjunction. That is the only problem with those rules. Therefore they were included in category **A**.

These data point out that SKAT currently is capable of generating only a few of the rules in NOSC's version of CAT. To some extent this inability could be expected. NOSC was not writing rules with CMU's idea of the template in mind. A rule in the NOSC system can contain much more "stuff" than the rule template at CMU would allow. The types of things that are allowed in the NOSC system vary considerably. Some of the differences are simple things that could be encompassed by changing the rule template as well as the SCL. Other differences are more profound in that they indicate other types of knowledge that SKAT currently is incapable of handling. This doesn't mean that SKAT is unable to incorporate this knowledge. These differences mean that it would take some major effort to include this knowledge.

Results of Analysis of Modified Version of SKAT

In order to better understand how much of the problem lies with simple template differences and how much lies with major differences in SKAT's understanding of a tactical inference rule, another analysis of the rules was performed. This analysis looked at the rules to see if they could be generated if certain simple changes to SKAT could be made. These changes are simple only requiring slight changes to the rule template. The changes that were included in the second analysis are

- Incorporate writes to screen on right-hand side of rule.

- Allow more than one conclusion per rule.
- Allow right-hand side removes on assertions.
- Incorporate source field checks in assertion condition elements.
- Allow matching and making of contexts.
- Incorporate ability to use current time on right-hand side.

Following are the number of rules for each category of "generatability" for the version of SKAT with the above changes:

****

Analysis on version of SKAT with minor modifications

Generation Category	Number of Rules
A	69
B	36
C	0
D	101

Database rules cannot be generated.
Explanation rules can be generated.

****

Simple changes to SKAT would provide some benefits to the generation process. Yet problems would still remain. The NOSC CAT rules have other knowledge, besides simple templated differences, encoded in them that SKAT currently is unable to deal with. The types of knowledge represented in CAT that are not fully accounted for in SKAT can be divided into three types:

- Knowledge about the target language.
- Knowledge about the domain.
- Knowledge about other parts of the system.

SKAT's Problems With Understanding of Target Language

SKAT only has a very limited knowledge about the data structures that are used in SKAT. Most of that knowledge is contained within print statements in the tactical rule generator. Thus the knowledge about what data structures in OPS83 are being used to represent assertions and the inference network resides with the SKAT programmer. If this knowledge was more explicit in SKAT, as well as knowledge about how to access different fields of the data structures, then SKAT could be extended to represent and allow referencing of other fields of the structures.

SKAT currently employs no knowledge about the control mechanisms used in OPS83. There are cases in NOSC's CAT rules in which the programmer used such knowledge to cause a rule to fire only in specific circumstances. This was done often through the use of a CONTEXT working memory element. Another example of this idea is contained in the use of the START working memory element in order to allow a rule to dominate other rules in conflict resolution because the rule with the START element would then be more specific than some other rules, which in the MEA

conflict resolution strategy is one of the filtering steps. It is unimportant that such programming practices may not be very "pretty." The importance is that such practices express a need to represent control in SKAT in such a way that similar types of control can be achieved in CAT, or at least similar functionality.

SKAT's Problems with Understanding of Domain

The CAT rules employed many "removes" on the right-hand sides of rules. Currently SKAT only deals with "makes" of assertions. The knowledge of when to remove assertions from the inference net or to modify assertions already in it is contained in the inference rules that produce the updates or superseding assertions. This points out the need for SKAT to attain understanding of the conditions in which an assertion is no longer true, or needs to be modified. This is an example of SKAT needing knowledge about the validity of assertions.

Another piece of knowledge that SKAT needs deals with the use of default reasoning in the CAT system. Many assertions in NOSC's CAT system are made in which REASON working memory elements are made but the evidence is not made. The evidence is one of the key elements for the inference network. In some cases this is indicative that the assertion is a default. In other cases it is a way of isolating the assertion from the inference net so that it is not removed and remade whenever changes are made to any of its antecedents. This again points out a need for SKAT to understand how long information is valid in the system and what should be done when the information is no longer valid. It shows a need for somehow breaking out default information into separate rules.

Inference rules in NOSC's CAT are more contextualized than in the CMU system. Some of the contexts are used as procedure calls to perform certain calculations (e.g., cpa). Others are used as procedure calls to check if a recently calculated value indicated a change in the old value (e.g., course or speed change). In those cases the rules are used to match the needed assertions to do the comparison, but the comparisons themselves are done on the right-hand side so they can specify a tolerance for the comparison (i.e., if courses are more than 5.0 degrees apart then there has been a course change). All of those cases indicate a need for SKAT to have more complete knowledge about the control issues involved with writing the system in a target language such as OPS83.

In many of the rules, REASON working memory elements are matched on the left-hand side. There are examples of rules using knowledge about how an assertion was made to decide whether or not to match it. The source fields in some assertions are also checked for this reason. These kind of examples show a need for SKAT to have more knowledge about how to deal with the source of the information in a rule.

SKAT's Problem With Understanding of Rule Interaction

SKAT doesn't employ any knowledge about what rules are affected by other rules. In other words, there is no knowledge about what right-hand side conclusions are matched on the left-hand side of other rules or the transitive closure of other rules. This is a manifestation of SKAT's nearsightedness about the other rules that it has already learned. This type of knowledge is important also for cases in which rules are inconsistent with the rest of the knowledge base, or are breaking a chain of reasoning by firing at inappropriate times.

POSSIBLE FUTURE RESEARCH

There are two main areas of research that have direct relevance to SKAT and to the solution of some of the problems discussed above. The two major areas are in task-level parallelism and expert system problem-solving methods.

One idea is to look at possibilities of task-level parallelism in production systems. CMU is looking at ways of exploiting parallelism in the match portion of the recognize-act cycle. Task-level parallelism would focus more on exploiting parallelism in domain dependent task areas.

This focus has direct implications for the development of SKAT. In order to generate systems that take advantage of parallelism at the task level, the parallel task areas first need to be uncovered. SKAT is a tool that could do such an analysis of the rule base. This implies that SKAT would have to develop a database of the rules that are in the system as well as a more thorough understanding of the ways in which they interact. In addition, a better understanding of the control information that is contained within the rules would have to be obtained. Development along those lines would alleviate some of the problems previously discussed.

The other major effort would be to explore the idea of a problem-solving method for CAT. The theory, attributable to John McDermott, et al., is that rule-based system can be reasonably maintained and developed if the high level problem-solving procedure that the system employs is understood and made explicit in the design of the system. The problem-solving method helps to point out where new knowledge needs to be added to the system in case the system is deficient in that area. It also helps control rule interaction by carefully defining at what points in the method each rule is available to be instantiated.

This type of development is relevant to SKAT in that an understanding of the problem-solving method used by the system would provide a thorough understanding of how and when the rules in the system should be employed. This means SKAT would need an understanding of how control mechanisms are represented.