INTRODUCTION

HACKER (Sussman 75) is a problem-solving system whose performance improves with practice. The system is a concrete realization of Sussman's theories on automating skill acquisition. HACKER develops skills in writing programs (or plans) to perform construction tasks in the Blocks World*. Problems submitted to HACKER consist of an initial Blocks World situation and a goal kernel (those relations which must be satisfied in the final situation). HACKER then creates a program which will transform the initial situation into a final situation which satisfies the goal kernel. The program which actually performs the Blocks Word manipulations is called the performance program. An example problem is:

```
| A | B |
---|---|
|   | C |
---|---|
```

Goal Kernel:

```
(MAKE (ON B C))
```

```
| A | B |
---|---|
|   | C |
---|---|
```

Initial Situation (A Plausible Final Situation)

During the process of creating the requested program HACKER brings to bear all its previously accumulated knowledge about programming in the Blocks World. New knowledge may be acquired which will be remembered for future use. As HACKER solves more problems, it is able to create new programs with greater accuracy and efficiency.

With no training, HACKER solves the problem illustrated above by first writing a program and then debugging it. The performance program is:

```
(PUTON A TABLE)
(PUTON B C)
```

By solving the example problem HACKER learns that in order to move a block its top must be clear.

* HACKER is a domain independent problem solver. The Blocks World was chosen as a well understood simple domain in which fairly complicated behaviour can be observed.
Sussman classifies problem-solving systems into two categories: the experts and the generalists. The expert systems are designed to perform efficiently within a narrow field of expertise. Examples of expert problem-solving systems are BUILD (Fahlman 74), an expert in Blocks World construction tasks), DENDRAL (Buchanan 69), SHRDLU (Winograd 72), and MACSYMA (Bogen 73). They achieve this efficiency by procedurally representing much of their knowledge. A procedure is associated with each type of problem (or subproblem) the system knows how to solve. The correct sequencing of actions has been programmed into these procedures in such a way that interactions (setting up, interfacing, cleaning up) among procedures are always handled correctly. Each piece of knowledge (procedure) explicitly makes reference (subroutine calls) to all the other knowledge which it finds relevant.

The generalist systems are designed to be applicable to any performance domain and be easily extensible. Examples of general problem-solving systems are GPS (Ernst 69) and STRIPS (Fikes 71). These systems have a uniform procedure that operates on a database of knowledge which consists of declarative information. Declarative knowledge is modular, with each piece representing a complete fact, making no explicit references to other facts. The interweaving of related facts is left to the uniform problem-solving procedure. New knowledge can be easily assimilated by the problem solving system. The addition of this new knowledge should at worst cause the updating of a small amount of information already in the database. By replacing the entire database the performance domain can be altered.

Modifying expert systems is difficult because of the lack of modularity of knowledge. Knowledge is distributed throughout all the procedures in the system in the implicit form of sequencing information. Performance in generalist systems is poor as a result of little interaction among the pieces of knowledge. Interactions must continually be deduced. Thus a fundamental problem arises: Efficiency is improved by the interactions among knowledge while extensibility is enhanced by the absence of these interactions.

HACKER attempts to solve this problem through its ability to represent and use knowledge in two different forms. A declarative representation is used for communication between the user and the program, while a procedural representation is used when the performance system is required to actually solve problems. A crucial aspect of HACKER is the ability to translate declarative knowledge into a procedural form and then remember knowledge accumulated during this transformation for use when appropriate.

Since the appearance of Sussman's thesis there has emerged a hybrid problem-solving system typified by Mycin (Shortliffe 76). This is an expert system for diagnosing infectious diseases, designed specifically to allow easy extensibility. Knowledge is declaratively represented within the system. What Mycin has achieved is a highly constrained knowledge format which is rich enough to encompass a large portion of the required knowledge and rigid enough to allow efficient manipulation by a search mechanism. One view of this system in light of the above dichotomy would be that the interfaces in Mycin's task domain are well understood to allow each piece of declarative knowledge added to the system to be compiled in, by determining all relevant facts that it will have to deal with. The interfaces between these compiled facts are uniform and can be handled easily by the search mechanism.
The structure of HACKER is illustrated in the above flow diagram. Polygons represent computations and circles represent knowledge sources. Control flows along the solid arrows while information flows along the dotted arrows. Polygons with stars represent possible recursive invocations of HACKER. Circles with plusses represent knowledge sources in which knowledge acquired by HACKER is placed.

When HACKER is presented with a problem it either retrieves a previously written program or writes a new one. Programs are written by performing a tree search in the solution space using all relevant knowledge. The program is then run and if any bugs become manifest the debugger is invoked. The program is then debugged and any globally useful information is retained for use during future program construction. We now explain in greater detail the steps taken by HACKER in solving a problem.

Included in this discussion is a complete list of the knowledge sources available to HACKER. A knowledge source is just a database of expressions which represent patterns. When the program debugger is invoked, HACKER checks for an applicable pattern in the problem. When several patterns apply, the debugger chooses one of them for demonstration purposes. The debugger then chooses the values of the variables and matches the pattern. In general, the pattern checks whether the program is to be continued.
the following examples all upper case words represent pattern constants while all lower case words represent pattern variables.
Answer Library: When a goal (or subgoal) is given to HACKER it first looks in the Answer Library for a program whose applicability pattern matches the desired goal. The Answer Library consists of a set of programs which advertise their function by an applicability pattern with the keyword TO, and are of the form (TO <goal> <code>).

The following is a typical piece of knowledge in the Answer Library. It states that if HACKER ever has a goal of making one block sit on another, it can accomplish this by executing a program which consists of the statement (PUTON x y). Both x and y are bound by the pattern matcher when matching the goal.

(TO (MAKE (ON x y)) (PUTON x y))

If a successful match is found the program is run. If the program runs to completion with no errors HACKER returns for a new problem. If the program fails it is re-run in CAREFUL mode.

Initially there are only a few rudimentary programs placed in the Answer Library. As HACKER solves more problems, the Answer Library is enhanced by either improving already existing programs or writing new programs, generalizing and subroutinizing them, and adding them to the Answer Library.

HACKER's Notebook: If no program is found in the Answer Library HackER checks its notebook to see if a program has ever been written to achieve a similar goal (such as achieving the same goal but with different objects). Whenever HACKER writes a new piece of code it is placed in HACKER's Notebook not directly in the Answer Library. If a program is found it is generalized (by abstracting which constants in the previously written program can take arbitrary values) and subroutinized (placed in the Answer Library with the appropriate applicability pattern). This two-level strategy keeps the Answer Library stocked with useful subroutines and not cluttered with programs that are too specific. For example, if HACKER's Notebook contained the information:

(TO (MAKE (ON A B)) (CLEARTOP A) (PUTON A B))

and the current goal was:

(MAKE (ON C D))

then HACKER would follow the above procedure and store:

(TO (MAKE (ON x y)) (CLEARTOP x) (PUTON x y))

in its Answer Library. HACKER then backs up to the beginning and starts again, now able to make use of the new program.

Program Proposer: If no similar program is found in HACKER's Notebook control passes to the program proposer which advertises for a method to achieve the goal. The Program Techniques Library and Blocks World Knowledge Library are used to transform the given goal into either a piece of code or a set of subgoals whose solutions imply a solution to the main goal. This is accomplished by replacing the goal with the macro expansion found during advertisement. HACKER tries to solve each of the newly created goals in the same way as the original. Eventually the goals bottom-
out and HACKER reaches a level at which code can be immediately produced.
OVERVIEW OF HACKER (cont.)

The Blocks World Knowledge Library contains facts specific to the Blocks World. These facts can be applied to change the representations of goals such as expanding a definition. There are also facts which state information such as prerequisites for Blocks World primitives. This entire library would have to be replaced if HACKER was to solve problems within another domain. An example fact from this Library is:

\[(\text{FACT}\ (\text{MEANING-OF}\ (\text{CLEARTOP}\ x)\ (\text{NOT}\ (\text{EXISTS}\ (y)\ (\text{ON}\ y\ x))))))\]

This fact states that the meaning of achieving a CLEARTOP for object \(x\) is that there not exist any object which has the property of being on \(x\).

The Programming Techniques Library contains facts which attempt to transform goals into runnable code. Programming knowledge typically transforms a goal into a skeletal piece of code in which alternate (simpler) goals are embedded. In the following two examples \(\text{pat}\) matches any goal and \(\text{vars}\) matches the list of variables in \(\text{pat}\).

\[(\text{FACT}\ (\text{CODE}\ (\text{ACHIEVE}\ (\text{AND}\ \text{pat1}\ \text{pat2})\ (\text{ACHIEVE}\ \text{pat1})\ (\text{ACHIEVE}\ \text{pat2}))))\]

\[(\text{FACT}\ (\text{CODE}\ (\text{ACHIEVE}\ (\text{NOT}\ (\text{EXISTS}\ (\text{vars})\ \text{pat}))))\]

\[(\text{UNTIL}\ \text{vars}\ (\text{CANNOT}\ (\text{ASSIGN}\ \text{vars}\ \text{pat}))\ (\text{MAKE}\ (\text{NOT}\ \text{pat}))))\]

The first fact states that one possible way to solve conjunctive goals is to concatenate the code needed to independently solve each conjunct. This technique is important enough to be given a name, the 'linear assumption'.

The second example is a fact telling how to write a piece of code to achieve a state in which it is impossible to find a set of bindings for the variables in \(\text{pat}\) which makes \(\text{pat}\) a true statement. The semantics of this fact is best illustrated with an example. Suppose HACKER wishes to write a program segment which achieves (CLEARTOP \(A\)). By using the example fact in the Blocks World Library this is equivalent to writing a program to achieve

\[(\text{NOT}\ (\text{EXISTS}\ (y)\ (\text{ON}\ y\ A))))\]

After performing a pattern match for this goal in the Programming Techniques Library we find that the piece of code which accomplish this goal is:

\[(\text{UNTIL}\ y\ (\text{CANNOT}\ (\text{ASSIGN}\ y\ (\text{ON}\ y\ A))\ (\text{MAKE}\ (\text{NOT}\ (\text{ON}\ y\ A)))))\]

\(\text{ASSIGN}\) is a program in the Answer Library which searches for an object with the property that it is on \(A\). If one is found \(\text{ASSIGN}\) binds that object to the variable \(y\). The code can be described as looping until \(\text{ASSIGN}\) can find no object with the property that it is on \(A\). Thus the code will loop until \(A\) has a clear top. Inside the loop we introduce a new (sub) goal which is simpler than the original. The original goal is concerned with removing all the objects from \(A\) while this goal is only concerned with removing the object bound to variable \(y\) from \(A\). Later, this goal will be rewritten by a fact which says that \(y\) being on anything which is not \(A\) suffices for \(y\) not being on \(A\). HACKER then finds an different object (typically the TABLE)
and code is then written to put y on this object.
OVERVIEW OF HACKER (cont.)

Let's return to examine another important function of the Program Proposer. Code is extensively commented by HACKER for possible future reference by the debugger. These comments distinguish among main steps and auxiliary steps (i.e. those needed to achieve a prerequisite or interface between two procedures). Comments help in keeping track of the purpose of each line of code as well as the scope of the goal which the code achieves. The scope of a goal is that interval of time over which the goal must remain achieved.

We now present the beginning of an example which will be used later to illustrate how HACKER debugs programs. We give a neophyte HACKER the problem of building a tower:

\[
\text{(MAKE (AND (ON A B) (ON B C)))}
\]

HACKER uses the above mentioned linear assumption and creates the partial program:

\[
\text{(HPROG ANDI)}
\]
\[
\text{(LINE1 (ACHIEVE (ON A B)))}
\]
\[
\text{(LINE2 (ACHIEVE (ON B C)))}
\]

which just states that HACKER will first try to get A onto B and then will try to get B onto C. The following comments are also produced:

\[
\text{(GOAL ANDI (MAKE (AND (ON A B) (ON B C))))}
\]
\[
\text{(PURPOSE L1 (ACHIEVE (ON A B)) ANDI)}
\]
\[
\text{(PURPOSE L2 (ACHIEVE (ON B C)) ANDI)}
\]

Focus on the second comment which states that the purpose of LINE1 (and thus all the code which it will finally produce) is to get A onto B. The comment also states that after the goal (ON A B) is achieved it must not be undone until after the code for ANDI is exited. The scope of the goal (ON A B) is the equivalent to the time during which ANDI is being achieved. Note that this will soon get HACKER into trouble since line L2 will force A to be removed from B in order to move B onto C. Later sections will discuss HACKER's ability to diagnose and recover from this problem.

Program Criticizer: When the Program Proposer finally has a completed program it is held up for criticism. The Criticizer reads the proposed program to all of the Critics in the Critics Gallery to determine if any known bugs have been created. Each Critic that spots a bug sends a message to the Criticizer specifying the code which must be debugged and the bug's class. The Bug Patcher is called to correct each bug that was spotted. If no bugs were spotted HACKER proceeds to run the program in CAREFUL mode. The Critics Gallery is initially empty but is extended as HACKER gains more knowledge about programming (see the Bug Summarizer section).

Bug Patcher: The Bug Patcher advertises for patches to each bug that the Program Criticizer found. The different available patches are contained in the Types of Patches Knowledge Source. The suggested patches are made and the program is held up for criticism again. The patch types are keyed on the bug's class and are all domain independent. An example fact is:

\[
\text{(FACT (PATCH (PREREQUISITE-CLOBBERS-BROTHER-GOAL prog line1 line2 prereq) (BEFORE line2 line1)))}
\]

Assume that the Program Criticizer gives the Bug Patcher two lines of a code
with the message that a PCBG bug is present. The above fact states that this class of bug can be fixed by making sure that line2 preceeds line1 in the final program. See the Bug Summarizer section for an example bug that can be fixed by this type of patch. HACKER then returns to the Program Criticizer (the patches might create new bugs) for another round of debugging.
Execution in CAREFUL mode: After reading a program without receiving any criticism the program is run in CAREFUL mode, which allows HACKER to immediately spot any bugs which become manifest. The types of bug manifestations are:

1) Unsatisfied Prerequisite:
   Tried to perform an operation without correctly setting up.
2) Protection Violation:
   Tried to perform an operation which negated a protected goal.
3) Double move:
   An aesthetic problem which may or may not be a bug.

Bug types 1 and 3 are discovered by the Blocks World primitive operations.

When a program is run in CAREFUL mode two things happen: First a complete chronological trace of the program's execution is kept. Second the protection mechanism is invoked, which causes goals to be protected over their scopes. This allows HACKER to discover if a goal is ever achieved and then undone within its scope (bug type 2). If any bugs become manifest, control, as well as the accumulated information from the program's execution, is passed to the bug classifier.

Bug Classification: If a bug becomes manifest the Bug Classifier determines its underlying cause. The classifier looks at the teleological model of the program in order to classify the bug. The teleological model consists of the chronological execution stack of the buggy program (what the program actually did) and the comments made during the construction of that program (what the program was supposed to do). The classification is done in a domain independent matter. The bug classes are:

1) Prerequisite-Missing (PM):
   Must establish a prerequisite before performing the requested operation.
2) Prerequisite-Conflict-Brothers (PCB):
   Two brother goals have prerequisites that conflict.
3) Prerequisite-Clobbers-Brother-Goal (PCBG):
   While trying to establish a prerequisite the current goal clobbered its brother goal.
4) Strategy-Clobbers-Brother (SCB):
   While using a strategy to achieve a goal a previous strategy's effects are undone.
5) Direct-Conflict-Brothers:
   Two goals cannot be independently satisfied.
6) Anomalous:
   Unclassifiable

After a bug has been classified it is abstracted by the Bug Summerizer which compiles a critic to spot portions of code in future proposed programs which have this bug.

Going back to our example with the linear assumption we see that a protection violation will occur when HACKER writes code to put B on top of C. The brother goal (ON A B) is violated by a prerequisite of putting B on C. The prerequisite (found in the Blocks World Knowledge Library) states that B must have a clear top before it can be moved. Thus we have a prerequisite clobbering a brother goal. This information along with the code that contained the error is given to the Bug Summerizer.
OVERVIEW OF HACKER (cont.)

Bug Summarizer: After the Bug Classifier determines the class of a bug manifestation the Bug Summarizer compiles a Critic to spot future attempts to use the buggy code construct. This Critic is then added to the Critics Gallery. The Bug Summarizer generalizes the buggy code construct and instantiates a template in the Types of Critics Knowledge Source (which is domain independent). The templates are advertised for by bug class. An example critic template is:

(FACT (CRITIC (PREREQUISITE-CLOBBERS-BROTHER-GOAL prog linel line2 pre) (WATCH-FOR (ORDER (PURPOSE linel (GOAL linel) target) (PURPOSE line2 (GOAL line2) target)) (PREREQUISITE-CLOBBERS-BROTHER-GOAL current-prog linel line2 prereq))))

Using the information gleaned from the Bug Classifier and the actual code we generalize the objects and instantiate the above template to form the Critic:

(WATCH-FOR (ORDER (PURPOSE linel (ACHIEVE (ON a b)) t) (PURPOSE line2 (ACHIEVE (ON b c)) t)) (PREREQUISITE-CLOBBERS-BROTHER-GOAL current-prog linel line2 (CLEARTOP b))))

This states that if the Program Proposer ever writes a program in which there is a piece of code to (ACHIEVE (ON a b)) before there is a piece of code to (ACHIEVE (ON b c)) then there is a bug of type PCBG. Recall that this type of bug can be patched by reversing the order of the goals.

After a bug is summarized the program is given back to the Program Criticizer where the bug will be spotted and then patched.
THEMES OF HACKER (cont.)

The Subroutine Commitment: Much of Hacker's power is derived from its philosophy of subroutinization. This has been discussed before but we now bring a fundamental problem to light. There are some 'anomalous' problems which cannot be solved by an arrangement of previously constructed subroutines. The factorization of tasks is wrong. The subroutines interfere with each other regardless of their order. That is, the solution requires a merged sequence of instructions from the subroutines (Waldinger 75, Tate 74). An example of this is a slight perturbation of our original two examples.

```
  |     |  |  |
  |  A  |   |  
  |     |  |  |
  |     |  |  |
  |     |  |  |
  |  C  |  |  |
  |     |  |  |
  |     |  |  |
  |     |  |  |
Goal Kernel: (MAKE (AND (ON A B) (ON B C))
  |  A  |  |  |
  |  B  |  |  |
  |  C  |  |  |
-------------------------------  ---------------------------
Initial Situation              Final Situation
```

If Hacker tries to first put B on C it is stuck with A on the bottom and must undo the achieved goal. If Hacker tries to first put A on B it is left with the top of the tower assembled but not on the bottom of the tower. Again a previously achieved goal must be undone. The bug classifier cannot classify this behaviour into any known type so it is classified as anomalous. The mechanism introduced to solve problems with an anomalous bug is called Deferred Goal Mode. This mode allows an achieved goal to temporarily become unachieved. The system then very carefully monitors execution and re-establishes the goal as soon as possible. This mechanism allows for the merging of arbitrary subroutines.

HACKER solves the above example by:

```
(PUTON B C) achieve the first goal
(PUTON B TABLE) temporarily undo the first goal*
(PUTON C TABLE)
(PUTON B C) re-achieve first goal
(PUTON A B) achieve second goal
```

* Sussman notes that it would be fruitful to study the problem of double move bugs in Deferred Goal Mode.
REFERENCES:

Bogen, et. al.
Macsyma User's Manual
MIT Project MAC (1973)

Buchanan, B.

Ernst, G. & Newell, A.
GPS: A Case Study in Generality and Problem Solving

Fahlman, S.E.
A Planning System for Robot Construction Tasks

Fikes, R. & Nilsson, N.
STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving
Artificial Intelligence, Vol. 2, No. 2-4, pp. 189-208 (1971)

Sacerdoti, E.
A Structure for Plans and Structure

Shortliffe, T.
MYCIN: A Rule Based Computer Program for Advising Physicians Regarding Antimicrobial
Therapy Selection.

Sussman, G.J.
A Computational Model of Skill Acquisition

Tate, A
INTERPLAN: A Plan Generation System that Can Deal with Interactions between Goals
Machine Intelligence Research Unit, Memo. MIP-R-189 (1974)

Waldinger, R.
Achieving Several Goals Simultaneously
SRI Artificial Intelligence Center Tech Note 107 (1975)

Winograd, T.
Understanding Natural Language
Academic Press, New York (1972)

Winston, P.H.
Learning Structural Descriptions from Examples
Chapter 5 (157-210) Psychology of Computer Vision