l descriptions in mental

oints from perspective
26, 256–267 (1984).
ier descriptors," *IEEE*
7).
nensional objects using
ems, Man, Cybernetics,

ovement using Fourier
PAMI-2(6), 583–588

craft recognition algo-
ics and Image Process-

New York, 1968.
New York, 1970.
oyal Soc. London, B.

ambridge, MA, 1979.
od by group represen-
Computer Vision and

IEEE Workshop on
ocessing, June 28–29,

uter systems," *Com-*

Proc. Image Under-
ford, CA, 193–203.
cture of a rigid body
and Dynamic Scene
w York, 1983, pp.

New York, 1976.
uter, August, 39–48

nalysis," *Computer*

tor," *IEEE Trans.*

ory of a moving car
h International Joint
592–697.
uter Vision, Graph-

ng in space," *IEEE*

onstraints: skewed
Picture Data Des-

l Intelligence, **17**,

ence, **8**, 201–231

spatial organiza-
, 269–294 (1978).

ORIGINAL

Hierarchical Representation of Problem-Solving Knowledge in a Frame-Based Process Planning System

Dana S. Nau*†
Computer Science Department, University of Maryland, College Park, Maryland 20742
Tien-Chien Chang
School of Industrial Engineering, Purdue University, West Lafayette, Indiana 47907

In most frame-based reasoning systems, the information being manipulated is represented using frames, but the problem-solving knowledge that manipulates the frames is represented as production rules. One problem with this approach is that rules are not always a natural way to represent knowledge; another is that systems containing lots of rules may suffer from problems with "exponential blowup" in the amount of computation required. This paper describes a way to address these problems by organizing the problem-solving knowledge not as rules, but in a particular kind of frame hierarchy. The approach described in this paper has been implemented in a problem-solving system called SIPP (Semi-Intelligent Process Planner), which produces plans of action for the manufacture of metal parts. The paper gives an overview of SIPP, compares its knowledge representation and problem solving methods to approaches used in other knowledge-based systems, and describes goals for further research.

I. INTRODUCTION

In most frame-based reasoning systems, the information being manipulated is stored in the form of frames, but the problem-solving knowledge that manipulates the frames is stored separately in the form of production rules. One problem with this approach is that rules are not always a natural way to represent knowledge; another is that systems containing lots of rules may suffer from problems with "exponential blowup" in the amount of computation required.

SIPP (Semi-Intelligent Process Planner) addresses these problems by organ-

*To whom all correspondence should be addressed.

†This work was supported in part by a Presidental Young Investigator Award to Dana S. Nau, with matching funds provided by IBM, General Motors, and Martin Marietta.

INTERNATIONAL JOURNAL OF INTELLIGENT SYSTEMS, VOL. 1, 29–44 (1986)

CCC 0884-8173/86/010029-16$04.00

izing the problem-solving knowledge not as rules, but in an unusual frame-based hierarchical organization. Problem solving in SIPP is done by means of a modified Branch and Bound procedure somewhat similar to forward chaining, which produces sequences of frames which form plans of action. For SIPP's application domain, this approach appears more natural than rule-based representation—and it can also be used to avoid problems with exponential blowup.

SIPP was developed for use in a manufacturing task called generative process planning. SIPP produces least-cost plans of action for the creation of metal parts using metal removal operations, based on knowledge about the intrinsic capabilities of each manufacturing operation. It is anticipated that the approach used in SIPP will be useful in other problem domains as well.

This paper gives an overview of SIPP, compares its knowledge representation and problem solving methods to approaches used in other knowledge-based systems, and discusses the virtues and drawbacks of Prolog as a language in which to write such systems. The paper also discusses the implications of this work for further research on AI problem solving, geometric reasoning, and automated manufacturing.

II. PROCESS PLANNING

Process planning (also called manufacturing planning) is the task of determining what machining processes and parameters are to be used in manufacturing a part. Process planning is distinct from *production planning*, which involves taking the process plans for all parts to be produced and scheduling a factory's resources to produce these parts.

Devising a process plan automatically using the part's specifications (e.g., a full technical drawing) is a very difficult problem. However, computer systems which provide even partial automation of process planning yield substantial benefits in the efficiency of a job shop. This promise has led to substantial research in computer-aided process planning.

In most existing computer-aided process planning systems, process selection is based on the use of Group Technology (GT) codes.[4] A GT code is a numerical code which a one may assign to a part, based on various of the part's features. In general, more than one part will have the same GT code—but parts which have the same GT code will be "similar" in the sense that they are produced using similar machining processes. Before such a process planning system can be used, one must decide upon an appropriate GT coding scheme, and in addition one must group similar parts into part families, with each family corresponding to one or more GT codes.

For each part family F, one must then prepare a standard process plan for F—i.e., a process plan for some representative part in F. This plan will presumably be similar to the process plans for other members of F. When a process plan for some part P is desired, a human enters the GT code for P into a database retrieval system. The system uses the code to find the part family F in with P belongs, and retrieves the standard process plan for F. The user then modifies this plan by hand to produce a process plan for P.

Systems using this approach, such as CAPP[14] and MIPLAN[23] are called

variant process pl
they allow process
process planning w
a GT code. For e
surface to be mac

A few process
use as input a rep

CPPP[15,8] is
rotational (lathe-t
"part family" con
gives CPPP a cod

The part fam
computer progran
model must have
planner, and it mu
the family. Becau
process model am
is used to select p

APPAS[27] is a
APPAS, the user
then uses knowle
each surface. AP
the geometrical r

CADCAM[3,
cesses. Its main fe
part interactively
decision tables. A
generated which

TIPPS[6] is a p
selection, param
modified bounda
augmented (to in
Process capabiliti
modeled in an *if-t*
on a CRT displa
plan using proce
language.

The use of r
mentally in GAR
expert system te
details. Each of th
it can produce.

*By a *machina*
surfaces which can
single machined su
information on ma

unusual frame-based
y means of a modified
'ard chaining, which
or SIPP's application
representation—and
wup.
d generative process
eation of metal parts
he intrinsic capabili-
he approach used in

wledge representa-
er knowledge-based
a language in which
ons of this work for
ng, and automated

the task of deter-
d in manufacturing
g, which involves
eduling a factory's

cifications (e.g., a
computer systems
yield substantial
bstantial research

process selection
de is a numerical
art's features. In
parts which have
produced using
tem can be used,
in addition one
sponding to one

process plan for
n will presuma-
process plan for
tabase retrieval
P belongs, and
is plan by hand

N[23] are called

variant process planning systems. Such systems are quite useful in industry, as they allow process plans to be made very quickly. However, fuller automation of process planning will require more complete information about the part than just a GT code. For example, detailed information must be available about each surface to be machined.

A few process planning systems have been developed experimentally which use as input a representation of each of the machinable surfaces of the part.*

CPPP[15,8] is a system which does process selection and sequencing for rotational (lathe-turned) parts. To use CPPP, a user gives CPPP the name of a "part family" containing the desired part, and for each surface to be machined, gives CPPP a code describing the surface.

The part family name is used to retrieve a process model, which is a simple computer program written in a language designed for that purpose. The process model must have been previously written and debugged by a human process planner, and it must be general enough to handle all the possible part features in the family. Because of the restrictions of the process planning language, the process model amounts to a decision tree in which information about the surfaces is used to select processes and put them into the appropriate order.

APPAS[27] is a system which does process selection for prismatic parts. To use APPAS, the user enters for each surface a code describing the surface. APPAS then uses knowledge of the capabilities of various processes to select a process for each surface. APPAS considers only individual surfaces, and does not consider the geometrical relationships which may occur among the surfaces.

CADCAM[3,5] is an extension of APPAS. It is limited to hole-making processes. Its main features are a graphics interface which allows the user to design a part interactively, and an encoding of the process planning decision logic into decision tables. As soon as a design is completed, a detailed process plan can be generated which provides quick feedback to the designer.

TIPPS[6] is a process planning system consisting of modules for input, process selection, parameter selection and report generation. The input to TIPPS is a modified boundary model of a part in an IGES-like format which has been augmented (to include information about tolerances and surface conditions). Process capabilities are modeled in a language called PKI in which each process is modeled in an *if-then* rule. The user identifies the machining surfaces interactively on a CRT display. After surfaces are identified, the system generates a process plan using process knowledge. The system is limited by the process modelling language.

The use of rule-based reasoning for process planning is being tried experimentally in GARI[7] and TOM[17] Another system[25] is said to use some elementary expert system techniques, but the paper describing it does not give any further details. Each of these systems has various limitations on the kinds of process plans it can produce.

*By a *machinable surface* we mean any geometric surface or combination of geometric surfaces which can be produced by a single machine tool operation. For example, a hole is a single machined surface, although it consists of both a cylinder and a cone. For more information on machining operations, see Ref. 2, Chap. 1.

As discussed in Refs. 17 and 19, more sophisticated approaches to generative process planning will require sophisticated techniques for representing and reasoning about three-dimensional objects, such as the techniques described in Requicha's work.[22] Our ultimate goal is to develop a system which integrates the use of AI techniques with solid modelling techniques such as those used in PADL-2.[24] SIPP represents a first step in this direction.

III. REPRESENTING STATIC KNOWLEDGE

In order to represent knowledge, SIPP includes a frame-based knowledge representation language. Except for the fact that the frame system was built in Prolog, the way that *static* knowledge is represented in SIPP is not particularly new. SIPP's frame system is reminiscent of several other such systems (for example, Ref. 21). Message-passing of the kind used in Smalltalk and Flavors (for example, see Ref. 26) was not implemented in SIPP because it was not necessary for our purposes; however, elementary message-passing would be relatively easy to add.[13]

SIPP's frame system allows the user to define two types of objects: *items* and *types*. Types correspond to sets, and items correspond to members of sets. Normally, the knowledge about a problem domain consists of declarations of types, and the data used in solving a specific problem in that domain (both the input data defining the problem and the intermediate data created while the problem is being solved) consists of items. Each type or item is defined by a frame.

Since SIPP's frame language was was implemented using Prolog, its syntax is reminiscent of Prolog syntax. However, the statements in this language are interpreted by the frame system rather than by Prolog. As an example the frames shown below describe a hierarchical structure in which a flat_surface is a polyhedral_surface, a polyhedral_surface is a non_cylindrical_surface, a non_cylindrical_surface is a surface, and a surface is a thing.

```
type(thing, nil).
slots(thing,[
        [parent, X, item(X,_)],
        [pathcost, X, number(X)],
        [goal, X, item(X,_)],
        [successes, X, list_of_atoms(X)],
        [status, X, true],
        [comment, X, true],
        [active, X, true],
        [precedence, X, number(X)]
defaultvals(thing, [successes=[ ]]).

type(surface, thing).
slots(surface, [
        [surface_finish, X, number(X)],
        [pos_tolerance, X, number(X)],
```

```
    [neg_tolerance, X, num
    [wall_thickness, X, num
        /* wall_thickness
           casting */
    [contains, X, list_of_atc
        /* surfaces conta
    [adjacent, X, list_of_at
        /* all surfaces ad
]).

type(non_cylindrical_surface, 

type(polyhedral_surface, non_
slots(polyhedral_surface, [
    [norm, [X,Y,Z], (numt
    [flatness, X, number(X
    [angularity, X, number
    [parallelism, X, numbe
]).

type(flat_surface, polyhedral_
```

The "slots" entries for polyhe
objects have, as well as what kinds
these slots. Slot declarations are
flat_surface is a surface, a flat_surfa
i.e., inheritance from more than c
although it is not needed in the c

The "defaultvals" entry for th
Any time a default value is specif
levels of the hierarchy. For exam

```
item(fl, flat_surface).
slotvals(fl, [
      norm=[1,0,0],
      flatness=0.1,
      pos_tolerance=0.1,
      neg_tolerance=0.1,
      surface_finish=100,
      adjacent=[f2,f3,f4,f5]
]).
```

Then fl will inherit the slot value
we instead define fl as:

```
item (fl, flat_surface).
slotvals(fl, [
      norm=[1,0,0],
      flatness=0.1,
```

phisticated approaches to generative echniques for representing and rea- ich as the techniques described in develop a system which integrates ling techniques such as those used this direction.

TIC KNOWLEDGE

includes a frame-based knowledge that the frame system was built in resented in SIPP is not particularly several other such systems (for ex- d used in Smalltalk and Flavors (for n SIPP because it was not necessary ge-passing would be relatively easy

efine two types of objects: *items* and s correspond to members of sets domain consists of declarations of problem in that domain (both the termediate data created while the h type or item is defined by a frame plemented using Prolog, its syntax , the statements in this language than by Prolog. As an example, cal structure in which a flat_surface ce is a non_cylindrical_surface, a urface is a thing.

```
      [neg_tolerance, X, number(X)],
      [wall_thickness, X, number(X)],
            /* wall_thickness=how much deeper than the stock or
                  casting */
      [contains, X, list_of_atoms(X)],
            /* surfaces contained by this one (e.g., holes & slots) */
      [adjacent, X, list_of_atoms(X)]
            /* all surfaces adjacent to this one */
  ]).
  type(non_cylindrical_surface, surface).
  type(polyhedral_surface, non_cylindrical_surface).
  slots(polyhedral_surface, [
      [norm, [X,Y,Z], (number(X),number(Y),number(Z))],
      [flatness, X, number(X)],
      [angularity, X, number(X)],
      [parallelism, X, number(X)]
  ]).
  type(flat_surface, polyhedral_surface).
```

The "slots" entries for polyhedral_surface, and thing specify what slots these objects have, as well as what kinds of data values are permissible to be stored in these slots. Slot declarations are inherited from above; for example, since a flat_surface is a surface, a flat_surface has a slot called *adjacent*. Multiple inheritance (i.e., inheritance from more than one set of ancestors) is also supported in SIPP, although it is not needed in the current knowledge base.

The "defaultvals" entry for thing specifies a default value for one of its slots. Any time a default value is specified for a type, it may be overridden at lower levels of the hierarchy. For example, suppose we define the item fl as:

```
  item(fl, flat_surface).
  slotvals(fl, [
        norm=[1,0,0],
        flatness=0.1,
        pos_tolerance=0.1,
        neg_tolerance=0.1,
        surface_finish=100,
        adjacent=[f2,f3,f4,f5]
  ]).
```

Then fl will inherit the slot value "successes=[]" from thing. However, suppose we instead define fl as:

```
  item (fl, flat_surface).
  slotvals(fl, [
        norm=[1,0,0],
        flatness=0.1,
        surface_finish=100,
        adjacent=[f2,f3,f4,f5],
          successes=[face_mill_1]
   ]).
```

Then the specified value "successes=[face_mill_1]" will override the inherited value.

IV. REPRESENTING PROBLEM-SOLVING KNOWLEDGE

The way that SIPP represents problem-solving knowledge is a bit more unusual. In most knowledge-based problem-solving systems that use frames, problem-solving is done by manipulating the frames using production rules of the form "IF *conditions* THEN *action*." There are two problems with this approach in the domain of process planning. The problems with combinatorial explosion in large rule bases are well known. In addition, the rule-based approach can be somewhat unnatural: someone who is writing down a set of preconditions for an operation such as face milling will probably find it more natural to concentrate on the characteristics that distinguish it from other kinds of milling operations, rather than including characteristics that distinguish it from *every* other kind of machining operation.

To address these problems, SIPP does not use production rules, but instead represents its problem-solving knowledge hierarchically within the frame system. In the example given below, rough_bore is a subtype of bore, which is a subtype of hole_improve_process, which is a subtype of hole_process. This means that rough_bore is applicable for creating a particular surface H only if the restrictions for hole_process, hole_improve_process, bore, and rough_bore are all satisfied by H.

```
type(hole_process, process).
relevant(hole_process, hole).
defaultvals(hole_process, [cost=1]). /* at least the cost of twist_drill */
restrictions(hole_process, H):- . . . various geometric restrictions
type(hole_improve_process, hole_process).
defaultvals(hole_improve_process. [
        precedence=20,
        projected_cost=1, /* at least the cost of twist_drill */
        cost=3 /* at least the cost of rough_bore */
]).
restrictions(hole_improve_process, H):- H?special_features eq none.

type(bore, hole_improve_process).
defaultvals(bore, [cost=3, precedence=22]).
restrictions(bore, H) :- . . . various tolerance restrictions

type(rough_bore, bore).
restrictions(rough_bore, H) :- H?pos_tolerance gte 0.002,
H?neg_tolerance gte 0.002.
actions(rough_bore, P, H) :-
        copy_item(H,G),
```

```
G:diameter ge
G:pos_toleran
G:neg_toleran
G:straightness
G:roundness g
G:parallelism
G:true_positic
G:surface_fini
subgoal(P,G).
```

This example illustra
user to retrieve and modif
and ">=" functions, resp
H:pos_tolerance and H?p
H. The difference betwee
pos_tolerance slot for H dc
but H?pos_tolerance asks
for example, "H:pos_tole
in the item H.

This example also illu
the processes. Suppose SI
by an item frame). If SIPI
then it will create an item
be used to create H. The
(how and when this happ
boring is a process whicl
hole—but the role has to l
it. Since the frame H desc
done, the *actions* statem
which must be present bel
H—but some of G's slot
the purpose of rough bor
roundness, parallelism, e
stringent values than the
increases the diameter of
smaller value than the dia
G must be created befor

The cost slots used in
the processes. Instead, t
process costs and shop pr
one might want to specif
these cost values are used

V. PR

SIPP creates a proces
its machinable surfaces.
least-cost-first Branch anc

verride the inherited

NOWLEDGE

wledge is a bit more
ems that use frames,
roduction rules of the
s with this approach in
inatorial explosion in
ised approach can be
f preconditions for an
ural to concentrate on
ling operations, rather
other kind of machin-

tion rules, but instead
thin the frame system.
, which is a subtype of
ess. This means that
only if the restrictions
_bore are all satisfied

st of twist_drill */
restrictions

drill */

atures eq none.

ons

.002,

```
G:diameter gets H?diameter - 0.005,
G:pos_tolerance gets 1,
G:neg_tolerance gets 1,
G:straightness gets 1,
G:roundness gets 1,
G:parallelism gets 1,
G:true_position gets 1,
G:surface_finish gets 125,
subgoal(P,G).
```

This example illustrates how the frame manipulation language allows the user to retrieve and modify slot values. "eq" and "gte" are similar to Prolog's "=" and ">=" functions, respectively, except that they interpret constructions such as *H*:pos_tolerance and *H*?pos_tolerance as references to the values of slots in the item *H*. The difference between *H*:pos_tolerance and *H*?pos_tolerance is that if the pos_tolerance slot for *H* does not have a value, then *H*:pos_tolerance causes an error but *H*?pos_tolerance asks the user for the value. "gets" is an assignment operator; for example, "*H*:pos_tolerance gets 1" puts the value 1 into the pos_tolerance slot in the item *H*.

This example also illustrates how the user may specify actions associated with the processes. Suppose SIPP is trying to make some hole *H* (*H* will be represented by an item frame). If SIPP decides to consider rough boring as a way to make *H*, then it will create an item frame *P* describing the particular rough_bore process to be used to create *H*. Then the *actions* statement for rough_bore will be invoked (how and when this happens will be described later in another example). Rough boring is a process which can be used to improve various characteristics of a hole—but the role has to be there already in order for rough boring to be used on it. Since the frame *H* describes the hole which is to be present after rough boring is done, the *actions* statement first creates a frame called *G* to describe the hole which must be present before rough boring is done. Initially, *G* is an exact copy of *H*—but some of *G*'s slot values are now changed by the *actions* statement. Since the purpose of rough boring is to improve various surface characteristics such as roundness, parallelism, etc., the *actions* statement sets these slots in *G* to less stringent values than the values that appear in *H*. In addition, since rough boring increases the diameter of a hole, the *actions* statement sets the diameter of *G* to a smaller value than the diameter of *H*. Finally, the *actions* statement tells SIPP that *G* must be created before the rough_bore process *P* can be done.

The cost slots used in the frames above do not denote exact dollar-values for the processes. Instead, they are relative costs which are derived from actual process costs and shop preferences. Depending on the particular machine shop, one might want to specify different cost values than the ones given above. How these cost values are used during problem solving is described in the next section.

V. PROBLEM-SOLVING STRATEGY

SIPP creates a process plan for an object by creating process plans for each of its machinable surfaces. For each machinable surface, SIPP uses a modified least-cost-first Branch and Bound search to find a least-cost sequence of processes for making that surface. This procedure is shown below; the lines are numbered for reference in the text.

```
procedure make(s):
(P1)    ACTIVE := {[$p$,$s$] | $p$ is a process type that is relevant for creating a
        surface such as $s$}
(P2)    loop
(P3)            if ACTIVE is empty, then return failure
(P4)            select $t$=[$p1$, $s2$, $p2$, $s3$, $p3$, ...] as described in the text, and
                  remove it from ACTIVE
(P5)            if the restrictions associated with $p1$ are satisfied then begin
(P6)                    expand $t$ as described in the text
(P7)                    if success has been achieved, then return the suc-
                          cessful plan
(P8)            end
(P9)    repeat
end make
```

As with most Branch and Bound procedures, SIPP uses an "active list" of all alternative process plans being actively considered. Each plan consists of some sequence of processes, along with the surfaces these processes create, culminating in the creation of the desired surface. For example, if we let $t = [p1, s2, p2, s3, p3, s]$, then t represents the following plan:

first use process $p1$ to create a surface $s2$,
then use $p2$ to transform $s2$ into $s3$,
then use $p3$ to transform $s3$ into s.

Each of $p1$, $s2$, $p2$, $s3$, $p3$, and s are represented by frames.

SIPP expands plans backwards from the ultimate goal of creating s, until a complete and successful plan for creating s is found. Since this expansion is done backwards, the processes $p2, p3, \ldots$, in the plan t will have already been completely determined, but the process $p1$ may not yet be completely determined. In this case $p1$ will be a type frame (such as hole_process or mill) that represents some class of processes. If the restrictions specified in the frame for $p1$ are not satisfied, then t has no expansion; otherwise, the expansion of t consists of the set of plans

$$\{[p1', s2, p2, s3, p3, s] \mid p1' \text{ is a subtype of } p1\}.$$

If $p1$ has been entirely determined—and if the restrictions given in the frame for $p1$ are satisfied—then the actions specified in the frame for $p1$ will state one of the following:

(1) that $p1$ is a process that can be performed directly (in which case SIPP has found a complete and successful process plan).
(2) that some other surface must first be present in order for $p1$ to be performed (in which case the processes necessary to create this surface may or may not also be given). For example, if $s2$ is a hole and $p1$ is a

rou
$p1$ (

sucl

SIPP se
determined
lower bounc

LB(t) =

The costs of
completely d
will contain
bound on th
bound is

LB(t) =

SIPP always
To illust
simple know
much simple
frames called
are numbere

(F1)
(F2)

(F3)

(F4)
(F5)
(F6)

(F7)
(F8)

(F9)

rough-boring process, then some hole s1 must already be present so that $p1$ can be done to $s1$ to create $s2$. In this case, the expansion of t is

{[$p0$, $s1$, $p1$, $s2$, $p2$, $s3$, $p3$, s] | $p0$ is relevant for creating a surface such as $s1$}.

SIPP selects plans for expansion one at a time. Which plan is selected is determined by means of lower bound on the costs of the plans. For example, the lower bound on the plan t described above is

LB(t) = the costs of $p2$ and $p3$
+ a lower bound (or the exact cost) of $p1$
+ a lower bound on the cost of any processes that must be done before $p1$.

The costs of $p2$ and $p3$ are given in the slots $p2$?cost and $p3$?cost. If $p1$ is completely determined, then $p1$?cost contain its exact cost; otherwise, $p1$?cost will contain a lower bound on its cost. $p1$?projected_cost will contain a lower bound on the cost of any processes that must be done before $p1$. Thus the lower bound is

LB(t) = $p1$?projected_cost + $p1$?cost + $p2$?cost + $p3$?cost.

SIPP always selects the plan having the least lower bound.

To illustrate the way SIPP's problem solving strategy works consider the simple knowledge base shown below, which contains frames reminiscent of (but much simpler than) the frames actually used in SIPP. This example includes four frames called process, hole_process, twist_drill, and rough_bore. The statements are numbered for reference in the text.

```
(F1)   type(process, thing).
(F2)   slots(process, [[projected_cost, X, number(X)], [cost, X,
       number(X)]]).
(F3)   defaultvals(process, [projected_cost = 0, cost = 0]).

(F4)   type(hole_process, process).
(F5)   relevant(hole_process, hole).
(F6)   slots(hole_process, [[cost, X, number(X)], [projected_cost, X,
       number(X)]]).
(F7)   defaultvals(hole_process, [cost = 0.5, projected_cost = 0]).
(F8)   restrictions(hole_process, H) :-
                 H?contained_in eq Surface,
                 Surface?type eq flat,
                 H?norm eq Hvector,
                 parallel(Hvector,Svector).

(F9)   parallel([Hx,Hy,Hz],[Sx,Sy,Sz]) :- Hx*Sy eq Hy*Sx, Hx*Sz eq
       Hz*Sx, Hy*Sz eq Hz*Sy.
                 /* parallel vectors have proportional components */
```

```
(F10)   type(twist_drill, hole_process).
(F11)   restrictions(twist_drill_process, H) :-
                H?diam gte 0.0625,
                H?diam lte 2,
                H?depth lte 6.
(F12)   actions(twist_drill,P,H) : - success(P).
(F13)   type(rough_bore, hole_process).
(F14)   defaultvals(rough_bore, [
                projected_cost = 0.5, /* at lest as much as twist_drill */
                cost = 1]).
(F15)   actions(rough_bore,P,H) :-
                copy_item(H,G),
                G:diam gets H?diam - (0.01 * H?diam^0.5),
                subgoal(P,G).
```

Statement F6 sets up two slots for hole_process, called *cost* and *projected_cost*. Since twist_drill and rough_bore are both subtypes of hole_process, they also have these slots. The twist_drill frame inherits from hole_process the default slot values given in statement F7, but in the rough_bore frame, these values are superseded by the values given in statement F14.

Statement F5 declares that hole_process is relevant for making holes. However, hole_process will be applicable for creating some hole h only if the restrictions given in statment F8 are satisfied when $H = h$. Since twist_drill and rough_bore are both subtypes of hole_process, they will not even be considered unless statement F8 is satisfied—and furthermore, twist_drill will be applicable only if statement F11 is satisfied.

Suppose we want to create some hole h. The user may either supply the necessary information about h beforehand in the form of an item frame describing h—or else the user may omit this information, in which case SIPP will create an item frame for h as it goes along, asking the user for information about h as needed.

We can start SIPP out by entering the command "make h." This starts SIPP on a least-cost-first search for ways to make h. In statement P1 of the *make* procedure, SIPP initializes the active list. The only kind of process that is relevant for creating h is a hole_process (see statement F5), so SIPP puts onto the active list the plan [hole_process, h]. The lower bound for this plan is

LB(hole_process, *h*])
    = hole_process?cost + hole_process?projected_cost
    = 0.5 + 0.0 = 0.5.

[hole_process, *h*] is the only member of the active list, so in statement P4 it is selected and removed from the active list. In statement P5, SIPP looks at the restrictions for hole_process to see if they are satisfied by h. Thus statement F8 is evaluated with $H = h$. Statement F8 says that h must be contained in a single surface which must be flat, and that the norm of the surface must be parallel to the axis of h.

Suppose these restrictions are satisfied. Then in statement P6, SIPP puts all subtypes for hole_process onto the active list, as alternate plans for creating *h*. From statements F10 and F13, there are two subtypes for hole_process: twist_drill and rough_bore. twist_drill?cost and twist_drill?projected_cost are inherited from hole_process, and thus [twist_drill, *h*] goes onto the active list with

LB([twist_drill, *h*])
= twist_drill?projected_cost + twist_drill?cost
= 0 + 0.5 = 0.5.

[rough_bore, *h*] goes onto the active list with

LP([rough_bore, *h*])
= rough_bore?cost + rough_bore?projected_cost
= 1 + 0.5 = 1.5.

Success has not yet been achieved, so SIPP returns to the beginning of the loop. In statement P4, it removes from the active list the plan having the least lower bound: [twist_drill, *h*].

Suppose the restrictions associated with twist_drill (statement F11) are satisfied by *h*. The twist_drill frame has no subtypes, but it does have actions (statement F12), and these actions are evaluated in statement P8. The "success" predicate in statement F12 tells SIPP that it has found the least-cost way to create *h*: twist drilling at a cost of 0.5.

Suppose, on the other hand, that the restrictions associated with twist_drill are not satisfied—or equivalently, suppose that the user is dissatisfied with twist drilling, and wishes SIPP to look for alternate solutions. Then SIPP returns to the beginning of the loop again, and selects the rough_bore plan (which has 1.5 as its lower bound). If rough_bore had any restrictions, they would be evaluated at this point—and if they failed, SIPP would inform the user that there was no other way to make *h*. However, since rough_bore has no restrictions, SIPP goes ahead and executes the actions given in statement F15, with *P* = rough_bore and *H* = *h*.

Statement F15 sets up the subgoal of creating a hole which is identical to h except that it has a smaller diameter than *h*. In particular, "copy_item(*H*,*G*)" creates an item (say, *h*_1) which is an exact copy of *H*, and binds the variable *G* to *h*_1. Thus "*G*:diameter gets *H*?diameter - (0.01 * *H*/diameter^0.5)" sets the value of the *diameter* slot in *h*_1 to 0.01 times the square root of the diameter of *h*. "subgoal(*P*,*G*)" tells SIPP that the creation of *h*_1 must be accomplished before rough_bore can be performed—so at this point, SIPP needs to consider ways to create *h*_1.*

Just as it did initially for *h*, SIPP puts onto the active list every process type that is relevant for creating any surface type *s* such that *h*_1 is of type *s*. From statement F5, hole_process is relevant to creating holes, so since *h*_1 is a hole, the plan [hole_process, *h*_1, rough_bore, *h*] is put onto the active list with the following lower bound:

*SIPP also allows the user to specify that the creation of of a surface *G* can only be accomplished using some specific process. This can be done by saying, for example, "specific_subgoal(*P*,*G*,rough_face_mill)" rather than "subgoal(*P*,*G*)."

LB([hole_process, *h*_1, rough_bore, *h*])
= hole_process?projected_cost + hole_process?cost + rough_bore?cost
= 0 + 0.5 + 1.0 = 1.5.

This is the only plan on the active list, so it is selected and removed in statement P4. If the restrictions for hole_process are satisifed by *h*_1, then in statement P6 (as was done with *h* earlier), the plans of creating *h*_1 using twist_drill and bore are placed on the active list, this time with the following lower bounds:

LB([twist_drill, *h*_1, rough_bore, *h*])
= twist_drill?projected_cost + twist_drill?cost + rough_bore?cost
= 0 + 0.5 + 1.0 = 1.5

LB([rough_bore, *h*_1, rough_bore, *h*])
= rough_bore?projected_cost + rough_bore?_cost + rough_bore?cost
= 0.5 + 1.0 + 1.0 = 2.5.

SIPP continues in this manner until it finds a successful process plan.

One difference between this example and the real operation of SIPP is as follows: In SIPP's knowledge base, the frame describing rough_bore includes information telling SIPP that rough_bore is not to precede another instance of rough_bore (nor various other processes). Thus SIPP would be intelligent enough not to consider plans such as [rough_bore, *h*_1, rough_bore, *h*].

VI. DISCUSSION

A. Current Status

SIPP is currently up and running as a prototype system whose knowledge base contains about 55 frames; details about its implementation are described in Ref. 18. SIPP can either read prepared data from a file, or (if some or all of this data is omitted) run interactively, asking the user for any needed information. Various user features have been implemented—such as the ability to go back and produce other process plans for a machinable surface if the user wants to see alternatives to the first process plan the system produces.

B. Geometric Reasoning

Currently, SIPP's reasoning capabilities about tolerance requirements are reasonably good—but (as with other existing process planning systems) its geometric reasoning capabilities are quite limited. For example, when drilling a hole or creating a pocket, SIPP may pay attention to the surface *s* that contains the hole or pocket, but does not look at any other surfaces of the object. Thus if the object is shaped in such a way that the surface *s* is inaccessible, SIPP will not realize this.

The addition of sophisticated geometric reasoning capabilities is a major future goal. This goal will require the integration of SIPP with a solid modeling system, the development of a feature extraction system to obtain relevant surface features from the solid modeling system, and the solution to various problems in subgoal interaction in the creation of machined surfaces. We are already doing research related to some of these problems.[20,12]

Even th
mate goal, tl
with forward
and if those
would be dor
modification.
into a forwar
Howeve:
SIPP. Earlier
rule-based sy:
domain, and
systems that

Our initi
it appears qui
knowledge h
static inform
hole_process
properties of
properties are
There ha
implemented
capabilities—
frame system
manner—but
suppose hole.
spade_drill. S
Since hole_pr
which is a hol
below, as the
whenever a sl
contain inform
and from whe
second-genera

In order
forward-chaini
evaluating the
If the system
blowup," in w
tions of each
OPS-5[10] and Y

C. Hierarchical Knowledge Representation

Even though SIPP's expansion of plans proceeds backwards from the ultimate goal, the way expansion is done is closely related to forward chaining. As with forward chaining, one first evaluates the restrictions associated with a frame, and if those restrictions are satisfied, one then either evaluates its actions (as would be done in a forward-chain system) or considers its subtypes. With suitable modification, the problem-solving knowledge used in SIPP could probably be put into a forward-chaining rule-based system as OPS-5[10] or YAPS.[1]

However, there are certain advantages to the hierarchical approach used in SIPP. Earlier, we mentioned that SIPP's approach addresses two problems with rule-based systems: the unnaturalness of production rules in the process planning domain, and the problems with "exponential blowup" that occur in rule-based systems that contain lots of rules. We now discuss these issues in more detail.

1. Naturalness of Representation

Our initial impression of SIPP's hierarchical representation technique is that it appears quite natural to use: Hierarchical representation of the problem-solving knowledge has many of the same advantages as hierarchical representation of static information. For example, by declaring class of processes such as hole_process or face_mill to be a subtype of some other class, one can use all of the properties of the larger class without having to remember explicitly what those properties are.

There have also been a few problems in using the frame system as it was implemented in SIPP, and these have given us ideas for possible extensions to its capabilities—particularly in the ways that inheritance is done. As with most other frame systems, SIPP frames inherit properties from above in a depth-first manner—but this type of inheritance is not always general enough. For example, suppose hole_process is a type with two different subtypes: twist_drill and spade_drill. Suppose further that twist_drill?cost = 1.0 and spade_drill?cost = 1.5. Since hole_process?cost is supposed to be a lower bound on the cost of anything which is a hole_process, one would want to synthesize hole_process?cost from *below*, as the minimum of {*P*?cost | *P* is a subtype of hole_process}. In general, whenever a slot value is not explicitly given, it would be useful for the frame to contain information telling exactly how that particular value should be inherited, and from where. One of our students is currently putting this feature into a second-generation version of SIPP which is being written in Lisp.[11]

2. Exponential Blowup

In order to decide which rules are applicable to a given problem state, forward-chaining production rule systems normally look at each of their rules, evaluating the preconditions of the rules to determine which rules are applicable. If the system contains many rules, this can cause a problem with "exponential blowup," in which large amounts of time are expended evaluating the preconditions of each of the rules. This problem has been alleviated in systems such as OPS-5[10] and YAPS[1] by providing ways to determine whether a rule is applicable without having to reevaluate all of its preconditions each time around—but if the system contains a large number of rules, exponential blowup will still occur. Another approach, which is used in KEE,[9] is to provide facilities whereby the user can divide a set of rules into smaller subsets such that each subset *S* is relevant for a different domain *D*.* Given a problem to solve, the system first determines which problem domain the problem is in, and then it uses the rule set *S* for that domain, ignoring all the other rules. But if *S* is a large set, exponential blowup can still occur..

Suppose a rule-based system is trying to solve a problem in some problem domain *D*, and suppose *S* is the set of rules for *D*. Each time the system applies a rule, this will change the system's current state *C*. The problem of exponential blowup would be greatly alleviated if the system could always tell from the description of *C* that only some small subset S_C of the rules in *S* were relevant to *C*—for then the system could temporarily ignore the other rules in *S*. The knowledge representation scheme used in SIPP provides a convenient and natural way to do this.

Finding the set rules relevant to the current state of a rule-based system corresponds in SIPP either to (1) retrieving the subtypes of some process type, (2) retrieving all process types relevant to the creation of a surface type (this occurs when a subgoal statement is encountered), or (3) retrieving a single specific process type (when a specific_subgoal statement is encountered). In each case, only a few of SIPP's frames are relevant—and the information about which frames are relevant is given explicitly in SIPP's frame system. Unfortunately, Prolog's way of retrieving information from its database does not take advantage of this information to prevent exponential blowup—but a second-generation version of SIPP is being written in Lisp which will take advantage of this information.

D. Pros and Cons of Using Prolog

SIPP is currently implemented in C-Prolog. At this point, we are building a second-generation version of SIPP in Franz LISP, and intend to do further work using the LISP version instead of the Prolog version. Below we discuss some of the considerations that went into this decision—both the advantages and the disadvantages of Prolog in comparison to Lisp.

For the most part, we did not make use of Prolog's backtracking capabilities—but intead used Prolog as a general-purpose AI language in which we wrote our own control strategies. This was surprisingly easy for the most part—often easier than it would have been in Lisp. The ability to bind variables using unification made for particularly nice coding in some cases. In addition, interpreted C-Prolog code runs as fast in some cases as the equivalent compiled Franz Lisp code!

Programming in Prolog does not appear to us to be "logic programming." Certainly, Prolog programs for various theoretical problems may be considered to be logic programs, but practical problems require non-logical predicates such as

*By "problem domain," we simply mean some class of problems.

repeat, *read*, *write*, *assert*, *retract*, *fail*, and the cut—as well as Prolog's left-to-right evaluation of clauses. Without these features, Prolog would not be a particularly useful programming language—and in fact, many of Prolog's current drawbacks are because these features do not always give sufficient functionality. Below are the main problems we encountered with Prolog:

(1) Prolog allows no way to change the value of a variable once it is bound. Although there are often ways to achieve the same effect in other ways, this limitation sometimes proves very inconvenient.

(2) Prolog has no direct way to handle loops. If this can be handled by replacing a loop iteration with a backtrack or a recursive call, things usually work out rather well—but this approach is not always reasonable. The only other alternative is a *repeat-fail* loop—which loses the variable bindings every time the loop begins its next iteration.

(3) Prolog is a poor language for writing large systems because there is no way to set up subroutine packages in which only certain routines are available to the user. Instead, all Prolog definitions are global. Until recently, most Lisp implementations had this same drawback, but recent Lisp implementations (such as the latest release of Franz Lisp) provide the user with ways to define packages.

(4) Students who were originally trained to program in a conventional programming language such as Pascal seem to find it easier to learn Lisp than Prolog—mainly because Lisp is less of a departure from conventional languages than Prolog is.

(5) Even with the above problems, Prolog is a sufficiently nice language that we still would have considered doing further work on SIPP in Prolog, were it not for the need to integrate SIPP with solid modeling and graphics software. This would require interfacing Prolog to subroutines written in other languages. This feature is not available in the Prolog system which we have, and we are told that it does not work very well in Prolog systems which do have it.

Probably most of these problems are present simply because Prolog is an immature language. Certainly the early implementations of Lisp were rather primitive capabilities when compared with the Lisp systems that are available now. We hope and anticipate that Prolog will evolve in such a way that many of our current criticisms of it will become irrelevant.

References

1. E.M. Allen, "Yaps: Yet Another Production System," Tech. Rep. TR-1146, Computer Science Department, University of Maryland, College Park, MD, February 1982.
2. G. Boothroyd, *Fundamentals of Metal Machining and Machine Tools*. Scripta, Washington, DC, 1975.
3. T.C. Chang, "Interfacing CAD and CAM—A Study of Hole Design." M.S. thesis, Virginia Polytechnic Institute, 1980.
4. T.C. Chang, "The Advances of Computer-Aided Process Planning," Tech. Rep. National Bureau of Standards, Gaithersburg, MD, October 1981.

5. T.C. Chang and R.A. Wysk, "An Integrated CAD/Automated Process Planning System," *AIIE Transactions*, **13**, (September 3, 1981).
6. T.C. Chang and R.A. Wysk, "Integrating CAD and CAM through Automated Process Planning." *International Journal of Production Research*, **22**(5) (1985).
7. Y. Descotte and J.C. Latombe, "GARI: A Problem Solver that Plans How to Machine Mechanical Parts," *Proc. Seventh International Joint Conf. Artif. Intel.*, August 1981.
8. M.S. Dunn, Jr., J.D. Bertelsen, C.H. Rothauser, W.S. Strickland, and A.C. Milsop, "Implementation of Computerized Production Process Planning," Rep. R81-945220-14, United Technologies Research Center, East Hartford, CT, June 1981.
9. R. Fikes and T. Kehler, The Role of Frame-Based Representation in Reasoning, *Comm. ACM*, **28**(9), 904–920 (1985).
10. C.L. Forgy, "The OPS5 User's Manual," Tech. Rep. CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1980.
11. M. Gray, Master's thesis, Computer Science Department, University of Maryland, 1985, in progress.
12. S. Joshi and T.C. Chang, "Feature Extraction and Recognition for Automated Process Planning," Work in progress, 1985.
13. K.M. Kahn, *Intermission—Actors in Prolog*, UPMAIL, Uppsala University, Uppsala, Sweden.
14. C.H. Link, "CAPP—CAM-I Automated Process Planning System." *Proc. 13th Numerical Control Society Annual Meeting and Technical Conference*, Cincinnati, OH, March 1976.
15. W.S. Mann, Jr., M.S. Dunn, and S.J. Pflederer, "Computerized Production Process Planning," Rep. R77-942625-14, United Technologies Research Center, November 1977.
16. K. Matsushima, N. Okada, and T. Sata, "The Integration of CAD and CAM by Application of Artificial-Intelligence, *CIRP*, 329–332 (1982).
17. D.S. Nau, "Issues in Spatial Reasoning and Representation for Automated Process Planning," *Proc. Workshop on Spatial Knowledge Representation and Processing*, May 1982. (By invitation; not refereed.)
18. D.S. Nau, "SIPP Reference Manual," Tech. Rep. 1515, Computer Science Department, University of Maryland, 1985.
19. D.S. Nau and T.C. Chang, "Prospects for Process Selection Using Artificial Intelligence." *Computers in Industry* **4**, 253–263 (1983). (Also available as Tech. Rep. TR-1268, Computer Science Department, University of Maryland.)
20. D.S. Nau, D. Jones, and K. Masai, "Converting Boundary Surface Representations Into Constructive Solid Geometry Representations, *IEEE Trans. System, Man. and Cybernetics*, 1985, to appear.
21. N.J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.
22. A.A.G. Requicha, "Representations for Rigid Solids; Theory, Methods, and Systems," *Computing Surveys*, **12**(4), 437–464 (1980).
23. TNO, *Introduction to MIPLAN*, Organization for Industrial Research, Inc., Waltham, MA, 1981.
24. H.B. Voelcker, A.A.G. Requicha, E.E. Hartquist, W.B. Fisher, J. Metzger, R.B. Tilove, N.K. Birrell, W.A. Hunt, G.T. Armstrong, T.F. Check, R. Moote, and J. McSweeney, "The PADL-1.0/2 System for Defining and Displaying Solid Objects." *ACM Computer Graphics*, **12**(3), 257–263 (1978).
25. P.M. Wolfe and H.K. Kung, "Automating Process Planning Using Artifical Intelligence," in *1984 Annual International Industrial Engineering Conference Proceedings*, Chicago, IL, 1984.
26. R.J. Wood, "Franz Flavors: An Implementation of Abstract Data Types in an Applicative Language," Tech. Rep. TR-1174, Computer Science Department, University of Maryland, College Park, MD, June 1982.
27. R.A. Wysk, "An Automated Process Planning and Selection Program: APPAS," Ph.D. thesis, Purdue University, West Lafayette, IN, 1977.

Toward a (
with Uncer
and Fuzzin

Ronald R. Yager
Machine Intelligenc

We describe the theor
that in the face of p
appropriately extendi
show that these two th

The constructic
sophisticated mecha
tion. At least three
role in these types
uncertainty), appea
narrowed down to a
This is manifested b
20 and 30. A very pc
a set, more precisel
situations in which
element. This is ma
A very powerful too
first type of uncerta
in which we know t
clearly delineate bo

The third type
variable assumed i
random experimen
used in many case
other two forms of

The theory of
fuzzy subsets and
theory of evidence