MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

WORKING PAPER 276 AUGUST 1985

IDEME: A DBMS OF METHODS

Jintae Lee

Abstract. In this paper, an intelligent database management system (DBMS) called IDEME is presented. IDEME is a program that takes as input a task specification and finds a set of methods potentially relevant to solving that task. It does so by matching the task specification to the methods in its database at multiple levels of abstraction. After isolating potentially useful methods, IDEME ranks them by how relevant they might be to the task. From the most relevant method, it checks if its operational demands, i.e. those conditions that have to be satisfied for the method to be applicable, are satisfied by the present task. If so, it presents the algorithm of the method relativized to the present task; otherwise, it goes on to the next method. In this paper, the focus will be on the representation scheme that is used by IDEME to represent methods as well as tasks.

Table of Contents

1. Introduction

1.1. IDEME

In this paper, an intelligent database program called IDEME[1] is presented. IDEME takes as input from the user a task specification and provides as output a set of methods that might be relevant to solving the task. It does so by matching the task specification at multiple levels of abstraction to the methods in its database. After finding such methods, IDEME orders them according to how relevant they might be to the given task. Then for each method, starting from the one judged most relevant until the user finds one he wants or it runs out of the methods to test, the program verifies its operational demands-- i.e. the conditions that have to be satisfied for that method to be applicable by asking the user if those conditions can be satisfied by the task in hand. If so, its algorithm is presented in the context of the present task together with some concrete examples of how that algorithm is actually used.

If the method thus selected is only a reference frame, i.e. one that does not give you an actual algorithm but only gives pointers to other potentially applicable methods, then the user would follow the path that it provides. Otherwise, the user would have several choices at this point:

1. He can look at the algorithm and decide to try it. In that case, IDEME will remember the point at which he leaves so that in case he does not find the method acceptable he can come back and pick up where he left off.

2. He can look at the algorithm but decide that he wants to look at the other methods that have been selected, in which case he will just remember the name of the present method (in case he wants to come back to that method later) and just tells IDEME to proceed with the next method.

3. Or he may decide to modify the task specification on the basis of suggestions the present method makes, in which case he will go back and edit the original task specification and start over at an appropriate point.

---

[1] The term 'ideme' is philosophers' name for the unit of ideas as gene is for the unit of genetic combination.

1.2. Motivation

A couple of motivations led to this work. First, there is the appreciation that ideas are not utilized as much as they could be. People come up with ideas-- many of them powerful ideas-- but often they are buried in the original context in which they first appear and not used again. When other people need the ideas, they don't know where to find them. They have to go through the primitive search process using keywords and even then have to go through a lot of filtering processes before actually finding something that they can use. In artificial intelligence, the situation is slightly better because the value of modularity has been widely recognized in the field. There is a tendency among the researchers to isolate ideas when they can in a form that is easily transportable and modifiable. Also some have made clear the concepts such as methods, task space, operational demands, and so on. which are useful in defining units of ideas.[2] However, I believe we can go further and create a database of methods and its management system that will help us find and adapt ideas when we need them. IDEME is such an attempt. It can also be viewed as a step toward establishing the science of design that Simon envisages in [Simon 81].

Another motivation is the desire for a database management system where the units are much larger and less structured than those in the conventional database management systems. There are many instances in AI research where the retrieval problem comes up. For example, in Winston's analogy program [Winston 80, Winston 82] there are many precedents from which we want to select one that matches most closely the description in hand. Here the database consists of precedents which cannot be characterized in terms of a fixed number of fields. Furthermore, we would like to represent precedents not at any one level of abstraction but at multiple levels so that precedents, eg. stories, can be matched gradually from top down or bottom up as the case may require. In IDEME, we present a scheme for a database management system whose database consists of structural units, namely methods, which can be represented at multiple levels of abstraction.

---

[2] In particular, Newell and his associates have made a large contribution in this matter by not only formulating the concepts such as those mentioned above, but also by actually identifying a set of 'weak' methods and the conditions under which they are applicable. [Laird&Newell 83], [Laird 84]

1.3. Overview

In the following section (Section 2), we give a scenario that illustrates a possible user interface with IDEME. It should give some idea of what IDEME is supposed to do and what the user can gain from it. The next two sections give more detailed description of the organization and the algorithm of IDEME, respectively. The section on the organization (Section 3) is divided into five subsections. The first (3.1) discusses the philosophy and the motivation behind the three main structures used in IDEME, namely System Objects, Task Specification, and Method Representation. The next three sections describe them respectively. In the section on algorithm (Section 4), we give a semi-formal algorithm for the retrieval of relevant methods for a given task specification. In the next section (Section 5), we discuss the present state of the research as well as the reasons for our choice of the domain and the performance of IDEME at this stage. In the last section (Section 6), we conclude by discussing the things that need be done, possible extensions of IDEME, and other potential applications of the concepts introduced here.

2. Scenario

Suppose you are interested in the Programmer's Assistant project -- especially the analysis aspect, i.e. recognizing a given code as an instance of some higher level plan or algorithmic cliche[3] [Rich&Shrobe 78, Rich 81, Waters 82]. You decide to consult IDEME for finding useful methods; so you walk up to a terminal and start talking to it.

[Notation: * is the prompt for user input; @ marks the beginning
of the system response; also the actual texts that the user types
are underlined.]

* Hello, IDE.
;;;IDEME recognizes who you are and sees if there
;;;are any files containing previous interactions. If there is, it
;;;starts from there. In this example, there wasn't any previous interaction. So it
;;;starts from the beginning.

@ Hello, Jin.
I am here to help you find potential methods for the task you

[3] Programmer's Assistant is an automatic programming assistant program at MIT that helps the programmer to document, analyze, implement, verify, and debug programs.

want to solve. There are two modes available. You could type in as much information as you want in the template for the task that will be provided and let me find a set of possibly useful methods; or you could specify a method and work with that method. Type T if you want the first mode; type the name of the method if you want the second. You could start with the first, and then change to the second later when you get some idea of which method you want to try.

* T
@ Below, you will be asked to characterize the task in hand in terms of input, output, and constraints. Output is what you want to achieve, Input is what you have available, and Constraints are the relations that should hold between the input and the output. Think about what is the type of the object that is given as input, what you want as output, and what the relations are that should hold between the input and the output objects; and write them down in appropriate places.
Unless stated otherwise, you are expected to answer with one of the objects whose types are recognized by the system (the list can be seen by typing the key, LIST; if you do not find an object type that you want, you can define your own type. See the manual for the procedure.) Whenever you specify one of these objects, you will be asked a series of questions. In specifying an object, use the following convention: Name Of the Object: Object Type (eg. problem space:SYSTEM). But feel free not to answer if you find the question irrelevant in specifying the task in hand. In that case, just press carriage return.

| INPUT: ProgramDescription: DESCRIPTION
| ;;Input is a description of the program (eg.
| ;;sentences such as "It has a loop").

| ProgramDescription: DESCRIPTION
|
| @ What does it describe?
| ;;The answer fills in OF slot of DESCRIPTION.
| * Program: STRUCTURE
| @ What language is the language used to describe it?
| ;;The answer enters in slot of DESCRIPTION.
| *
| ;;The user need not fill in a slot if he thinks it is
| ;;not relevant.

| Program: STRUCTURE
|
| @ What are the objects that make up the

| structure?
| * ProgramConstructs: LIST
| ;;LIST is a system object to be used when there
| ;;are more than one type of object.

| ProgramConstructs: LIST
|
| @ How many different type of elements are there
| in the list? (Type a number, 'Finite', 'Infinite',
| or 'Unknown')
| * Finite
| ;;This value enters the Cardinality slot of LIST and
| ;;used in determining which questions to ask further.
|
| @ What are some of the element types in the list?
| *
| ;;Again the user feels it is not important. Since
| ;;there was no information supplied at this level,
| ;;;IDEME stops nesting and goes back to the higher
| ;;;level that still has any unfilled slot.

| @ What are the important relations that exist
| among the objects? Use your own terms to
| describe them.
| * Ordered, Nested
| ;; cf. Section 3.2 for how this
| ;; information is used.

|
|
|
|

| OUTPUT: PlanDescription: DESCRIPTION

| PlanDescription: DESCRIPTION
|
| @ What does it describe?
| * Program ;;; Program here refers to the same program
| which appeared as an answer for ProgramDescription.
|
| @ What is the language that is used to describe it?
| * PlanCalculus: LANGUAGE

| PlanCalculus: LANGUAGE.

```
      |
      | @ What type of Alphabet does the language have?
      | *
      |
      | @  What type of Formation Rules does the language have?
      | *
   |
   |
|
|
```

| CONSTRAINTS: Describe (PlanDescriptions, Structures-Of(Program))

;;;IDEME searches the database and finds a set of potentially
;;;useful methods. The selection algorithm will be described in
;;;detail in Section 4 (Algorithm).
;;;Now we assume that IDEME has found WSFD as the method judged
;;;most relevant to the specified task.

@ Given the task specification you provided, WSFD[4]
seems the most relevant method. If you want a description of the
method, type DESCRIBE; otherwise type GO, then I'll start
checking if the method can in fact be applicable to the present
task.

* DESCRIBE

WSF is a method which, given an object or its structural description, can produce its functional description, identify what kind of entity it is, learn from precedents rules to be used for such descriptions or identifications in the future. WSFD is the submethod which takes as input structural description of object (eg. "The object X has a handle, a broad base, and empty concave space.") and produces its functional description (eg. "It is stable, liftable, and can contain something inside." Or combining these properties, WSF can assert that the object is a cup.)

WSFD works the following way. When structural description is given, it checks to see if there are rules in its database whose precondition match some of the descriptions. If so, it asserts the functional descriptions which are in the action part of those rules. For example, a rule might be "If X has a broad base, it

---

[4] It stands for the Description submethod of Winston's Structure/Function method. [Winston 83]

is stable." Then given the description, "It has a broad base.", it asserts "It is stable." It does so for all the rules whose preconditions are satisfied.

If there are no such rules, then it looks for precedents that contain some of the structural descriptions given in the input, If not found, WSF fails. If found, it checks if these precedents relate some structural description that appear in the input to a functional description (for eg. a precedent involving a suitcase might have a sentence such as "The suitcase is liftable because it has a handle", which relates a structural description "X has a handle" to a functional description "X is liftable".) Assert the functional descriptions thus related.

Given the functional descriptions thus asserted, WSFD looks up its database containing rules relating functional descriptions to other functional descriptions to see if further or more concise descriptions are available. For example, there might be a rule such that "If X is stable, liftable, and can contain things, then it is a cup." If so, one function of the object can be 'being a cup'. Collect all the final functional descriptions and produce them as the output.

If you want more information, see Winston,P.H., T.O.Binford, B.Katz, and M.Lowry. "Learning Physical Descriptions from Functional Definitions, Examples, and Precedents." National Conference on Artificial Intelligence, Washington,DC. 1983.

*GO

@ Matching at the top level for: INPUT.

Programmer's Assistant: WSFD
analysis

DESCRIPTION: DESCRIPTION:
ProgramDescription ObjectDescription

@ Matching at the second level for: INPUT[DESCRIPTION], ProgramDescription::ObjectDescription

Of: STRUCTURE STRUCTURE
Program Object

In: PROGRAMMING LANGUAGE

@ Matching at the third level for: INPUT[DESCRIPTION.OF], Program::Object

Objects: LIST
Relations: Ordered, Nested

@ Matching at the top level for: OUTPUT

OUTPUT DESCRIPTION DESCRIPTION
PlanDescription FunctionalDescription

@ Matching at the second level for: OUTPUT[DESCRIPTION]
PlanDescription::FunctionalDescription

OF. Program - Object

;;IDEME knows that these two objects, Program and Objects,
;;have been matched before. So it simply prints the names
;;without their type. Also if the match here is inconsistent
;;with the existing bindings, then IDEME prints the message
;;to that effect.

IN. LANGUAGE
Plan Calculus.

@ Matching at the third level for:
OUTPUT[DESCRIPTION.OF] Program::Object
;;IDEME knows from the previous match that there
;;are no further matches at this level. So it
;;stops here.

@ The binding so far produced is the following:
(ProgramDescription::ObjectDesription
Program::Object
PlanDescription::FunctionalDescription)
Now I will check if the operational demands of WSFD can be satisfied.

@ Can you have Rules relating ProgramDescription and PlanDescription? If so, give me an example.
* "If it has a loop in which there is COND clause, it is Iterative Filtering."
;;;The user does not have to come up with examples immediately. He can quit and think about it. When he comes back, IDEME will remember where he left off.

@ Can you have 'Functional Definitions' which relate a set of Plan Descriptions to a Plan Descriptions? If so, give me an example.
* "If it has trailing-search and iterative-generation, then

assert "It has trailing-generation + search."

@ Can you have 'Precedents' which include sentences relating program description and plan description? If so, give me an example.
* the case of iterative generation:
"... It does iteration because it has the loop structure and the variable is being modified inside the loop. ..."

@: It seems that the method WSFD can be useful to the present task. Try the following algorithm.

1. Given ProgramDescription, first search for Rules whose condition part satisfies the descriptions.
2. If there isn't any, go to 4.
3. If there are some, assert the PlanDescriptions associated with the result part of the Rules.
4. See if there are any Precedents which include sentences that relate some of the ProgramDescription and a PlanDescription. If so, assert the PlanDescription. Repeat until there aren't any more Precedents with such sentences.
5. See if there are Functional Definitions whose conditions are satisfied by the PlanDescriptions so far asserted. If there aren't any, the set of PlanDescriptions so far collected will be FunctionalDescriptions of the Program. If there are some, assert PlanDescription in the result part of the rules.
Repeat until there are no more Functional Definition whose conditions are satisfied.

Eventually, we would like IDEME to interface with problem solvers in such a way that the problem solver can take over and tackle the task from this point on. But for now we concentrate on developing a representation and an organizational structure that will accomplish the above task.

3. Organization

3.1. Philosophy

There are three main structures in the organization of IDEME: system objects, task specification, method representation. Each of them will be described in detail in the following sections. Here, we give a picture of the overall organization by discussing the motivations and assumptions underlying these structures and how they interact. The overall structure of IDEME is based on a few basic assumptions. The first is what we might call the philosophy of frame-- namely

the assumption that describing an object means filling in the slots that is associated with the object. The slot association may be explicit or implicit.

Slots for a given object may be numerous and hierarchical (eg. for an expert) or sparse (eg. for novices) and unorganized. Slots for the same object may be different depending on the contexts in which they are looked at. Nevertheless, what one tries to do in describing an object is to identify the slots associated with the object and fill them in with appropriate values as much as possible in that context. This frame assumption, in one form or another, underlies many of the work in artificial intelligence. For example, it is the basis for many representation languages in AI such as KRL, FRL, and KLONE. In IDEME, it pervades the entire organization. The following are the examples:

* A task specification consists of filling in the following slots:
    - Input, which describes what is given.
    - Output, which describes what you want to accomplish
    - Constraints, which describes constraints that should hold between Input and Output.
    - Context, which describes the context that the task is situated in. If the context value is specified by the user, then only the methods with that context value will be retrieved.
* The representation of a method consists of filling in the following slots:
    - Application Conditions, which describes the conditions which are likely to call for the present method. This slot in turn consists of the same three slots as those of task specification : Input, Output, Constraints.
    - Context, which describes the context that the method is presupposing.
    - Operational Demands, which describes the conditions which must be satisfied for the method to be applied.
    - Strengths, which describes the strengths of the method.
    - Limitations, which describes the weaknesses of the method.
    - Algorithm:
* Each of the system object, which together form a vocabulary to be used in representing tasks and methods, has associated with it a supertype slot and a number of other slots. For example, the type SYSTEM has a supertype slot, whose value is OBJECT, and two

additional slots, STATES and OPERATORS, indicating that describing a system consists of describing what the states of the system are and what operators are available to manipulate them. Each of the slot has a restriction on what type of objects can fill in the slot. For example, State slot of the object above, SYSTEM, can have as its value only an object of type STATE or its supertype. The object STATE has in turn, of course, has its own slots-- namely, Supertype, Objects, and Relations, indicating that describing a state consists of describing what objects there are in the state and what relations hold among them. Also the user defines a new object by specifying its slots and relating them to the slots of its supertype object.

Another principle that plays an important role in IDEME is abstraction. Abstraction technique is used in at least two major ways in IDEME. First, the structure of task as well as method is characterized in terms of a few limited number of abstract concepts which is known to the system. For example, a possible structure of generic search task, (i.e. a task structure one might specify if what one is interested in is search) is: INPUT: SYSTEM(Problem Space), STATE(Initial State), STATE(Final State) and OUTPUT: SEQUENCE[OF:OPERATOR (Legal Operators)]. This abstraction eliminates a lot of arbitrariness in the task specification and the method representation, thereby facilitating matching. For example, if we are given a task whose type is search, then there would be only a few possible specifications of that task, one of which will be the above. Then the methods which can help searching can be retrieved by matching the above structure to their application condition. If there were no abstraction, there would be so many possible ways to specify a task that the match would become impossible. The list of the system objects so far defined are given in the following section. We are aware that any fixed set of concepts would eventually prove inadequate for representing the various tasks and methods that we might want IDEME to accommodate in the future. Thus, we provide a means for the user of extending them so that the list can be modified with respect to the individual users.

The other way that abstraction is used in IDEME is through allowing multiple levels of description via what we call slot expansion. As described above, each system object has slots associated with it. Each of these slots takes as its value a system object, which has its own slots, which can then have as its value a system object, and so on. As we saw in the scenario, this expansion allows us to describe the structure of a task or a method at multiple levels of abstraction

without giving way to arbitrariness. Through this multiple abstraction, not only matching is facilitated but also a task can be described in as much detail as one thinks relevant.

Given these structures, the overall picture of IDEME will be the following. The user comes in and types the specification of the task he wants to solve. As we discussed above, the specification consists of the values for the three slots, Input, Output, Constraints; and the objects that will appear as values of the slots will be one of the system objects, which in turn will have their own slots. Each of these slots will then be filled by a system object as its value, and so on until to the level that the user judges to be relevant. In the database, each method is represented in terms of the following slots: Input, Output, Constraints, Context, Operational Demands, Strengths, Limitations, Algorithm. When a task is specified, the first three slots comprising the application condition are matched hierarchically with those of the task[5] Given a task specification, IDEME starts matching from top down-- i.e. first looks for those methods with the application condition that matches the task specification at the most abstract level and then continue, among the methods thus selected, matching down as far as it can go to look for methods which match the task specification at as detailed level as possible. The selected methods, once found, will be ranked and, from the one judged most relevant, the operational demands of each method will be verified by interfacing with the user. If the operational demands of a method is satisfied, IDEME presents the algorithm relativized to the present task, i.e. the algorithm whose objects have been changed to the appropriate objects in the task domain. If there are many methods whose conditions are satisfied, then the Strengths and Limitations slots will be presented to the user so that he can select which one to try first. If no method is found or if none of the methods found is applicable, then IDEME searches for those methods whose application conditions match with a task description that is generalized from the original task specification by ignoring the most detailed level in the slot expansion hierarchy, and so on. A fuller description of the algorithm is given in Section 4.

---

[5] unless the context slot was specified, in which case the methods selected are restricted to those with the same value in the context slot.

3.2. System Objects

3.2.1. Principles

System objects are those that the system knows about.[6] Their purpose is to reduce arbitrariness by providing a vocabulary for specifying tasks and methods, to facilitate matching, and to allow slot expansion, which in turn allows multiple level representation and matching. An object is a system object if it is in the initial set of the objects that the system knows about or it is an object defined by the user in the manner described below. There is intrinsically no distinction between the initial objects and the user-defined objects. Once an object is defined, it becomes a system object. However, any object being defined is related to the objects initially known to the system via supertype relation. The initial objects have been chosen because they are the most general object types that are frequently used to characterize the AI tasks and methods at non-trivial levels of abstraction.

Each of the system objects has associated with it a number of slots. For example, the type SYSTEM has as its slots Supertype, States, Operators, meaning that describing a system consists of describing what its states are and what operators are possible to manipulate them. When a user finds that there are no system object with which he can adequately characterize a task or a method, he can create new objects on top of the system environment or one of the environments that may have been built so far and kept in the library [Goldstein&Bobrow 81]. The user creates a new object in the following way.

1. The user specifies the name and the supertype of the object being defined. The supertype has to be one of the objects that the system already knows about.

2. The user then proposes a set of new slots for the object being defined. Each of the new slots is either an additional slot in addition to the slots of its supertype or it has to be related to the slots of its supertype (eg. by means of restriction, transformation, etc.). In any case, all the slots of its supertype has to be accounted for. That is, every one of the supertype slots should be either associated with a slot of the object being defined or it should be filled with the constant NR (Not Relevant), meaning that it is not important to know the value of that slot in the context of the new object.

There is a partial ordering among the system objects imposed by Supertype relation. If A is a

---

[6] System objects should be really called system object types because it is really the types not the actual objects that instantiate them that we are talking about in this section. But we omit that distinction for brevity in what follows.

supertype of B, then A dominates B. And for any slot for which A can enter as a value, so can B. The list of the initial objects as well as some examples of user-defined objects are given below together with the slots associated with each.

The list is not complete and subject to modification. It will be finalized after we try representing an extensive number of methods and tasks.

3.2.2. Initial Objects

The format for describing a system object will be as follows:

```
Name of the Object
Supertype Slot: Supertype Object ;;which is the supertype of Object.
        Slot 1 (of Supertype Object): X
            ;;where X is the name of the slot of Object
            ;;to which this slot is related to or the
            ;;constant NR (standing for Not Relevant).
        .
        .
        Slot n (of Supertype Object): X'

Slot 1 (of Object): [Y ],
    ;;where Y is the type of objects that can fill this slot.
.
.
Slot m (of Object): [Y']
```

Note: Below it is assumed that if Y is the type of object that can enter as a value for the slot S, then a set of Y, SET(Y) also can as well as any one of its supertype object. If it is necessary to specify that only a singular object of type Y, as opposed to a set of Y, can be the value of S, then it will be prefixed by "!" in front like !Y.

1. OBJECT

    Supertype: OBJECT

2. SYMBOL

    Supertype: OBJECT

3. BOOLEAN PREDICATE

    Values:[0,1] ;;VALUE slot is used when one wants to specify

;;fixed set of subtypes of this type.
Supertype: SYMBOL

4. NUMBER

Supertype: SYMBOL

5. NAME

Supertype: SYMBOL
Of: [OBJECT] ;what it is a name of.

6. STATE

Supertype: OBJECT
Objects: [OBJECT]
Relations: [RELATION]

7. PROCEDURE

Supertype: OBJECT
Before: [OBJECT]
After: [OBJECT]

8. SYSTEM

Supertype: OBJECT
States: [STATE]
Operators: [PROCEDURE]

9. SET

;;;SET is a structure in the context of set theory.

Supertype: STRUCTURE
Objects:: Elements
Relations: NR
System: ;Set-Theory (cf.Section 3.2.3).
Elements: [OBJECT]
Cardinality: [NUMBER]

10. CONSTRAINTS

Supertype: RELATION
On: [SET]
In: [LANGUAGE]

11. STRUCTURE

Supertype: OBJECT
Objects: [OBJECT]
Relations: [RELATION]
Systems: [SYSTEM] ;System is the framework in which Objects and Relations
;are identified or interpreted.

12. LIST

;;;LIST is different from SET in allowing its members to be
;;;of different types.

Supertype: STRUCTURE
Objects: Elements
Relations: Ordered
Systems:
Cardinality: [NUMBER]
Element1: [OBJECT]
Element2: [OBJECT]
.
.
ElementN: [OBJECT]
;;where N is the cardinality specified.

13. SEQUENCE

Supertype: STRUCTURE
Objects: Elements
Relations: Sequenced
Systems:
Elements: [OBJECT]

14. GRAMMAR

Supertype: RULE
Condition: LeftSide
Result: RightSide
LeftSide: [OBJECT]
RightSide: [OBJECT]

15. LANGUAGE

Supertype: SYSTEM
  Objects: Alphabet
  Relations:
  Operators: FormationRules
Alphabet: [SYMBOL]
FormationRules: [RULE]

16. LOGIC

Supertype: LANGUAGE
  Alphabet: Alphabet
  FormationRules: FormationRules
Alphabet: [SYMBOL]
FormationRules: [RULE]
Derivation Rules: [RULE]
Axioms: [SEQUENCE(Elements:SYMBOL)]

17. TREE

Supertype: STRUCTURE
  Objects: Nodes
  Relations: Parent-Of, Child-Of
  Systems: [SYSTEM]
Nodes: [OBJECT]

18. GRAPH or NET

Supertype: STRUCTURE
  OBJECT: Nodes
  RELATIONS: Links
  SYSTEM: ;Graph Theory
Nodes: [OBJECT]
Links: [RELATION]

19. MAP

Supertype: PROCEDURE
  Before: Domain
  After: Range
Domain: [SET]
Range: [SET]

20. RULE

Supertype: PROCEDURE
Before: LeftSide (or Condition)
After: RightSide (or Result)
LeftSide: [OBJECT]
RightSide: [OBJECT]

;;;When the leftside type and rightside type are fixed, RULE
;;;is interchangeable with MAP by treating PRECONDITION TYPE as
;;;DOMAIN and ACTION TYPE as RANGE.

21. OPERATOR

Supertype: PROCEDURE
Before: Domain
After: Domain
Domain: [SET]
Arity: [NUMBER]

22. FUNCTION

Supertype: MAP
Domain: Domain
Range: Range
Domain: [SET]
Range: [SET]
;;;FUNCTION is a MAP where one to many mapping is
;;;prohibited.

23. RELATIONS

Supertype: OBJECT

We find it more difficult to come up with the initial set of relations. One possibility is the list of very basic and syntactic relations such as: Equality, Symmetry, Transitivity, Reflexivity, etc. But they seem too low level relations to capture the level we want to deal with. Another possibility is to look into linguistics for help. People have identified a list of "primitive" cases that can be construed as relations. For example, [Green ?] lists thirteen categories of relations (such as: Actants, Spatial Localization, Temporal Localization, Belonging, etc.) that are purported to express all the other relations. Or [Langacker 83] claims that all the relations reduce to four basic ones (Inclusion,

Separation, Identity, Association). Leaving aside the legitimacy of their claims, they seem still not at the right level of abstraction. Yet another possibility is an empirical list, i.e. a list of relations that come up often in the domain of AI-- eg. Membership, Inclusion, Satisfaction, Derivation, Equality, Relation, Identification, Match, Interpretation, Inheritance, etc. This probably is the most useful way, but we need some way of systematically selecting them.

At the moment, we use thesaurus approach. We let the user to describe the relations in his own terms, provided that only a single term is used for a relation. Then in matching we consider as match any other terms which are synonymous to the term chosen by the user and has the same types as well as the same number of arguments. For example, CHARACTERIZE(X,Y) would be considered as matching DESCRIBE(W,Z) if the types of X and W, Y and Z are compatible. We also use the thesaurus approach for matching the values for Constraints slot. That is, in describing the relation between input and output, we allow the user to use any phrase he wants and then we determine the matches by referring to a thesaurus.

3.2.3. Examples of User-Defined Object

1. MATHEMATICAL SYMBOL

   Supertype: SYMBOL

2. PROGRAMMING LANGUAGE

   Supertype: LANGUAGE
      Alphabet: Alphabet
      Grammar: Grammar
   Alphabet: [SYMBOL]
   Grammar: [GRAMMAR]

3. DESCRIPTION

   Supertype: STRUCTURE
      Objects: NR
      Relations: NR
      System: IN
   Of:[OBJECT]

In: [LANGUAGE]

4. TASK

Supertype: STRUCTURE
  Objects: Objects
  Relations: Relations
  Systems:
Objects: [OBJECT]
Relations: [RELATION]
Input: [OBJECT]
Output: [OBJECT]
Operators: [PROCEDURE]
Constraints: [OBJECT]

5. THEORY

Supertype: SYSTEM
  Objects: In.Alphabet
  Relations: NR
  Operators: In.FormationRules
Of: [OBJECT]
In: [LANGUAGE]

6. SET THEORY

Supertype: THEORY
  Of: [SET]
  In: In
In: [LANGUAGE]

3.3. Task Specification

A task specification consists of filling in the following slots:

* Input, which describes what is given.

* Output, which describes what you want to accomplish

* Constraints, which describes constraints that should hold between Input and Output.

* Context, which describes the context that the task is presupposing. This slot is used to filter only those methods that are specifically used in the specified context.

Some examples are shown below, but we should note that there is no one way to represent a task. The same task can be represented in several different ways depending on perspectives. The implication of non-canonicity in representation will be discussed in the discussion section.

PA Analysis:

```
Input-- ProgramDescriptions:DESCRIPTION
        Of: Program:STRUCTURE
            Objects: ProgramConstructs:LIST
            Relations: Ordered, Nested
            System: -
        In: PL:PROGRAMMING LANGUAGE
            Alphabet: -
            FormationRules: -
Output-- PlanCalculusDescriptions: DESCRIPTION
        Of: Program
        In: PlanCalculus: LANGUAGE
            Alphabet: -
            FormationRules: -
```

Learning a concept from training instances:

```
Input-- TrainingInstances:SEQUENCE
                Element: AnInstance:LIST
                        Cardinality: 2
                        Element1: Sample:PATTERN
                        Element2: TF:BOOLEAN
Output-- Concept:DESCRIPTION
            Of: X:OBJECT
Constraints-- IF TF = T in AnInstance, THEN Instantiate(Sample,X);
        IF TF = F, THEN Not(Instantiate(Sample,X)).
;;; IF, THEN, and a few other constructs are available for
;;; expressing constraints.
```

3.4. Method Representation

The representation of a method consists of filling in the following slots:

* Application Conditions, which describes the conditions which are likely to call for the present method. This slot consists of the following same three slots as those of task specification : Input, Output, Constraints.

* Operational Demands, which describes the conditions which must be satisfied for the method to be applied.

* Context, which describes the context that the method presupposes.

* Strengths, which describe the strong points of the method compared to the other methods with the same application condition.[7]

* Limitations, which describe the weak points of the method.

* Algorithm:

The following are a few examples of method representation. Please note that there are usually more than one way to represent the same method and the examples below illustrate only some ways the methods can be represented.

WSFD: Winston's Structure/Function Description:

```
Application Condition::
   Input-- ObjectDescriptions: DESCRIPTION
            Of: X:OBJECT
   Output-- FunctionalDescriptions: DESCRIPTION
            Of: X ;i.e. X specified above.
   Constraints--
Operational Demands::
   * Can you have RULE's relating Objectdescription
   and FunctionalDescriptions?
   * Can you have 'Functional Definitions' which
   relate a set of FunctionalDescriptions to a FunctionalDescriptions?
   * Can you have 'Precedents' which include
   information relating Objectdescription and FunctionalDescriptions?
Strengths:
Limitations:
Algorithm: ;;The algorithm is given in the scenario, so
      ;;omitted here.
```

MEA: Means Ends Analysis:

```
Application Condition::
   Input-- Given:LIST
            Cardinality: 2,
            Element1: InitialState:STATE,
            Element2: GoalState:STATE
   Output-- Solution:SEQUENCE
            Elements: Ops:OPERATOR
   Constraints-- Solution (InitialState) = GoalState
   ;;i.e. SEQUENCE of OPERATORS that represent Solution when applied
   ;;to the initial state should yield the goal state.
Operational Demands::
   * Can you identify the types of differences between
```

[7] This slot, as well as the next-- Limitations, is used to help the user choose among the methods that seem equally applicable.

InitialState and GoalState?
* Can you identify operators that will reduce the differences?

Algorithm::

1. Make a table that associates with each type of difference all the operators that reduce it.
2. Let SOLUTION be initially a null list and let CurrentState be InitialState.
3. Find the difference between CurrentState and GoalState.
4. If there is no difference, then we are done because the current state is the goal state. Return SOLUTION.
5. If there is a difference, look up in the table all the operators which reduce the difference identified and let OQ be a queue containing all the operators found.
6. Until OpQ is empty,
    6.1 Let CurrentOperator be the first element in OpQ.
    6.2 If CurrentOperator is not applicable to CurrenState, then try to satisfy the preconditions of Current Operator by using Means Ends Analysis (with InitialState bound to CurrentState and FinalState bound to the precondition state of CurrentOperator) so that CurrentOperator will become applicable.
        6.3.1 If we succeed, then append the result to SOLUTION.
        6.3.2 If we fail, then pop OpQ.
    6.3 If CurrentOperator is applicable to CurrentState, apply it. Append CurrentOperator to SOLUTION and let CurrentState be the new resulting state. Go to 3.
7. If we get to this stage, then we fail.

4. Matching Algorithm

We assume in the following algorithm that the database is organized and indexed hierarchically by the object types of the subslots of the application condition slot of the methods. That is, at the top level, the database is partitioned by the equivalence relation "having the same object type for the slot INPUT". Then each equivalence class is further partitioned by the relation "having the same object type for the slot OUTPUT". Each of the resulting equivalence classes is further partitioned by the relation "having the same object types related by the same types of relations for the slot CONSTRAINTS". Then the same process is repeated with the slots next level down. Depending on the subslots of ObjectType for INPUT slot, the database is partitioned by the relation "having the same object type for the first subslot of INPUT slot", and so on. This relation is well-defined because the methods in the database in consideration by now should all have the same object type, thus the

same subslots, for INPUT slot. This organization is useful for narrowing down the search space of methods as the matching proceeds, as we will see below.[8]

```
Let LIST be null.
        ;;LIST is the list of methods that have been
        ;;successfully matched to TaskSpecification

Find-Methods (TaskSpecification, Database, 1)

Procedure Find-Methods (TS, Db, Level#)
;;;Find-Methods takes as arguments a task specification, a database
;;;of methods, and a level number, which indicates the level (in methods
;;;representations) at which the matching should be done
;;;and returns a list of methods that matches the task
;;;specification ordered by how well they match, the best match the first.

    Let the object types for the slots of TaskSpecification at
    level L be S1, ..., Sn.
    Let Q be the queue (S1, ..., Sn)
    Let Slot# be 1.
    Let RD be Restrict-Database (Pop(Q), Db, Level#, Slot#)
    ;;RD is the list of methods that match the task up to the slot#
    ;;at level Level#.
    If RD is NIL,        ;if no match
     Then Return LIST.    ;then stop matching and return what you have.
     Else If Empty (Q),   ;if all the slots at the level have been
                ;matched,
        Then append RD to the front of LIST
                ;then include the methods so far survived in
                ;LIST
          Until there is no Si which has further subslots,
            L# <- Level# + 1      ;and go on to the next level
            Find-Methods (TS, RD, L#);and do the same.
         Else S# <- Slot# + 1 ;if slots are still left at this level,
             Restrict-Database (pop(Q), RD, Level#, S#)


Procedure Restrict-Methods (ObjectType, DB, Level#, Slot#)
;;;Restrict-Methods takes as arguments an object type, a database
;;;of methods, a level number, and a slot number which indicates
;;;the position of the slot with which ObjectType should be
;;;matched, and returns the class of methods whose
;;;representations match that of the task up to that level.

    Let RD be the subclass of DB which is defined by ObjectType for the
    slot# slot at level# level.
```

[8] The algorithm below is very tentative. It is presented to give an idea in a semi-formal way. I would appreciate bug reports and suggestions.

```
If RD is Null, ; If there is no match with the given object type,
 Then Let Supertype (ObjectType) be ST.
          ;; try its supertype.
      If ST is NIL,   ;If there is no further supertype,
       Then Return NIL ;let them know the match failed.
       Else Relate-to-Supertype (ObjectType, ST)
           ;; Procedure Relate-to-Supertype, omitted here,
           ;; does the necessary bookkeeping in using the
           ;; supertype in place of ObjectType-- such as
           ;; transforming the slots of ObjectType into
           ;; those of its supertype, and adjusting the
           ;; slot # accordingly.
          Restrict-Database (ST, DB, Level#, Slot#)
           ;; then do the same with its supertype.
Else Return RD.
```

5. Discussion

At the present, only some twenty methods have been encoded into the database. Some methods like Means-Ends Analysis are very general but others like Zeroing Coefficient are specialized in the sense that for it to be applicable to a task, that task has to satisfy many conditions such as having to be a description in mathematical symbols including variables with coefficients. Many methods have been extracted from the AI research on learning.[9] There are Winston's methods for structural learning, Michalski's AQ11 [Michalski&Ryszard 80], Lenat's AM [Davis&Lenat 82] , and Langley and Simon's BACON [Langley 83], and Mitchell's version space [Mitchell 77], and others. Also there are some methods like the search methods whose type is not strictly learning. They are there because they are general methods which appear as components in learning methods.

So far, the experiment with IDEME has been done only manually and it has been limited due to the few number of methods in the database as well as the limited collection of system objects. But what IDEME does show with the tested cases so far is promising. For example, we tested IDEME the following way. When representing methods into the database, we also kept a list of the original tasks for which the methods were used. Then we had some other people specify those tasks to see if the

---

[9] We chose the domain of learning as the initial experimental ground because I have been interested in learning and also because there are several tasktypes in the domain (such as rote learning, learning from examples, analogical learning, learning by being told, and learning multiple concepts) so that I can test the capacity of IDEME to differentiate and/or relate the different but similar tasks and methods.

list of potential methods returned by IDEME would at least include the original methods used for the task. In most cases, which include tasks like "learning the concept 'arch'" (for Winston), "learning to classify soybean diseases" (for Michalski), "learning the concept of 'flush' in poker" (for Mitchell), "learning new concepts" (for Lenat, and for Langley), it did provide the method that was originally used for as well as others.

However, there were a few cases where the methods have not been selected for the task for which they should have been. For example, the weak methods such as means-ends analysis or the various search methods should have been selected for most tasks because they are very general methods, but often they were not chosen by IDEME. The reason is that for those methods to be applicable a particular perspective is needed in interpreting the task structure. In the case of the above examples, the perspective needed is what Newell calls Problem Space perspective. However general that perspective may be, it does require that one adopt that perspective instead of others. Although the use of system objects and multiple level description help reducing the arbitrariness in representing tasks and methods, it cannot force a perspective on the user. It cannot guarantee unique description because usually there are inherently more than one way of looking at things. We can look at the same task or method from many perspectives and characterize it in several different ways involving different system objects. For example, we may characterize the task of learning various soybean diseases as taking a sequence of training instances (TrainingInstances: SEQUENCE [Elements: TrainingInstance:LIST [Cardinality: 2, Element1: Symptom:PATTERN, Element2: Teacher:BOOLEAN-PREDICATE]])[10] as input and producing, as output, rules relating the symptoms and the disease names (Rules: RULE [DOMAIN: Symptom, RANGE: DiseaseName:NAME]). But we may also characterize it as a means-ends analysis task which takes as input the initial and the desired states of the system that we want to teach (Givens:LIST [Cardinality: 3, Element1:SYSTEM, Element2: InitialState:STATE, Element3: FinalState:STATE]) and produce as output a sequence of operation (Solution: SEQUENCE[Elements: Op:OPERATOR]) that will transform the initial state to the final state.

---

[10]The notation used to specify an object is: (ObjectName:ItsType [Name-of-Slot1: { the specification of the object that enters this slot}, ...]).

Often which perspective or which system objects to use for representing a given task is determined by the slots associated with the system objects. A sentence could be a PATTERN or a DESCRIPTION, but if we are treating it as a semantic object then we may want to specify what it refers to. If we want to do so, we better characterize it as a DESCRIPTION because it is the one that has the slot OF that will allow the specification of the reference. Nevertheless, it remains that these guides are not enough to make the representation unique. And if the same task or method can be represented in several ways, then there is the problem of possibly missing out potential methods just because the perspective in which the method was represented differs from that for the task. At the present, we deal with this problem by representing tasks and methods in all the possible perspectives we can think of, thereby making them robust. Furthermore, we let the system know how to transform certain representations into other alternative representations. For example, IDEME can have the knowledge that when everything fails, it can try to transform the present task specification into the problem space formulation and then retrieve the weak methods whose application conditions match with the resulting task specification. However, we can go further by having IDEME learn the possible relations among the perspectives through precedents and use that knowledge to attempt the transformation. But at the moment, it is only a plan among many that need be given more thoughts. I discuss others in the next section.

6. Further Research

6.1. To be done

There is much to be done before IDEME gets actually implemented. First of all, the system objects have to be given more thoughts. We have to think about whether there can be some principled way of choosing the initial set of objects, whether we can come up with a better way to deal with relations as well as continue testing with the objects we already have to see if they are adequate to represent what we want to represent.[11] Also, I would test IDEME by continuing to build the

[11] We are exploring the use of DL, an object and procedure description language designed by Winograd [Winograd 83] and/or of KANDOR, a representation language by [Patel-Schneider et. al. 84, Patel-Schneider 84], for our purpose.

database and trying out tasks against it. The tasks will be selected so that they will test the different facets of IDEME. For example, those tasks which are known to be analogous or solvable by the same method can be used to see if the methods retrieved by one are retrieved by another. While I will continue to focus on the domain of learning as the source of methods, I will test the generalizability of IDEME by trying to characterize the tasks and methods in other domains as they come across. Since none of IDEME's structures except the actual list of system objects should depend on the domain chosen, I believe generalization should present no problem. The system objects even as they are now seem very general that they can encompass a pretty wide range of domains.

6.2. Implementation

We are thinking of implementing IDEME within the paradigm of message passing semantics such as Hewitt's ACTOR system. The scheme might take something like the following form. Each method would be represented as an actor residing in the database. The user would specify the task as a set of messages. These messages would go to an actor, say Referral Specialist, which knows to whom, i.e. to which methods, to forward these messages. Then, those methods who think they can handle the specified task report themselves with some sort of matching score. Then another actor, Selector, would choose a few methods most likely to succeed and allocate resource. The chosen actors would then try to satisfy the operational demands associated with them and in doing so generate questions. The questions would be forwarded to Question Handler who will classify and organize them in an order that is best to be presented to the user. The answers will be received in an interactive session with the user, and returned to the actors which originated them. Using the actor scheme would provide the advantages that are inherent in object oriented, message passing systems such as modularity, distributedness, extendability, and parallelism [Hewitt&deJong 82].

6.3. Extensions

There are many ways that IDEME can be extended. A fully automatized version might be something like this. IDEME takes a task description in natural language, abstract what is relevant, and translate the description into a formal specification. Given the task specification, IDEME would check and see whether it would be better to divide the task into several subtasks. It may do so by looking up a library which contains suggestions for doing that or by having an expert rule system. For each task or subtask specification, IDEME would check and see if that specification can be re-presented from alternate perspectives. If so, the search is made with the alternative specifications as well. For each specification, it looks for the methods in the manner described in this paper. After finding a method for each subtask, it would synthesize them so that they can be combined to produce the original output from the original input. Eventual automatization may even hope for automatic interface with actual problem solving programs; what is needed as input for those programs may be supplied by IDEME on the basis of the information obtained from the user and once it is obtained, the actual program that implements a method can take over and produce the desired result without the need for human intervention. But that is a story not easily to come in the immediate future.

There are many places where learning can enter. Analogy can be used to find methods for a task when the normal search through the database turns out to be not successful. That is, if no useful method is found for a given task, that task can be matched with precedents, i.e. the tasks that have been solved before. If there is a task that matches closely, then the methods that have been used to solve that task may be adapted to the present task. Doing so opens many learning possibilities in other areas as well. Once matched tasks are found, their task specifications can also be matched to learn from them how to formulate alternative task specifications, how one system object or a group of system objects may be viewed as another system object, or how one task might be profitably divided into several tasks. IDEME can store this knowledge for use in the manner described above.

Not only can IDEME be helped by analogy but it can also facilitate analogical reasoning itself. It does so by virtue of using the similar representations for tasks and methods. Note that the slots for a

task specification is a subset of the slots for a method representation. In that sense, the method representation is just an extension of the task specification. That is, it consists of a few additional slots (OPERATIONAL DEMANDS, CONTEXT, STRENGTHS, LIMITATIONS, ALGORITHM) on top of those of the task specification (INPUT, OUTPUT, CONSTRAINTS, CONTEXT). It means that when a task is solved, we could turn it into a method by adding an algorithm slot among others and filling it with the algorithm that was used to solve the task. It facilitates analogy because when we later face another task with similar structure, it will have a similar task specification. So it will be matched with the task-turned method, and the associated algorithm will be available for the present task. Note that matching, which is the hardest problem in analogy, is tamed here much via our use of system objects and multiple level abstraction.

Acknowledgment:

This research was motivated by the following work:

* Winston's work on analogy, on structure and function, and on learning.

* Newell's weak methods, functional reasoning; Laird and Newell's universal weak methods, universal subgoaling as well as other work of Newell's on the foundation of AI.

* Simon's proposal on the science of design.

In addition, the following ideas have influenced this research:

* Semantic primitives in Schank, Rieger, and others.

* Abstractions in GPS and in Kling's ZORBA.

* Hewitt's ACTOR system.

Finally,I thank the following people for saving us a lot of trouble by providing the necessary material in readily available form:

* Edward Feigenbaum, Avron Barr, Paul Cohen, and others for the Handbooks of AI, v.1-3.

* Winston for *Artificial Intelligence*, 2nd ed.

* Newell and Laird for the catalogue of the weak methods.

References

[Davis&Lenat 82]
Davis, Randall & D. B. Lenat, editors.
Knowledge-Based Systems in Artificial Intelligence.
McGraw-Hill Book Company, New York, 1982.

[Goldstein&Bobrow 81]
Goldstein, Ira P. & D. Bobrow.
Layered Networks as a Tool for Software Development.
In *Proceedings of the Seventh IJCAI.* August, 1981.

[Green ?]
Green ?
Essays in Lexical Semantics.
In ?. ?, ?

[Hewitt&deJong 82]
Hewitt, Carl & P. deJong.
Open Systems.
AIM 691, MIT, 1982.

[Kling 71]
Kling, Robert.
A Paradigm for Reasoning by Analogy.
Artificial Intelligence , 1971.

[Laird 84]
Laird, John.
Universal Subgoaling.
CMU-CS-84 129, Carnegie Mellon University, 1984.

[Laird&Newell 83]
Laird, John & A. Newell.
A Universal Weak Method.
CMU-CS-83 141, Carnegie Mellon University, 1983.

[Langacker 83]
Langacker, Ronald W.
Foundations of Cognitive Grammar.
Indiana University Linguistics Club, Bloomington, Indiana, 1983.

[Langley 83]
Langley, Pat, G. L. Bradshaw, & H. A. Simon.
Rediscovering Chemistry with the BACON System.
Tioga Publishing Company, Palo Alto, CA, 1983, .

[Michalski&Ryszard 80]
Michalski, Ryszard S. & R. L. Chilausky.
Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis.
International Journal of Policy Analysis and Information Systems 4(2), 1980.

[Mitchell 77]
Mitchell, Tom M.
Version Spaces: A Candidate Elimination Approach to Rule Learning.
In *Fifth International Joint Conference on Artifical Intelligence*, pages pp.305-310. Cambridge, MA, 1977.

[Patel-Schneider 84]
Patel-Schneider, Peter F.
Small can be Beautiful in Knowledge Representation.
FLAIR Technical Report 37., Fairchild Artificial Intelligence Laboratory, 1984.

[Patel-Schneider et. al. 84]
Patel-Schneider, Peter F., H.J. Levesque, and R.J.Brachman.
ARGON: Knowledge Representation meeets Information Retrieval.
In *First Conference on Artificial Intelligence Applications.* IEEE Computer Society, Denver, Colorado, 1984.

[Rich 81]
Rich, Charles.
A Formal Representation for Plans in the Programmer's Apprentice.
In *IJCAI-85.* August, 1981.

[Rich&Shrobe 78]
Rich, Charles & H. Shrobe.
Initial Report on A Lisp Programmer's Apprentice.
IEEE Transaction on Software Engineering 4(5), November, 1978.

[Rieger 76]
Rieger, Chuck.
On Organization of Knowledge for Problem Solving and Language Comprehension.
Artificial Intelligence (2), 1976.

[Schank 72]
Schank, Roger C.
Conceptual Dependency: A Theory of Natural Language Understanding.
Cognitive Psychology (4), 1972.

[Simon 81]
Simon, Herbert A.
The Sciences of the Artificial, 2nd ed..
MIT Press, Cambridge, MA, 1981.

[Theriault82 82]
Theriault, D. G.
A Primer for the Act-1 Language.
AIMemo 672, MIT, 1982.

[Waters 82]
Waters, Richard C.
Programmer's Apprentice: Knowledge Based Program Editing.
IEEE Transaction on Software Engineering 8(1), January, 1982.

[Winograd 83]
Winograd, Terry.
Language as a Cognitive Process: Syntax.
Addison-Wesley Publishing Company, Reading, MA, 1983.

[Winston 80]
Winston, Patrick H.
Learning and Reasoning by Analogy.
Communication of the Association for Computing Machinery 23(12), 1980.

[Winston 82]
Winston, Patrick H.
Learning New Principles from Precedents and Exercises.
Articial Intelligence (3), 1982.

[Winston 83]
Winston, Patrick H., T. O. Binford, B. Katz, & M. R. Lowry.
Learning Physical Descriptions from Functional Definitions, Examples, and Precedents.
In *National Conference on Artificial Intelligence*. 1983.
Washington, D.C.