Approaches to Knowledge Acquisition:

The Instructable Production System Project

Michael D. Rychener

29 December 1981

Carnegie-Mellon University
Department of Computer Science
Schenley Park
Pittsburgh, PA 15213

(submitted to Carbonell, Michalski, & Mitchell, Eds., *Machine Learning*)
(published by Tioga, Palo Alto, CA)

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

ABSTRACT

In building systems that acquire knowledge from tutorial instruction, progress depends on determining certain functional requirements and ways for them to be met. The Instructable Production System (IPS) Project has explored learning by building a series of experimental systems. These systems can be viewed as being designed to explore the satisfaction of some of the requirements, both by basic production system mechanisms and by features explicitly programmed as rules. The explorations have brought out the importance of considering in advance (as part of the kernel design) certain functional components rather than having them be filled in by instruction. The need for the following functional components has been recognized:

- interaction language;
- organization of procedural elements;
- explanation of system behavior;
- accommodation to new knowledge;
- connection of goals with system capabilities;
- reformulation (mapping) of knowledge;
- evaluation of behavior;
- and compilation to achieve efficiency and automaticity.

Since the experimental systems have varied in their effectiveness, some general conclusions can be drawn about relative merits of various approaches. Seven such approaches are discussed here, with particular attention to the three whose behavior can be most effectively compared, and which reflect the temporal development of the project.[1]

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

1. The Instructable Production System Project

The Instructable Production System (IPS) project [22] started out in the fall of 1975 to study the construction and behavior of large-scale systems of production rules. Our hypothesis, extrapolated from work in cognitive psychology [17], was that intelligence would result as a system grew in size, from an ability to deal with more situations and to apply more knowledge to solve problems. The motivation to use production systems had the same source [17]. To increase the scientific interest of building such systems, and ultimately to improve the chances of continuing growth and viability, it was stipulated from the start that the system was to be built by gradual "instruction" rather than by deliberate programming.[2] The research evolved into a series of explorations of the design of a starting system (Kernel), from which the much larger system would be grown. The explorations spanned a four-year time period, until mid-1979, and involved the efforts of over a half dozen people[3].

The setting in which instructional experiments took place was chosen to be one of "learning by doing". In this paradigm, the instructor of the system watches and advises the system while it is solving problems in its chosen domain of expertise (cf. the work of Anzai and Simon [3]). This is a good way to study learning because it combines attributes of both learning by being told and learning by independent exploration, while avoiding some of their drawbacks. That is, the instructor still instructs by telling, but the fact that the system is doing something at the same time allows the instructor to verify (partially) that new knowledge is appropriate to the system's current knowledge. In addition, the system is in a sense exploring in an environment that has new situations for it, under the guidance of the instructor and in the framework of problems posed by the instructor. When new knowledge interacts in some way with the system's existing knowledge, that interaction has the

---

[2] Actually, production systems are quite difficult to program, so an instruction mode has the potential of bringing a large system into the realm of feasibility. What is desired is that the production system itself be able to manage its knowledge, find interactions of new knowledge with old [19], check consistency, formulate and select answers for questions that arise when new and old knowledge statements are compared, and do assorted other tasks that can't even be predicted at this time. To do this knowledge management task would require a great deal of knowledge itself, and the IPS project has only begun to realize what might be required for this much larger research goal.

[3] See the Acknowledgments section near the end of the paper.

greatest chance of being understood in the context of a situation where that knowledge is being applied. The system is forced to deal with new situations in its own way, using its own conceptual system, with the extra help of the instructor's advice. But advice to the system is often limited, in that the system's knowledge may not be stored so as to be brought to bear in all appropriate situations, and in that the instructor can often see only the effects of the knowledge, rather than the knowledge itself, depending on how well the system can describe itself.

More precisely, the dialog between instructor and system is ruled by a number of constraints:

- The instructor of the system gains all information about IPS by observing its interactions with its environment (including the instructor).
- The dialog takes place in (restricted) natural language.
- The dialog is mixed initiative, with both participants free to try to influence the direction.
- Instruction may be about any topic or phenomenon in the system's external or internal environment (subject to the other restrictions).
- Knowledge accumulates over the lifetime of the system.

These constraints are intended to embody the essence of instruction as it occurs in a number of natural situations. At the same time, they tend to rule out explicit "programming" by the instructor, and thus place a larger burden on the system's learning abilities, and indeed on its general intelligence.

Throughout the IPS experiments, the underlying knowledge organization was ***Production Systems*** (PSs) [6, 26, 1, 20, 17], a form of rule-based system in which learning is formulated as the addition to, and modification of, an unstructured collection of production rules. As mentioned above, this assumption of architecture has some support from psychological theory [17]. Behavior is obtained through a simple ***recognize-act cycle*** with a sophisticated set of principles for resolving conflicts among rules [13, 21]. The dynamic short-term memory of the system is the ***Working Memory*** (WM), whose contents are matched each cycle to the conditions of rules in the long-term

memory, ***Production Memory***. As will be explained in a later section, information transfer from the environment (including instructor) to the system takes place by depositing conventionalized symbol structures into the WM. Those structures then become subject to manipulation by the system's procedural methods expressed as rules (to be defined and illustrated after the next subsection). The IPS project developed several dialects of the OPS language [6, 8, 24] to support its experiments.

1.1. Relation to Other Learning Research

In terms of a model recently proposed for learning systems by Buchanan, *et al.* [4], the IPS work focused on certain aspects of the learning problem while neglecting others. Their model consists of:

- A performance module that actually performs tasks.
- A critic that evaluates performace, locates errors, and recommends corrective actions.
- A learning module that responds to the critic by modifying performance.
- An instance selector that poses training problems.
- A blackboard [12] for globally modifiable data and inter-module communication.
- A world model for domain-specific knowledge and assumptions.

In all of the IPS explorations, both performance and learning modules were embodied in the Production Memory, and were thus intermixed. This paper is concerned principally with elaborating and refining the subcomponents of these two modules. This emphasis is inherent to the instructional situation, where the instructor plays the role of critic and instance selector. WM functioned as the blackboard, and world-model knowledge (usually minimal) was represented as rules whose actions placed facts into WM and otherwise maintained consistency with the domain's assumptions.

To further the comparison of the IPS project with other artificial intelligence and psychology research, it is useful to discuss briefly our position with respect to a number of current issues. The topic of instruction for an IPS can be characterized as

- self-contained procedures for specific tasks,

- problem-solving operations within such procedures,
- and domain-specific heuristics, in the same context,

rather than such things as

- rules or heuristics that work only within the computational context of a special-purpose control structure or mechanism different from the recognize-act paradigm of PSs (as in various "expert" systems, e.g., those for medical diagnosis),
- causal models for explanation and prediction (as in attempts to model physical devices, Socratic tutoring approaches, etc.),
- concepts (as in various pattern-classification and concept-formation studies),
- language grammars,
- and numerical functions and relationships.

Thus, the IPS work was not concerned primarily with such mechanisms as generalization, specialization, discrimination, property-intersection, rule induction, and pattern induction. These mechanisms were considered to be second-order refinements[4] on what we gave an IPS by instruction, so in fact we expected them to become more relevant as the basic problems with IPS were solved and the system began to exhibit coherent and interesting task behavior. Also, they are mechanisms that are best applied when much larger quantities of empirical data or knowledge are involved. In other words, the emphasis was on the gradual transfer of knowledge from instructor to system, and our focus remained the structure and contents of a body of knowledge, and its effective use to obtain behavior. This is in contrast to having the system develop the knowledge from general axioms, from knowledge primitives, or from large bodies of unstructured facts, which would involve abstract manipulations, inductions, and searches. These would reduce the amount of interaction, and would require more searching and intelligence on the part of the system. They would take place in large spaces that would be distant from instructional and interactive situations, and thus hard to formulate heuristics for. Similarly, because of our limited understanding, we neglected such issues as credit and blame assignment, convergence of learning over time, speed of convergence, and

[4]This is not to say that they are second-order in all knowledge domains and studies, but just in our narrow focus. It is a matter of *relative* importance.

searching as an alternative to direct instruction. In fact, PSs as an architecture are amenable to a number of interesting operations with regard to the above-mentioned topics, leaving open many research avenues.[5]

To state the matter more positively, learning in an IPS was by accumulation of fairly specific rules and methods. In many cases, the rules acquired could be viewed within some well-known organization such as means-ends analysis or schemas, but usually this organization was not obtained from an act of specializing or instantiating an existing general knowledge structure. Rather, as discussed in later sections of this paper, either the instructor or the system was oriented towards maintaining a particular organization on the specific knowledge that it received. The IPS work has a closer kinship to studies in intelligent computer-aided instruction and perhaps in educational psychology (particularly programmed learning) than to other attempts at learning systems. (This kinship will be discussed further in the concluding section.) There is also a strong relation to the construction of "expert" systems, involving accumulation of a body of specific domain knowledge. More relationships are disussed at the end of the next subsection.

1.2. Basic Definitions and Discussion

There are a few key concepts whose definitions will clarify some issues with respect to the IPS project's approach to encoding knowledge. These also reveal a position on planning and other control structure topics.

A ***goal*** is a data structure that represents an external command, an internal need to achieve some state or a need to execute successfully some sequence of actions. An example, taken from a simulated manufacturing domain, is:

```
Make a car for a customer's order
```

where the customer's order is another data structure describing details of the item to be made. In the

[5] Anderson's 1982 paper, in this collection, addresses such topics.

OPS3 [24] dialect of OPS, this might be represented as

```
ws011: (make car goal (order ws014))
ws014: (customer order data (type car) (body sedan) (color blue)
                            (engine-size medium)
                            (accessories (radio a/c)))
```

These structures consist of an internal name, a three-element header, and then a set of attribute-value pairs, where the value may be a set of items. Details of this and other representations used within various versions of IPS are beyond the scope of this paper. For the remainder of the introductory examples that follow, a liberal English translation is used for readability.

A ***rule*** (i.e., a production) in OPS consists of a number of conditions and a number of actions. Each condition is a pattern that matches some element of WM, such as a goal (in various states of activation: active, suspended, succeeded, failed), a structure describing something perceived in the environment, or a data structure describing some internal state. The actions of a rule typically assert new data structures or goals, and can also modify or delete existing structures.

A ***method*** in IPS is a set of rules that work together to satisfy a goal. It is typically very specialized to a certain goal class, and usually consists of a number of steps, with various intermediate data generated to indicate the progress towards completion. The following is a method for satisfying the above sample goal. It is not meant to reflect accurately all of the details of actual IPS methods, but just the general flavor of the approach.

```
M1: If there is a goal to make a car for a customer's order
    and the order specifies the car's body as some type,
then have the goal to make a body of that type for the car.

M2: If there is a goal to make a car for a customer's order
    and the order specifies an engine of some size for the car
    and the car's body has been made,
then have the goal to install an engine of that size in the car.

M3: If there is a goal to make a car for a customer's order
    and the order specifies accessories
    and the car's engine has been installed,
then know that the car is ready for accessories.

M4: If there is a goal to make a car for a customer's order
```

```
    and the order specifies a radio
    and it is known that the car is ready for accessories,
then have the goal to install a radio in the car.

M5: If there is a goal to make a car for a customer's order
    and the car has a body as specified in the order
    and the car has an engine as specified in the order
    and the car has all of its accessories installed,
then know that the goal to make a car has been satisfied.
```

The first two rules, M1 and M2, generate subgoals for doing specific subtasks of the main goal. The completion of one subgoal, in this method, triggers the rule that generates the next. The rule M3 recognizes some conditions signifying a certain stage in the method's progress, and summarizes that in a new data structure, so that later rules in the method don't need to make tests that are overly specific or detailed or that would multiply the number of combinations of conditions needed. M4 is an example of a rule that takes advantage of M3's summarization, and M5 is a rule that recognizes the completion of the main goal, by testing each of the required aspects of the finished product. (An alternative, but less reliable, test would involve simply knowing that each step in a process was performed successfully.)

The total set of rules to perform the making of the car would of course be much larger than is shown above, in order to specify the details of the various subgoals of the above method. (Subgoals ultimately reduce to primitives such as those described in a later section.) One rule from a method for one of the subgoals is the following:

```
S1: If there is a goal to install a radio in a car,
then have the goal to move the car to the accessory assembler
    and have the goal to get a radio to the accessory assembler
    and have the goal to put the radio in the car using the assembler.
```

As shown, rule S1 asserts a number of subgoals. Though they are given in a particular order, the first two apparently could be done without regard to their order, and the last would probably make use of the results of the first two in order to ensure that the "assembler" machine has been provided with all the necessary inputs. The actual, detailed representations may include goal-subgoal pointers (e.g., expressed as attribute-value pairs). All of the sequencing implied by this discussion, though, would

be readily implemented as the presence (or absence) of conditions that would be recognized by rules. The generality of the recognize-act computational paradigm, with its global WM holding goals and data, relieves the rule encoder of some of the burden of specifying control information. This facilitates both initial instruction and later elaboration of the knowledge. As will be brought out further below, this ability to represent procedural knowledge as collections of rules, such as the ones just given, is one of the principal reasons for using PSs as a medium for instructable systems that are to grow by gradually adding details.

It can now be pointed out that the work with IPS takes a peculiar position on the central artificial intelligence topic of ***planning***, differing from a number of past approaches. The essence of the approach here is for the system to "muddle through"[6] tasks that are problematic, rather than doing a lot of planning, preparation, and anticipation of difficulties. A deliberate plan is never formulated and stored in a data structure for analysis, but behavior simply unfolds in response to changing conditions. Flaws or other interruptions in the flow of behavior are treated as new subproblems, and resolved by calling forth applicable methods or further instruction. It is not excluded that later on the system might be instructed to plan ahead in some fashion, or to add a reflective capability that would allow recognition of general classes of problems with known solutions and treat them accordingly [3].[7] The main aim here is to understand the basic goal structures and knowledge in a domain where many specific facts, brought to bear appropriately, are sufficient to produce effective behavior. Current general methods are unable to cope with such problems due to inability to control the search in such a large space.

---

[6] A system muddles through a problem when it engages in trial and error, without carefully considering consequences of its actions, relying instead on taking corrective actions after mistakes occur.

[7] Carbonell's 1982 paper, in this collection, also bears on this topic.

1.3. Overview

Through analysis of seven major attempts to build instructable PSs with various orientations, there were gradually formulated eight main functional components. Defining the eight components sharpened our understanding of the problems of the performance and learning modules, making them amenable to further research and design efforts. Beyond the narrow focus of the IPS project, this clarification can perhaps contribute to research on learning systems in general. After the eight components are listed in the next section, a broad overview of the IPS project is undertaken. The seven attempts, forming an evolutionary sequence, are cast into the functional component framework. In the process of doing this, lessons are extracted that apply to the whole enterprise as well as to individual explorations.

Members of the IPS project are no longer working together intensively to build an instructable PS, but individual studies that will add to our knowledge about one or more of these components are continuing. Progress in developing efficient PSs has been important to the IPS project [7], but will not be discussed further here.

2. Essential Functional Components of Instructable Systems

The components listed in this section are to be interpreted loosely as dimensions along which learning systems might vary.[8] In constructing a particular system, a point in a design space is located and developed. It is assumed that the mechanisms of a particular design embody approaches to several, or perhaps all, of these dimensions[9]. Almost all of the systems discussed in the next section, in fact, do not represent complete designs with respect to all functional components, but rely to some extent on further instruction to fill them in (usually this optimism was not justified). Also, as is the case in many design areas, a single mechanism can serve to fulfill the demands of several components at

---

[8] This approach owes a lot to Moore and Newell's dimensions for understanding systems [15].

[9] It is thus not considered fruitful to design systems that do each of these functions separately, or to talk about the structure of one without considering the overall system structure and orientation.

once. Observation of a system's behavior allows the formulation of the kinds of modifications, with respect to the design space of components, that could lead to improvement in the overall ability to build IPSs. To the extent that the functions of these components are expressed by explicit goals in an IPS, there is opportunity to exercise the overall system in the improvement of particular components.

2.1. Interaction

The content and form of communications between an instructor and an IPS can have a lot to do with ease and effectiveness of instruction. In particular, it is important to know how closely communications correspond to internal IPS structures. Inputs from the instructor can be in the form of entire methods or individual rules, in the form of more elementary WM units (whose composition into rules is thus less prominent in the external interactions), or in some other fragments even further removed from actual construction of rules. For example, consider the following rule (which is taken from the example of the preceding section):

```
M2: If there is a goal to make a car for a customer's order
    and the order specifies an engine of some size for the car
    and the car's body has been made,
then have the goal to install an engine of that size in the car.
```

One approach might be to give the rule in its entirety. Alternatives that make the interaction more fine-grained would have the instructor saying things like:

```
Note that the order specifies an engine of medium size for the car.
What size of engine does the order specify?
Test the previous result.
Try installing it.
```

With respect to the system-output direction of interaction, we must ask how well the manifest behavior of an IPS indicates its progress on a task. This issue is subject to considerations similar to those for input.

An IPS can have various orientations towards interactions, ranging from passive acceptance to active scrutiny. For instance, it can attempt, with varying degrees of effort, to maintain consistency and to assimilate new structures into existing ones. An IPS will be most effective when its orientation

is expressed as goals and thus subject to refinement by instruction.

2.2. Organization

Each version of IPS approaches the issue of obtaining correct and coherent behavior by attempting to organize its "procedural" knowledge. The need for such an attempt arises from two sources: one is to move the instructor away from having to specify control constructs, ie away from programming (which is difficult and violates the idea of instruction); another is that some form of systematic approach to control is needed, due to the inherent weakness[10] of production systems in this area. This may involve such techniques as collecting sets of rules into methods and using signal conventions for sequencing within methods. Whether IPS can explain its static organization and whether the instructor can see the details of procedural control are important subissues.

To illustrate some alternative organization approaches, recall the following rule:

```
M2: If there is a goal to make a car for a customer's order
    and the order specifies an engine of some size for the car
    and the car's body has been made,
then have the goal to install an engine of that size in the car.
```

In this rule, control is maintained by the third condition, which ensures that the rule will not become true until the preceding step of making the car's body is finished. One imaginable alternative is simply to remove that condition, and have the subgoal asserted potentially before it can be properly worked on. In this case, of course, the method for the subgoal would be likely to stop, blocked by the lack of a car body in which to install the engine. This shortened version of M2 is probably easier to modify and more modular, but it may make it more difficult for the instructor (for instance) to explain or coordinate the extra unfinished goals in WM. Another alternative makes the local sequencing of M2 more explicit by a step "counter" that is common to all rules in a method - knowing the current step is a way of knowing or summarizing the method's progress:

```
M2s: If it is step 2 of a goal to make a car for a customer's order
    and the order specifies an engine of some size for the car
then have the goal to install an engine of that size in the car.
```

---

[10] "Weakness" refers to a lack of a definite theoretical position built into the language itself.

```
M2t: If it is step 2 of a goal to make a car for a customer's order
     and the car's engine has been installed,
then mark the step of the goal to be 3.
```

These examples bring out an important trade-off in control conventions: explicit steps reduce the number and complexity of contextual conditions that a rule must test, and thus simplify it, but they reduce the flexibility of control by locking the system into some particular order of execution.

A key question facing the builders of IPS, and even of PSs more generally, is whether a procedural organization can exploit the full flexibility that seems inherent in PS architectures. Flexibility derives from having the control be open, on each PS cycle, to global recognitions that can: change the direction of processing by noting new facts; eliminate unnecessary steps by recognizing the satisfaction of the current goal or some higher one; and in general maintain the ability to switch to more efficient means for satisfying a goal. Flexibility enhances adaptability: to changes in the situation, to new knowledge or techniques (acquired, perhaps, without regard for actual application situations), to recognizable errors, and to new orderings of sequences of actions that might be appropriate to different situations. Certainly PSs can be programmed like conventional algorithmic languages, but there is potential for much more flexible, "intelligent" procedures.

2.3. Explanation

A key operation in an instructable system is that of explaining how the system has arrived at some behavior, whether correct or not. In the case of wrong behavior, IPS must reveal enough of its processing to allow the more intelligent instructor to determine what knowledge is missing, incorrect, or improperly represented. In the case of correct behavior, the instructor may wish clarification or elaboration on how it resulted. Ideally the explanation can occur at a point where it is also possible to make necessary corrections and additions before IPS gets too far off the track.

For example, the state of WM in the middle of executing the "make a car" method might look like

```
a goal to make a car for a customer's order,
the car's body has been made,
```

```
the car's engine has been installed,
a goal to make a radio,
the car's location is L24,
there is junk at location L25.
```

The explanation component would have to be able to detect unfinished goals, partially finished methods, unusual objects in the environment, and so on. This would be facilitated, for instance, if goals and subgoals had pointers to each other, if operators left some record of attempts, and so on - but too much of this sort of information can degrade the system's performance. Another problem is posed for the explanation component in selecting a small enough subset of critical items so that their communication is tolerable to the instructor.

2.4. Accommodation

When corrections to IPS's knowledge have been formulated by the instructor, the next step involves getting IPS to accommodate itself to new knowledge, i.e., to augment or modify itself, in response to the usual form of interactions with the instructor. In the IPS framework, these modifications are taken to be changes to the rules of the system, rather than changes to the less permanent WM. As with interaction, IPS can assume a passive or active orientation toward this process. A key problem in the process of accommodation is to properly modify behavior in one situation while maintaining other correct behavior from past instruction. One aspect of this is to find the location in the knowledge structure of the system where the modification is to occur, so that related, interacting knowledge can be taken into account.

Suppose, in the preceding (explanation) example, that a problem is caused by a failure to satisfy the prerequisites for making a radio. Then a rule like the following might suffice to fix the problem:

```
If there is a goal to make a radio
    and there is a goal to start the radio machine
    and there is not a power supply at L14
then have the goal to get a power supply at L14.
```

Note that this patch rule has to have enough conditions in it so that it can win the conflict resolution[11]

[11] The relevant conflict resolution principle here is specificity: a rule that matches more data, or more specific (detailed) data, will be preferred; see [13, 21, 6] for details.

over another (incomplete) rule, especially the rule that causes the starting of the radio machine without having all its requirements filled. Presumably there would be a rule in the system to set up subgoals to fulfill the prerequisites of making a radio, so that an alternative to the above patch rule might be to find and edit that rule by adding another subgoal. The deeper cause of why the rule was incorrect, e.g., in analyzing the inputs to the radio machine, is more difficult to deal with, but might be worth the extra accommodation effort, as it might avoid future errors. One approach might be to set up a rule as a monitor to watch for similar errors (i.e. those that omit some item of data) in the fulfilling of prerequisites.

2.5. Connection

This functional component and the ones that follow are considered "advanced" as opposed to the preceding "basic" componenets: they are much more difficult to formulate and implement.

Manifest errors are not the only way a system indicates a need for instruction: inability to connect a current problem with existing knowledge that might help in solving it is perhaps a more fundamental and frequent failing. An IPS needs ways both to assimilate problems into an existing knowledge framework and to recognize the applicability of, and discriminate among, existing methods. This concept of connection might also be termed "near contact", in that a close (but not exact) match to existing methods is involved, with differences resolvable by a few simple operations on the goal. An interesting issue revolves around how actively IPS processes new problems both for present and future connection. Connection abilities, particularly recognizing close or partial matches and transforming goals [16], are important due to the desirability of having IPS know when it needs instruction versus when it can make use of existing knowledge. The other side of this coin is the problem of discriminating among several methods that appear to be appropriate to a given new problem.

As a simple example, suppose the familiar "make a car" goal had been stated,

```
Make a sedan for a customer's order.
```

This can be readily transformed into the known form, if the possibility of mapping it is recognized. It might require noticing that sedan is a value of the "body" attribute in "make car" goals. A definition of "sedan" might also provide sufficient clues.

2.6. Reformulation

Another way that IPS can substitute for instruction is for it to reformulate existing knowledge to apply in new circumstances. This can also be termed mapping, analogy, transfer, serendipity, or "far contact". There are two aspects to this function: finding knowledge that is potentially suitable for mapping, and performing the actual mapping.[12] In contrast to connection, this component involves permanent transformation of knowledge in rules, either directly or by altering rules' effects at each firing, dynamically.

For example, suppose the goal,

```
Make a truck for a customer's order
```

were to come along and a method specifically for making trucks did not exist. Then some kind of analogical process might be appropriate, given the existing method for making a car. Namely, the goal might be transformed to "make a car", with the proviso that when "make a car" ran into problems, control would revert to an analogy method that would try to bridge the gap and fill in the missing step so that the "car" method could be resumed. This might be the case for making the truck's body, which would require special action, but we can suppose that adding an engine and accessories might be nearly identical in cases of truck and car.

2.7. Evaluation

Since the instructor has limited access to what IPS is doing, it is important for IPS to be able to evaluate its own progress, recognizing deficiencies and errors as they occur so that instruction can take place as closely as possible to the dynamic point of error. Defining what progress is and

[12] Carbonell's 1982 paper, in this collection, does this using means-ends analysis.

formulating relevant questions to ask in order to fill gaps in knowledge are two key issues. The assignment of blame for an error is the responsibility of the instructor in this IPS framework, with the explanation component assisting in diagnosis. It can also be helpful to include in evaluation some capabilities for having IPS produce additional external behavior, as in a "monitoring" or "careful execution" mode of operation.

The following rules illustrate the recognition of some possible error conditions:

```
E1: If an object with type junk is produced by a machine,
then have the goal of warning the instructor that
        the machine has produced that object.

E2: If there is a goal to make a car for a customer's order
    and more than 20 minutes have elapsed since the order arrived
    and there is not the result that the car's body has been made,
then have the goal of warning the instructor that
        progress is slow on the order.
```

2.8. Compilation

Rules initially formed as a result of the instructor's input may be amenable to refinements that improve IPS's efficiency. This follows from several factors: during instruction, IPS may be engaged in search or other "interpretive" execution (including a richer goal structure); instruction may provide IPS with fragments that can only be assembled into efficient form later; and IPS may form rules that are either too general or too specific. Improvement with practice is the psychological analog of this capability. Anderson *et al* [2] have formulated several approaches to compilation, such as condensing, into a single rule, rules that typically occur in a fixed sequence.

The improvement that can be obtained from compilation is illustrated by the following rule, whose actions consist of direct environmental commands rather than goals and subgoals:

```
C1: If there is a goal to make a car for a customer's order
    and the order specifies a sedan body and a medium engine
then start the sedan machine
    and start the engine4 machine
    and move an object from L22 to L23.
```

2.9. Discussion of Components

It is evident that realizing the components described in this section is made difficult by the myriad combinations of knowledge that can occur. Because an IPS is potentially working in various environments of different complexity, it is difficult to take advantage of stereotypes in procedural forms. Others have in fact made progress by assuming fixed-format rules (e.g., transformational grammars) or simplified execution schemes (e.g., backward chaining). Our approach contrasts with those in avoiding any assumptions on the form of the environment and in leaving the system architecture open for general procedures.

3. Survey of Approaches

Each attempt to build an IPS has started with a hand-coded ***kernel*** system, with enough structure in it to support all further growth by instruction. The kernels established the internal representations and the overall approach to instruction. At the very least, such kernels require the ability to interact with the instructor and to construct new rules. Three properties are desired in such a kernel system:

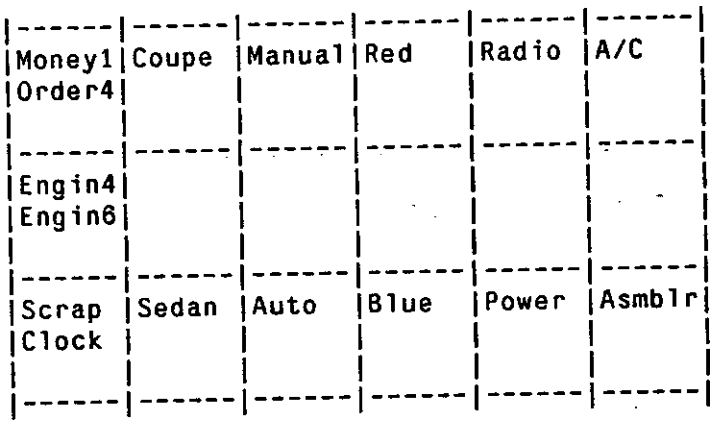
- It is to be hand-coded, and as modular as possible.
- Everything in it is to be potentially modifiable by instruction. Usually it is constructed as if it were acquired by instruction, i.e., with rules of similar form to those resulting from instruction.
- It is to be open to expansion in any of a number of directions, depending on which problems the instructor wishes to explore.

Seven kernels or kernel approaches were studied during the history of the IPS project, and they are presented below in roughly chronological order. Kernel1, ANA, Kernel2 and IPMSL were fully implemented. The remainder either were suspended at various early stages of development (with their best features incorporated into newer proposals) or are still being elaborated and developed in the context of other research. There is a table near the end of the section that summarizes a number of attributes of the kernels.

3.1. The Abstract Job Shop Task Environment

The task domain for the IPS project was the manipulation of objects in a symbolic ***task environment*** (TE), a simulated, simplified "factory", in which an IPS system has a limited set of "sensory" and "motor" ***operators***. A typical job shop is shown in the accompanying figure. Each ***object*** in this toy environment is represented as a LISP property list. The TE itself is an object with a particular set of components, termed ***locations***, arranged in an array and represented as rectangles in the figure.

ABSTRACT JOB SHOP



The entire ensemble, in the spirit of keeping it as an "external environment", is separate from the processes and memories of the PS architecture, except for the interface provided by the following operators:

- View. A representation of its argument, an object, is placed in WM (as if obtained through an "eye").
- Scan. An object is sought in the TE containing a given attribute-value pair. It is Viewed, if found.
- Trans. The top object at one location is transported to another location.
- Start. A machine (an object with a special set of properties) is started. It goes through one cycle of its operation, which is all within the action cycle of the rule containing the invocation.
- Compare. The values of a specified attribute of two objects are compared, producing a

difference according to the values' type.

Note that the above operators are invoked as actions in rules, making modifications in the TE and reporting changes in the TE by asserting data into WM. All of this occurs within a single recognize-act cycle of the PS. The most important and complex operator is Start, which activates machines. A machine is a special-format object that takes some objects as inputs (in some cases consuming them) and produces other objects as outputs. Usually constraints on the machine's operation make problems in the domain more challenging.

Some sample problems, of varying difficulty, are the following:

- Examine the object at the top position of some location.
- Compare two objects.
- Find an object with a given set of properties.
- Transport an object with a given set of properties to a given location.
- Manufacture an object with a given set of properties, within some budgetary and time limits.

The "find" class of task involves searching through the TE, viewing objects and comparing them with the desired description. It is thus a prototypical task of interest in instructional situations. "Transport" problems are complicated by a feature of objects stored at a location: they are stacked on top of each other such that to move one, it has to be at the top of a stack. Getting an object to the top can involve moving objects elsewhere, with the potential for creating conflicts with other subgoals in a larger plan. While details of the pictured TE need not be given, it can be described as an assembly-line layout for making automobiles. While this TE is straightforward, the language for defining TEs can express great complexity.

3.2. Kernel Version 1

The starting point for IPS was the adoption of a pure means-ends strategy: given explicit goals, rules are the means to reducing or solving them. Four classes of rules are distinguished:

- means rules;
- recognizers of success;
- recognizers of failure;
- and evocation of goals from goal-free data.

The Kernel1 [22] approach goes further than this in its organization component, which consists of ways for grouping rules into methods (as defined and illustrated in the first section above). The main mechanism of grouping is to have rules of the above types share a common goal pattern. The interaction component consists of a straightforward processor for language strings that correspond to methods and to system goals (among which are queries). Keywords in the language are used to signal that the Kernel is to insert method sequencing tags. There are also keywords that delimit rule boundaries within methods. The explanation component is unspecified at the start, leaving it to the instructor to develop (and instruct) methods that could generate helpful information by piecing together various goals and data in WM. This reliance on instruction turned out to be a serious weakness, though a lot of the right kind of information was available in WM.

Although Kernel1 was used as a basis for instruction, its effectiveness was severely hampered by its weak or non-existent components for explanation, accommodation, connection and reformulation. Only small progress was made in the areas of evaluation[13] and compilation[14]. Much of the flavor of the means-ends approach was retained in later Kernels.

Kernel1 is illustrated in the protocol below, whose objective is to instruct IPS to perform the simple

---

[13] Described briefly in an unpublished appendix to this paper, available from the author.

[14] This consisted of recognizing the applicability of techniques such as those in [2], to our means-ends rules

task of examining the top object at some location in the TE. The method to be instructed can be summarized as follows: To examine the top object at some location, first View the location, then test if any objects are there; if so, find the first and use it as the result; otherwise, "nothing" is the result. Note that the "test if any objects" part of this method is a subgoal, to be instructed separately. The first clause of this method is given to Kernel1 as follows:[15]

```
To examine the object at the top position                    (A)
          of some location ,
    want view location that location in the TE               (=> B)
    then want test the status of the value of the            (C, (A1))
          composition of that location ,

    . . .
```

The marginal notations in the above indicate that the instruction gives rise to a rule with a condition element 'A', the main goal of the method, and two action elements ('B' and 'C'), which are subgoals of the main goal. In addition, there is a modification (indicated by the subscript '1') to the goal element to achieve sequencing to the next step of the method (not shown). The complete input for this method involves four clauses of similar length and form to the one given, all given without a break for system responses. Kernel1 adds some sequencing control to other rules in the method by inserting the main goal as a condition, suitably modified with step counters. These additions are one advantage of using Kernel1 over programming directly in OPS rules, although the distance between the two forms of coding is not conceptually large - they are both forms of programming, as distinct from tutorial instruction.

While a large fraction of the rules of Kernel1 are devoted to processing the (admittedly clumsy) input language illustrated above, the main design objective and achievement was to embed simple means-ends connections, as expressed by instruction text, in an organization that would ensure production of the desired behavior - i.e., organization rather than interaction was the main focus.

---

[15] An unpublished appendix (available from the author) to this paper contains the full instruction text, along with a more detailed explanation.

Unfortunately, two properties of the above style of interaction are very detrimental to effectiveness. First, Kernel1 accepts the input passively, with no interaction (e.g., questioning) involved. Second, the instructor receives no feedback on the correctness of the many parts until the entire method is tried. Kernel1 failed to provide an adequate basis for interaction, explanation, and performance due to a number of practical considerations: difficulty in knowing the side conditions of rules (those other than the main goal), lack of a mechanism for constructing tests of proper goal satisfaction, lack of having goal-subgoal links created automatically, and goal representation deficiencies, particularly failure to distinguish different occurrences of the same goal (as in recursion) and to allow goals to be augmented with new information as processing developed. The instructor was relied on to provide too much programming detail, in a situation where a programming approach is considered harmful.

In spite of its shortcomings, Kernel1 accomplished a few important tasks, as far as overall IPS project goals were concerned. It established the basic means-ends form for the organization component. It clarified the need for more PS efficiency, and for improvement in the explanation, accommodation, and other functional components. In short, it gave us a better appreciation for the difficulty of the instruction task.

3.3. Additive Successive Approximations (ASA)

Some of the drawbacks of Kernel1, especially those surrounding interaction, can be remedied[16] by orienting instruction towards fragments of methods that can be more readily refined at later times. Interaction consists of having the instructor designate items in IPS's environment (especially WM) in four ways: condition (for data or configurations that are important context to be taken into account while working on a goal), action (for operators appropriate to solving a goal), entity (to create a symbol and some associated knowledge about the entity), and relevant (to associate one of the other three designated items with a particular goal). The system is to respond to a 'relevant' designation by building rules with the given conditions or actions, or by building rules that create or augment

[16] These ideas were introduced by A. Newell in October, 1977.

knowledge expressions. These designations result in methods that are very loose collections of rules, each of which contributes some small amount towards achieving the goal. Accommodation is done as ***post-modification*** of an existing method in its dynamic execution context, through ten method-modification methods. Some of these are: delay an action, advance an action, remove an action, conditionalize an action, and put two actions into a strict sequence.

Though the ASA ideas were never implemented, some aspects of the approach were used in the Kernel2 system, described in detail below. Probably ASA would suffer from the same difficulties described in connection with Kernel2.

3.4. Analogy (ANA)

A concerted attempt to deal with issues of connection and reformulation is represented by McDermott's ANA program [14]. Starting out with the ability to solve a few very specific problems, it attacked subsequent similar problems by analogizing from its known methods. Initial methods to solve TE problems were hand-coded, a deviation from the kernel constraints given above. In ANA, connection is achieved by coding special ***method description*** rules, which recognize the class of goals that appear possible for a method to deal with by analogy. The possibility that an analogy may work is discovered by following taxonomic links originating at a given goal's actions and object arguments. When a link is traversed, revealing the object (class) or action (class) at the end of it, a method description rule may become satisfied, thus making a connection on which an analogy can be based. A preliminary analogy is set up using the discovered correspondence of objects or actions, the goal is modified by substitution, and the method is started. As it executes, rules recognize points where the analogy breaks down. General analogy methods are able either to patch the method directly with specific substitutions or to query the instructor for new means-ends rules.

In either case, reformulation occurs because rules record the patches for use in later similar problems. Compilation occurs, with visible improvement in performance, as fewer and fewer of the

error-recognition rules are brought into play. Thus, ANA combines connection, reformulation, evaluation and compilation components.[17]

3.5. Kernel Version 2

With basic ideas similar to ASA and to Waterman's Exemplary Programming [25], the Kernel2 approach [23] focused on the process of IPS interacting with the instructor to build rules in a dynamic execution context. The instructor essentially steps through the process of achieving a goal, with IPS noting what is done and marking elements for inclusion in the rules to be built when the goal is achieved. The organization of methods in Kernel2 is less adventurous than proposed in ASA, keeping more to the traditional control and means-ends concepts of Kernel1. Similarly, it provides few innovations with respect to accommodation, explanation, and the other IPS components. Kernel2 includes a semantic network of information about its methods, for use as a "help" facility. It is the basis from which the IPMSL system, below, is built.

Based on the preceding description, Kernel2 would seem to offer little advance over Kernel1. However, the following protocol illustrates the large difference that occurs with the shift towards interaction that is more frequent and in a dynamic method-execution context. The protocol below is not an actual interaction between instructor and Kernel, due to a shift in the domain of IPS away from the TE used for Kernel1. Kernel2 was actually implemented, though, and the following is representative of the style of instruction achieved. The actual interaction language has been smoothed out somewhat in this protocol. "I:" precedes inputs from the instructor, and "K:", the outputs of Kernel2.

```
I: Examine the object at the top position of L23

K: Unable to do WS003:
       Examine object at top position of L23

I: Goal of rule: WS003
```

(A)

[17] For another approach to learning by analogy, see Carbonell, 1982, in this collection.

```
K: OK, working on new rule P39

I: Add: want view location L23 in the TE                    (=> B,)

K: View location result is L23 in the TE                    (Note 1.)

K: L23 has member TE, composition (M1 M2 C5),
       type stack

I: Add: want test the status of the value of the                    (C)
       composition of L23

K: Unable to do WS009: Test status of value of
    composition of L23

I: L23 is a variable in P39

K: OK

I: Build P39                                        (rule: A => B, C)

K: OK

I: Goal of rule: WS009

K: OK, working on new rule P40
```

Expressions are given at the right margin to mark places in the text where certain rule elements have been specified, in a way similar to the protocol for Kernel1. The rule that is constructed as a result of this dialog is, in fact, almost identical to the one built by Kernel1. Of interest is what is happening at the point where "*Note 1.*" occurs: here is an example of the utility of dynamic context. At this point, Kernel2 actually executes an operator on the TE, and displays the result, so that the instructor can easily formulate what he wants done with that result in the succeeding steps of the method being instructed.

To summarize the key aspects of Kernel2, interaction happens in the context of a concrete attempt at solving a goal. The effect of this immediate feedback is that the instructor can have a much better idea of how much the system knows about the present context. Interactions are far removed from the constructed rules, corresponding both to small pieces of those rules and to interactive commands

that might be given to the system to have it do something. The dynamic context provides stimuli to the instructor, helping to avoid errors and omissions, and placing less burden on the instructor's memory. When a piece of a rule given by the instructor is a recognizable goal to the system, it automatically tries to achieve the goal, and the instructor can watch this activity and observe its results. Kernel2 is much simpler in structure than Kernel1 (fewer rules, and more easily coded), due to radical simplification of its input language. Instructions to Kernel2 are much shorter, and feedback to the instructor is immediate.

3.6. Conclusions on Direct Approaches

The above approaches are all ***direct*** in the sense that the orientation is towards rules and pieces of rules rather than towards knowledge that is structured in some other more natural form. One conclusion from the direct approaches is that instruction must be organized in units other than rules - rules are too large and tend not to be a natural form for instruction, especially when various PS control and supporting structures are taken into account. Also, rules tend to require a belabored, repetitious style of instruction, where the natural tendency is to make assumptions about the capabilities of the receiver of instruction, and to use various forms of ellipsis. The instructor should not be allowed to perceive instruction as programming, as this is an unnatural mode of instruction.

In the ***higher-level*** approaches that follow, more is attempted in terms of functional components for explanation, accommodation, and the advanced components. Another common theme is the need for a more active, "agenda" orientation, including system goals that are pursued along with those of the instructor.

3.7. Problem Spaces

Problem spaces [18][18] were proposed as a higher-level organization for IPS, in which all behavior and interactions were to be embedded in search. A ***problem space*** consists of a collection of

[18]This approach was formulated by A. Newell and J. Laird in October of 1978.

knowledge elements that compose ***states***, plus a collection of ***operators*** that produce new states from known ones. A ***problem*** consists of an initial state, a goal state, and possibly path constraints. Control in a problem space organization is achieved through an executive routine that maintains and directs the global state of ongoing searches. Newell's Problem Space Hypothesis (*ibid.*) claims that all goal-oriented cognitive activity occurs in problem spaces, not just activity that is problematical.

According to the proposal, interaction would consist of giving IPS problems (presumably WM structures) and search control knowledge (hints as to how to search specific spaces, presumably expressed as rules). Every Kernel component would be a problem space too, and thus subject to the same modification processes. The concrete proposal as it now stands concentrates on interaction, explanation (which involves sources of knowledge about the present state of the search), and organization.

3.8. Semantic Network (IPMSL)

The IPMSL (Instructable PMS Language, where PMS is a computer description formalism) system [23] viewed accumulation of knowledge as additions to a semantic network. In this view, interaction consists of definition and modification of nodes in a net, where such nodes are PS rules. The network stores four classes of attributes: taxonomic (classifying methods and objects), functional (input-output relations for methods), structural (component parts of methods and objects), and descriptive (various characteristics). Display and net search facilities are provided as aids to explanation and accommodation, though the instructor is responsible for storing the required information in the network in the first place. The availability of traditional semantic network inferences makes it possible for IPMSL to develop a more uniform approach to connection and reformulation, since they provide a set of tools for relating and mapping knowledge into more tractable expressions. Because IPMSL is a digression into a task domain different from the Abstract Job Shop, emphasis was placed on descriptive capabilities rather than on mechanisms for procedural organization. Thus the semantic network orientation was not integrated with Kernel2 approaches to the acquisition and description of

procedural knowledge. The remaining step towards integration would be to have Kernel2 record information in the network, as methods are built up using its interaction primitives.

3.9. Schemas

The use of schemas as a basis for an IPS kernel[19] made slot-filling the primary information-gathering operation. A slot would be implemented as a set of rules. The slots are: executable method; test of completion; assimilation (connects present WM with the schema for a goal); initialization (gathers operands for a method); model (records the instruction episode for later reference); accommodation (records patches to the method); status (records gaps in the knowledge); monitoring (allows careful execution); and organization (records method structure). Orientation towards instruction would be active, as in ASA. Explanation would consist of interpreting the model slot, and accommodation, of fitting additions into the model. Connection would use a discrimination network composed of the aggregated assimilation slots of all schemas. Compilation would be needed here, to map model to method.

The following protocol illustrates the flavor of the proposed schema approach. A system based on this approach, but outside the IPS framework, is currently under development, but for expository reasons, the protocol here is a simulated one. There are notations similar to those for the Kernel2 protocol above. Notes in the right margin in some cases give the names of slots being filled by the adjacent interactions. Other notes refer to explanations in the next paragraph.

```
I: Examine the object at the top position of L23

K: Unknown goal WS003:                                        (A)
       Examine object at top position of L23

K: What is the goal WS003 attempting to achieve?

I: The name of some object,                                   (TEST)
       the position of that object is 1,
       the member of that object is L23
```

[19] Schemas were first proposed for IPS by Rychener, May, 1978

```
K: Does this result differ from a view object in the TE,
       because of "at the top position"?

I: Yes                                                    (ASSIMILATION)

K: Part of the desired information of the goal can be
       obtained from a view operation

K: To view requires an object or location to work on,              (INIT)
       which L23 seems to satisfy
                                                           (=> B)
K: L23 has member TE, composition (M1 M2 C5),              (Note 1.)
       type stack

K: Viewing L23 did not produce the desired attributes

I: You need to view the object at the top position of L23

K: I lack objects with position 1 with member L23

I: Test the status of the composition of L23                       (C)

K: Unknown goal WS009:                          (rule: A => B, C)
       test status of composition of L23                   (Note 2.)

K: What is goal WS009 attempting to achieve?

I: Whether the composition of L23 is empty or nonempty             (TEST)

K: I have nothing that produces that result

I: The composition of L23 is (M1 something),
       so its status is nonempty
```

The most striking feature of this protocol is that IPS is taking much more initiative in determining the direction of the dialog. It is questioning the relation of new information to what already exists, and is actively trying to fill in missing slots in its new schemas. At "Note 1" IPS tries an experiment with a TE operator, to get at least a partial result that might aid in solving the goal. At "Note 2", IPS is entering into the context of a new schema. Presumably, the former schema will be reactivated at a later time when the subgoal's schema is satisfactorily filled in.

The schemas approach as sketched here makes a few significant additions to Kernel2. The orientation towards filling slots is the primary source of the improved, natural style of interaction.

Schemas implemented as PSs are self-activating and procedural; the slots can contain more than just stereotyped values, namely arbitrary methods and structures of rules; and goals for filling slots will reside in the global WM, and thus can be filled flexibly, and in a data-dependent way (i.e., as data become available that are recognized as relevant).

3.10. Conclusions on Higher-Level Approaches

The approaches discussed above illustrate the advantages of using higher-level organizations for the overall instructional process. The importance of carefully attending to the style of instruction should be evident. Adopting these approaches has the two-fold benefit of providing a more natural communication medium for the instructor of the system, and of providing goals and methods for the system itself to mold new knowledge into well-organized, flexible, complete, and reliable methods. The system can also be more free than before to experiment for itself, given its agendas and search mechanisms. Higher-level approaches aid in developing effective versions of the more advanced functional components, in that such components are natural consequences of adopting any of the above specific approaches. The accompanying figures summarize the seven approaches.

4. Discussion

The IPS project has invented and explored the consequences of a number of plausible learning system components in the "learning by being told" paradigm. One is the means-ends organization of Kernel1, along with its approach to debugging using a dynamic goal tree context and to compiling by eliminating temporary goal structures. Means-ends also holds the promise of expanding a system's abilities in directions where explicit goals can be formulated. The use of explicit tests and failure recognizers can add reliability and robustness to means-ends execution. A second contribution has been the study of knowledge acquisition in a dynamic execution context (illustrated by the Kernel2 dialog above). Other contributions include the development of the problem space idea, the orientation of a learning system towards active assimilation and accommodation (as in schemas), the ability to dynamically use analogies, the use of rules to implement semantic networks, and the organization of rules into schemas. This paper has tried to motivate the need for more study of

IPS: Component	Kernel1	ASA	ANA	Kernel2
Interaction	whole method query goal	four desig. forms	patch goal	many desig. query goal
Organization	sequenced methods	loose means-ends links	hand-coded for analogy	sequenced methods help net of functional info
Explanation	(WM data)	?	?	(Kernel1)
Accommodation	whole rules - blindly	method-modif. methods	(Kernel1) (see Reform.)	(Kernel1)
Connection	(ad hoc means-ends rules)	?	method descr. rules taxonomy search	(Kernel1)
Reformulation	?	?	patch rules map actions & objects	?
Evaluation	monitor of goals	?	recog. of break-down of analogy	instructor, in dynamic context
Compilation	(compose out goal structures)	(needed even more than in Kernel1)	patch rules analogize faster	(Kernel1)
Implemented?	yes	no	yes	yes
Reference	[22]	(see Kernel2)	[14]	[23]
Failings	too much like programming poor goal repr. overemph. language instructions too long weak explanation no method-modif. methods orientation too passive	would control work?	no approach to instruction	too slow task was shifted
Starting size	325 rules, incl. 50 in monitor, added later		295 rules, incl. 55 in TE methods	45 rules
Instruction	9 elementary tasks = 160 rules	0	4 tasks = 140 rules	Kernel grew = 55 rules
Final size	485 rules		435 rules	100 rules (see IPMSL)

Key: Potential or theoretical capabilities ("left to instruction") are in ()s; numbers of rules are rounded.

IPS: Component	Problem Spaces	Semantic Net (IPMSL)	Schemas
Interaction	problems search control	Kernel2 + net defining, updating	actively fill slots + Kernel2
Organization	problem spaces for all compon. & methods	Kernel2 + network: function, taxonomy, structure, description	schemas with 9 types of slots; esp. method, model, init, test
Explanation	knowledge about state of search	supported by network info	(model slot interp.)
Accommodation	(problem space)	supported by network info	(edit model) (status slt)
Connection	(problem space)	(net inferences & searching)	discrim. net
Reformulation	(problem space)	(network info)	?
Evaluation	executive	?	monitor slot
Compilation	specific, ad hoc search control	?	(transform model slot)
Implemented?	no	yes	no
Reference	[18]	[23]	
Failings		too big & slow	
Instruction/ Testing		160 rules in net	
Start size		450 rules = 100 Kernel2 + 120 basic net + 230 advanced net	
Final size		610 rules	

Key: Potential or theoretical capabilities ("left to instruction") are in ()s; in unimplemented proposals, ()s are reserved for very vague possibilities; numbers of rules are rounded.

approaches to instruction and of ways of achieving the functional components, by exhibiting an evolutionary sequence and by pointing out the deficiencies of various partial designs. Our studies to date into these issues have been greatly facilitated by the use of a flexible, expressive medium, the OPS PS architecture.

Two key problems remain unsolved and open for further research: achieving the kind of procedural flexibility and robustness that would seem to be inherent in the PS architecture; and devising ways for a system to effectively manage its knowledge (however organized), i.e., techniques for accommodation as defined above. Procedural flexibility has been discussed above in association with the organization component. The ideal flexibility ought to derive from the global recognize-act cycle, where heuristics and optimizations could be applied at each step to guide and complete goal processing. For a system to manage its knowledge, much more needs to be known about the structure of methods and how they are modified and augmented. The IPS project has failed to get beyond the most basic of method manipulations, partly due to its emphasis on other aspects of the overall problem and partly due to the inherent difficulty of the problem area.

Explorations within our particular framework can profit from and stimulate research in information-processing psychology. Of particular interest would be a protocol analysis of instructional dialogs in an environment similar to our TE, after the fashion of Newell and Simon [17]. Additional information would be provided by querying the subject to determine what rules have been learned, after a session with an unknown problem environment. The structuring of the instructional session by a human tutor with a human subject is important, as it may give some indication of the underlying knowledge representations involved. The best attempts by psychologists at studying instructional learning at this level of detail seem to be found in work such as Klahr's collection [9]. On the AI side, the work of Collins [5] seems to be the closest in spirit. It may be in general that people do not require the painstaking explanations that seem to be needed by PSs. At least, this holds for PSs with very little knowledge, as discussed here. That is, humans are better learners because they know more and can

fill in gaps in the instructional interaction. Thus it may be that our PS work must develop new techniques that haven't been necessary with human education. On the other hand, humans' learning might improve if we knew better how to organize instruction to suit their internal knowledge structures, or if we could train them to use a more efficient knowledge organization.[20]

Acknowledgments. Much of the work sketched above has been done jointly over the course of several years. Other project members are (in approximate order of duration of commitment to it): Allen Newell, John McDermott, Charles L. Forgy, Kamesh Ramakrishna, Pat Langley [10, 11], Paul Rosenbloom, and John Laird. The present author's perspective, emphasis, and statement of conclusions may differ considerably from those of other project members - the broad scope of the IPS Project fostered and encouraged a diversity of approaches. But much credit goes to the group as a whole for the overall contributions of the research. Helpful comments on this paper were made by Allen Newell, Jaime Carbonell, David Neves, Robert Akscyn and Kamesh Ramakrishna. The editors and reviewers of this book have also been very helpful.

References

1. Anderson, J. R.. *Language, Memory, and Thought.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1976.

2. Anderson, J. R., Kline, P. J., and Beasley, C. M. Jr. A Theory of the Acquisition of Cognitive Skills. Tech. Rept. 77-1, Yale University, Dept. of Psychology, January, 1978.

3. Anzai, Y. and Simon, H. A. "The theory of learning by doing." *Psychological Review 86*, 2 (1979), 124-140.

4. Buchanan, B. G., Mitchell, T. M., Smith, R. G., Johnson, C. R. Jr. Models of Learning Systems. Tech. Rept. STAN-CS-79-692, Stanford University, Computer Science Dept., January, 1979.

5. Collins, A. Explicating the Tacit Knowledge in Teaching and Learning. Tech. Rept. 3889, Bolt, Beranek, and Newman, Inc., March, 1978.

[20] The author would appreciate references to current work along the lines discussed in this paragraph.

6. Forgy, C. and McDermott, J. OPS, a domain-independent production system language. Proc. Fifth International Joint Conference on Artificial Intelligence, 1977, pp. 933-939.

7. Forgy, C. L. *On the Efficient Implementation of Production Systems.* Ph.D. Th., Carnegie-Mellon University, Dept. of Computer Science, February 1979.

8. Forgy, C. L. OPS4 User's Manual. Tech. Rept. CMU-CS-79-132, Carnegie-Mellon University, Dept. of Computer Science, July, 1979.

9. Klahr, D.. *Cognition and Instruction.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1976.

10. Langley, P. W. *Descriptive Discovery Processes: Experiments in Baconian Science.* Ph.D. Th., Carnegie-Mellon University, Dept. of Psychology, May 1980.

11. Langley, P., Bradshaw, G. and Simon, H. A. Rediscovering chemistry with BACON.4. In *This Volume*, Tioga, 1981.

12. Lesser, V. R. and Erman, L. D. A retrospective view of the HEARSAY-II architecture. Proc. Fifth International Joint Conference on Artificial Intelligence, 1977, pp. 790-800.

13. McDermott, J. and Forgy, C. Production system conflict resolution strategies. In *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F., Eds., Academic, New York, NY, 1978, pp. 177-199.

14. McDermott, J. ANA: An Assimilating and Accommodating Production System. Tech. Rept. CMU-CS-78-156, Carnegie-Mellon University, Dept. of Computer Science, December, 1978. Also appeared in IJCAI-79, pp. 568-576

15. Moore, J. and Newell, A. How can MERLIN understand? In *Knowledge and Cognition*, Gregg, L., Ed.,Lawrence Erlbaum Associates, Potomac, MD, 1973, pp. 201-252.

16. Mostow, David J. *Mechanical transformation of task heuristics into operational procedures.* Ph.D. Th., Carnegie-Mellon University, Dept. of Computer Science, April 1981.

17. Newell, A. and Simon, H. A.. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, NJ, 1972.

18. Newell, A. Reasoning, problem solving and decision processes: the problem space as a fundamental category. In *Attention and Performance VIII*, Nickerson, R., Ed.,Lawrence Erlbaum Associates, Hillsdale, NJ, 1980.

19. Rychener, M. D. The STUDNT production system: a study of encoding knowledge in production systems. Carnegie-Mellon University, Dept. of Computer Science, October, 1975.

20. Rychener, M. D. *Production systems as a programming language for artificial intelligence applications.* Ph.D. Th., Carnegie-Mellon University, Dept. of Computer Science, December 1976.

21. Rychener, M. D. "Control requirements for the design of production system architectures." *SIGART Newsletter 64* (August 1977), 37-44. ACM

22. Rychener, M. D. and Newell, A. An instructable production system: basic design issues. In *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F., Eds., Academic, New York, NY, 1978, pp. 135-153.

23. Rychener, M. D. A Semantic Network of Production Rules in a System for Describing Computer Structures. Tech. Rept. CMU-CS-79-130, Carnegie-Mellon University, Dept. of Computer Science, June, 1979. Also appeared in IJCAI-79, pp. 738-743

24. Rychener, M. D. OPS3 Production System Language Tutorial and Reference Manual. Carnegie-Mellon University, Dept. of Computer Science, March, 1980. Internal Working Paper

25. Waterman, D. A. Rule-Directed Interactive Transaction Agents: An Approach to Knowledge Acquisition. Tech. Rept. R-2171-ARPA, The Rand Corp., February, 1978.

26. Young, R. M. Production systems for modelling human cognition. In *Expert Systems in the Micro Electronic Age*, Michie, D., Ed.,Edinburgh University Press, Edinburgh, 1979, pp. 35-45.

Appendices: Details Omitted from Survey

I. Details on Kernel1

As a first approximation to an evaluation component, Kernel1 has a goal-monitoring facility, expressed fully as rules, which exploits the clean structure of the means-ends approach. This monitor is enabled by a monitoring goal, and the monitoring rules contain general enough patterns to allow the matching of arbitrary goals, so that points in IPS's behavior that involve new goals can be interrupted and examined (cf. "talking aloud" protocols of human problem solving behavior). Thus what is monitored includes only new goals, but this allows the instructor to follow the system's progress in greater detail than is possible by simply watching its behavior in the TE or by noting its utterances. There is a potential for more advanced monitoring, in that ad hoc rules could be constructed to recognize particular error situations, degree of progress towards achieving certain goals, and others referred to in the definition of evaluation, above.

Another feature of a pure means-ends approach is the opportunity for compilation: usually the instructor formulates a rich goal tree during instruction, but the tree can usually be considerably trimmed down, with intermediate, temporary goals being replaced by direct concatenation of existing rules (Anderson, *et al.* [2]).

The following is the full instructional text given to Kernel1, for the method described in the text.

```
To examine the object at the top position                    (A)
        of some location ,
   want view location that location in the TE                (=> B)
   then want test the status of the value of the             (C, (A1))
        composition of that location ,

Next if the test status result is non-empty is               ((A1), CR)
        the value of the composition of
        that location , ,
   want find the element first of the value of the           (=> D, (A2))
        composition of that location ,

Else if the test status result is empty is the value         ((A1), CR)
```

```
        of the composition of that location , ,
    the examine object result is nothing at the top                    (A_R)
        position of that location ,

Next if the find element result is some object is first                ((A_2), D_R)
        of the value of the composition of
        that location , ,
    want view object that object in the TE                             (E)
    and the examine object result is that object is at                 (A_R)
        the top position of that location .
```

Several notational conventions in the above need to be explained. *Italics* are used to indicate words in the input language that are known to Kernel1, namely words that delimit boundaries of various phrases and clauses, and also words that indicate variablization ("some" and "that" mark pattern variables). The remaining ones become the contents of goal expressions, and are essentially unparsed. At the right margin are placed a number of indicators in parentheses that label the various phrases involved. There are also indications of boundaries between condition and action of the corresponding rule ('=>') and between rules (blank lines). For example, the first five lines give three rule elements (labelled A, B, and C), and the first is a condition while the rest are actions in the resultant rule. If a label is parenthesized further, this is an indication that it is implicit, i.e., Kernel1 adds that extra condition or action to the rule. Subscripts indicate certain special modifiers of the corresponding phrases: the numeric ones are "step" indicators (the method given above has two steps in addition to the beginning, indicated by the main goal phrase being A, A_1, and A_2); an "R" indicates a "result" marker, i.e., the result of satisfying a goal. Note that some variant of "A" occurs in each of the rules formed, and that the whole method ends with the A_R result.

The major part of Kernel1 consists of a set of rules for processing (but not parsing or semantically understanding) language like that given above, and forming the resulting rule fragments into methods that can be executed with the desired results. The language itself is quite clumsy due to the emphasis placed on getting the output of the processing into a form that could reliably execute (within the PS architecture). That is, the emphasis was placed on the organization component as defined above.

Not only is the language clumsy, but it was realized after some work with Kernel1 that a couple of other properties of the above style of interaction are very detrimental to effectiveness. First, Kernel1 accepts the input passively, with no interaction (e.g., questioning) involved. Second, the instructor receives no feedback on the correctness of the many parts until the entire method is tried.

Kernel1 embodied a hypothesis that simple means-ends rules could provide an adequate basis for interaction, explanation, and performance. A number of practical considerations prevented satisfactory realization of our goals in this respect. First, it is hard for instructor to know side conditions - certainly the main goals are easily picked out from WM (via queries that are not illustrated), but it is considerably more difficult to know which other items of data are the relevant conditions that are tested in a rule along with the goal. Knowing such conditions is crucial if a method is to be corrected by adding rules that contain further discriminating conditions, discovered to be necessary for proper behavior subsequent to the initial instruction. Second, tests of proper satisfaction of a goal upon completion of a method are not utilized enough - such tests match a "result" with other data conditions in WM to verify that all has gone according to expectations. Tests are not constructed automatically by Kernel1, but rather the instructor must remember to go through the extra effort of instructing them. Third, goal-subgoal links among goals are not established by Kernel1 - if the instructor understands that such conventions are useful in reconstructing goal trees as an attempt at explanation, for instance, he or she must go through the extra steps involved in specifying those links and keeping them up to date with extra rule actions. Fourth, having a goal expressed as a string of words doesn't suffice to distinguish separate instantiations of the same goal (as in recursion), nor does it provide a flexible way to augment a goal with new information during processing. To summarize, much too much of the burden of programming detail is placed on the instructor, and instruction remains too much like programming. Furthermore, the Kernel1 language fails to provide the capabilities to support such programming.