Integrating Learning and Problem Solving Within a Chemical Process Designer

by

Ajay K. Modi, Arthur W. Westerberg

EDRC 06-82-89

Integrating Learning and Problem Solving within a Chemical Process Designer

Ajay K. Modi and Arthur W. Westerberg

Department of Chemical Engineering
and
Engineering Design Research Center
Carnegie Mellon University
Pittsburgh, PA 15213

This work has been supported by the Engineering Design Research Center, a NSF Engineering Research Center.

Table of Contents

List of Figures

to be the core of the entire process. Before the level of automation can be significantly increased over current levels, integrated systems that can perform many more of the tasks required for design than is possible with existing artificial systems, will have to be developed.

The knowledge requirements of a complex activity such as chemical process design are enormous. Endowing a system with the knowledge needed to perform this activity, an enterprise referred to as knowledge engineering, can be a laborious, time-consuming and costly effort. Many person-years of work may be required. Furthermore, it is a task that never really ends. The knowledge required to perform design, as is needed to perform any other activity, is dynamic. That which is imparted to a system at creation time may not be adequate at problem solution time. This implies a need by a design system of an ability to learn. Learning can aid in two ways. First, it can help alleviate the knowledge acquisition bottleneck alluded to above, thereby significantly reducing, and perhaps even completely eliminating, the knowledge engineering effort. Second, it can allow design systems to improve with experience, thus enabling them to increase the efficiency and effectiveness with which they perform their tasks.

Recent work in the fields of artificial intelligence (AI) and cognitive science has resulted in the development of a number of integrated software architectures that combine problem solving and learning in their functionality. One of these systems, Soar [Laird *et al* 87] has already demonstrated its potential in both these functions.

The primary motivation of our research has been to understand the issues involved in building artificial design systems that integrate both learning and problem solving in their performance. This is crucial if machine learning is ever to play a role in chemical process design. Since Soar has already shown itself to be a powerful problem-solving engine and its learning mechanism, chunking, has been applied in a wide variety of learning situations, it was decided to conduct our experiments within the framework provided by this architecture. The work presented here describes our experiences thus far in this arena. We report on two systems that were constructed using the Soar architecture. The first, CPD-Soar, is a system for the design of unintegrated distillation sequences. The second, Interval-Soar, is a system that can learn the intersection point of two arbitrary and unknown functions. In this latter system, we solve what appears to be an extremely easy problem and discover that we need a few hundred production rules to do it. We will try to make clear the actual complexity of the learning activity required. A description of each system is provided together with its performance. We also indicate how the functionality of a system like Interval-Soar could be used to further improve the performance of a chemical design system such as CPD-Soar.

2 The Soar Architecture

Soar is a general model of human cognition [Newell 89] that has been implemented as a software system. Problems ranging from traditional AI "toy" problems such as the eight-puzzle and the Tower of Hanoi to complex real world knowledge-intensive tasks such as a portion of the computer configuration performed by the R1 (XCON) expert system [Rosenbloom *et al*

85] have been solved by Soar systems. The system has five key architectural features: problem spaces, recognition memory, decision cycle, impasse-driven subgoaling and chunking. We shall illustrate these features later when we step through the execution of CPD-Soar and Interval-Soar.

2.1 Problem Spaces

All tasks in Soar are formulated as search in problem spaces. A problem space consists of a set of states and a set of operators. Applying an operator to the current state generates a new state and a goal is achieved when a desired state is reached. The task is accomplished when the top-level goal is attained. Multiple goals correspond to a task decomposition and each of these may require different problem spaces to be searched. All search is realised by two generic functions: task-implementation and search-control. Task-implementation functions involve the retrieval or generation of problem spaces, states and operators. Search-control functions, on the other hand, involve the selection of objects (problem spaces, states, operators) from among those competing.

22 Recognition Memory

All long-term knowledge in Soar is stored in an associative recognition memory, realised as a production system. All knowledge about the current problem-solving situation is stored in working memory as a collection of data elements. Each production consists of a set of conditions and a set of actions. The conditions test working memory for the presence or absence of simple patterns (attribute-value elements) whereas the actions add new elements to it Productions encode all knowledge required to perform a task. This knowledge can pertain to either task implementation or search control. An important characteristic of Soar as a production system is the absence of any conflict resolution. All productions that are instantiated, i.e., have their conditions satisfied, are selected to fire, thus allowing the retrieval of knowledge in parallel. Although the attribute-value representational scheme employed by Soar is basic, both attributes and values may be other objects, hence complex frame-like structures can be constructed.

23 Decision Cycle

All problem solving in Soar revolves around a number of decisions; what goal should be attained, what problem space should be searched to attain the goal, what state should the search proceed from and what operator should be applied to the state. These decisions occur in a sequence of decision cycles, each of which consists of two phases. During the first phase, the elaboration phase, all instantiated productions fire. Since productions may create working memory elements that satisfy other productions, this process could continue for many elaboration cycles. It terminates when it runs to quiescence, i.e., when no more productions can fire. Elaboration results in two kinds of data elements being added to working memory. The first kind are new task-implementation objects such as problem spaces, states and

operators or augmentations to existing objects. The second kind are preferences. Preferences are special elements that encode knowledge about the acceptability and desirability of problem spaces, states and operators for any role in the total problem-solving context. This selection of an object for a role is made during the second phase of the decision cycle, the decision procedure phase. Begining with the oldest goal, the decision procedure considers each slot in the goal-context-stack. Within a context, the problem-space role is considered first, followed by the state and operator roles respectively. All preferences relevant to a slot are gathered and interpreted to determine an object for its role. If a unique decision can be made for an object for one of the slots in the context hierarchy, that object will be selected. This act signifies the end of the current decision cycle and problem solving then proceeds with the elaboration phase of the next cycle.

2.4 Subgoals

A situation may arise in the problem solving when a unique decision cannot be made for any of the slots in the current context. This may be due to either incomplete or inconsistent information. Soar deals with such a situation, known as an impasse, by subgoaling. This happens dynamically. Furthermore, subgoals may occur within subgoals, thus resulting in a hierarchy. The architecture recognises four kinds of impasses: rejection, no-change, tie and conflict To illustrate some of these, consider the following examples. An operator tie impasse occurs when several operators have been made acceptable, but not enough knowledge exists to select one. A state no-change impasse occurs when a state has been selected, but no operators are proposed to apply to it The operator tie impasse would be resolved when preferences that allow Soar to uniquely select one of the candidate operators are thrown into working memory. The state no-change impasse would be resolved when a preference for an operator that can be applied to the state is generated.

2.5 Chunking

Soar learns from its experiences in resolving impasses by constructing productions, known as chunks, for insertion into its long-term or recognition memory. The chunks summarise the problem solving that occurred in the subgoals and are created whenever results are generated. These results form the actions of the chunks. The conditions are the pre-impasse situation upon which these results depend. This requirement that the chunk conditions only be working memory elements that existed prior to the subgoal that were instrumental in generating the results is one way the chunks are generalised Another involves converting identifiers to variables, a process known as variablization. Also, results created in a subgoal whose function is solely to guide the research, i.e., search-control knowledge, are not used in the creation of chunks. These generalisation processes thus ensure that the chunks learned may apply in future problem-solving situations that aren't exactly the same as the ones they were created under, only similar. Like any other learning system, it is possible for Soar to acquire incorrect knowledge. This may result if the system either makes an incorrect inference or receives incorrect information. Laird [Laird 88] has described how Soar systems can recover

from any incorrect knowledge they may have captured in their long-term memories.

3 CPD-Soar

CPD-Soar is a system for the design of distillation sequences. It can currently create unintegrated sequences of sharp-splitting columns only. Given a feed specification, it first generates the splits that have to be applied to the feed stream to create the desired products and then sequences these splits. Multicomponent products can be handled. Search is controlled through the application of a number of heuristics commonly used in distillation sequence design.

3.1 Problem Spaces

There are eight problem spaces in CPD-Soar design[1], feed, order, split, sequence, update, output and selection. All of them, except design and selection are operator-implementation spaces. The top space, design, has eight operators: *get-feed, order-components, link-components, make-splits, identify-forbidden, sequence-split, update-stream* and *write.* Figure 1 shows the decomposition of the system into its problem spaces and operators.

The *get-feed* operator interacts with the user to obtain the feed specifications. This operator is implemented as a problem space called feed which contains two operators, *make-feed* and *get-comp. Make-feed* prompts the user for the name of the feed stream and the number of components. *Get-comp* obtains the following information about each component from the user flowrate, volatility and product in which it is desired.

Order-components ranks the components in a stream in descending order according to volatility. The lightest component, i.e., the one with the highest volatility, is given a rank one. The *order-components* operator is instantiated for all streams that are unordered and is implemented as a problem space called order. This space contains a single operator, *rank,* which is instantiated for all unranked components in the selected stream.

Streams are represented as linked lists and columns are modelled as list-splitters, i.e., as perfect splitters. Each component in a stream, other than the one with the highest rank, has an attribute "lighter-than" whose value is the identifier of the component that is adjacent to and heavier than it The operator *link-components* links all the components in a stream that has already had its components ordered.

Make-splits generates all the possible sharp splits that can be applied to the feed stream. For a stream with N components, the number of possible sharp splits is (N - 1). Each split is

---

*We will use the convention of **boldface TfmesRotnan** font to denote problem spaces and *boldface italics* font to denote operators.

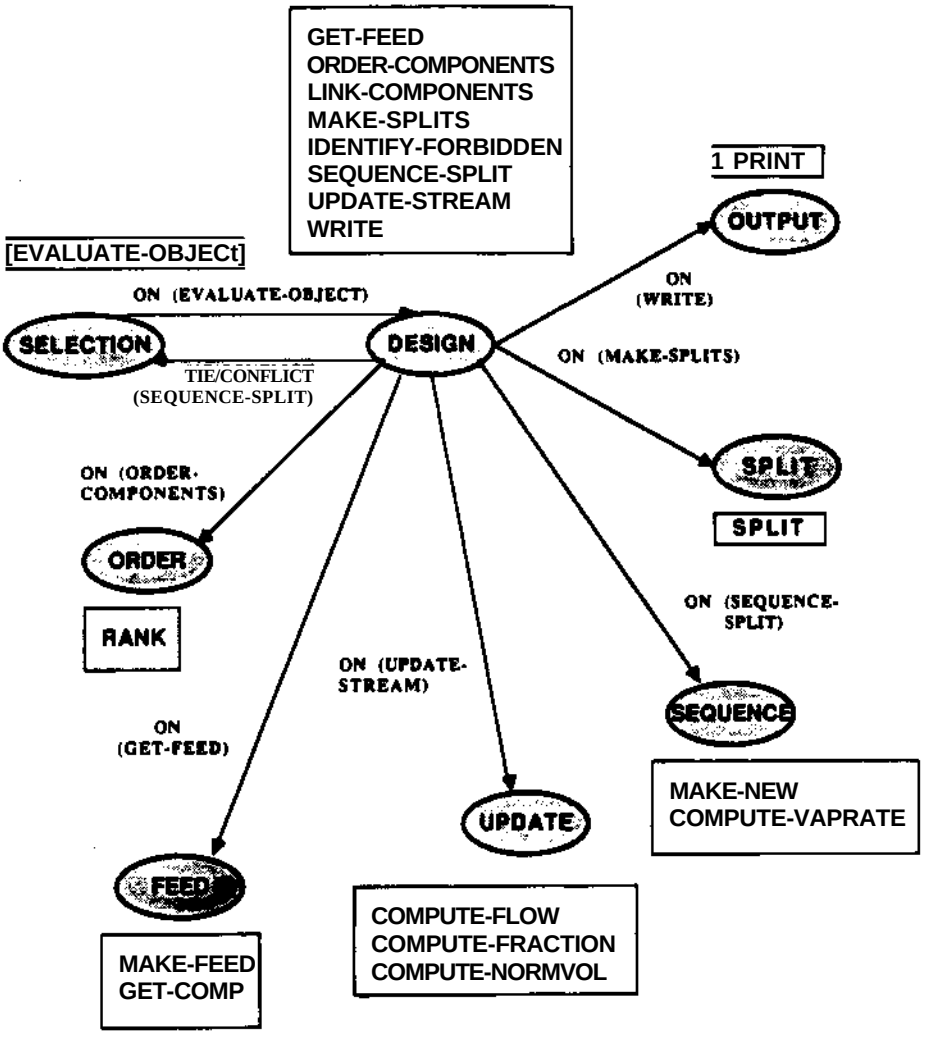

Figure 1: Hierarchy of Problem Spaces and Operators in CPD-Soar[2]

characterised by a light and heavy key. The **split** problem space implements the *make-splits* operator. This space also contains a single operator with the same name which generates a split and computes the ratio of the volatilities of the light and heavy keys. *Identify-forbidden* tags

[2]**Each problem space is depicted by an oval and the operaton it contains are listed in the box next to iL Impasses arc noted next to the directed lines linking problem spaces. (ON refers to an Operator No-change impasse).**

each of the splits generated by *make-splits* as either allowed or forbidden. A split is forbidden if its two keys coexist in the same product

The operator *update-stream* computes the mole fractions of a stream's components, normalises their volatilities with respect to the heaviest component and computes the total flowrate of the stream. It is implemented as a problem space called update, which contains three operators: *compute-flow, compute-fraction* and *compute-normvol.*

When all the allowable splits have been sequenced, the *write* operator writes them to the screen in the order they are to be applied. The operator is implemented as a problem-space called output. This space contains a single operator called *print* which outputs a split to the screen.

The selection space is selected to resolve ties or conflicts among competing problem spaces, states or operators. It contains one operator, *evaluate-object,* whose function is to compute an evaluation for a competing object The next section describes the use of the selection space to resolve ties or conflicts among competing *sequence-split* operators in CPD-Soar.

3.2 Resolving Ties and Conflicts among Splits

The above described operators are all concerned with routine book-keeping and input/output functions required by the design problem. The most important operator is *sequence-split.* This ranks all the allowable splits in the order they are to be applied. It is implemented as a problem space called sequence which contains two operators, *make-new* and *compute-vaprate.* The *make-new* operator performs a number of functions. It generates two new streams corresponding to the distillate and bottoms products and augments these streams with their components. It also generates a column that is augmented with its feed and product streams. *Compute-vaprate* computes the vapour flowrate in the column using a simplied function that Douglas [Douglas 88, p. 463] has proposed for a binary column.

The *sequence-split* operator is made acceptable for all unranked allowable splits that may be applied to streams that have not already been split and are not products. Splits that apply to different streams are made indifferent to each other. The following heuristics are used to select among splits that apply to the same stream: easiest separation best (similar to hardest separation worst), removal of lightest key best and removal of component with largest flowrate best Since these heuristics discriminate on the basis of different attributes, conflicts are to be expected in many cases. Most previous works in the field have dealt with this problem in one of a number of ways. One approach involves ranking the heuristics in order of importance. In all cases however, the ranking function used is very subjective and usually does not have any basis. A second approach does away with the use of heuristics altogether. Instead, evaluation functions are employed to rate all the competing splits. However, this is a policy that can prove to be computationally expensive for even moderately sized problems. A third approach uses

only a small subset of all the heuristics that have been shown to be useful. This subset is selected carefully so as to avoid the possiblity of conflicts arising. However, this approach loses out in situations where the weeded-out heuristics could have applied. Soar's ability to deal with inconsistent and incomplete information by subgoaling avoids the weaknesses of all these approaches. The power of using heuristics as a means of controlling search is exploited, and only when the heuristics are in conflict, or result in ties, are the evaluation functions employed. CPD-Soar resolves conflicting and tieing splits by applying in turn, each split to the stream, generating the product streams and the column, and computing the vapour flowrate in the column. The split resulting in the column with the lowest vapour rate is selected. In other words, the search strategy employed in tie or conflict instances is a simple one-step lookahead.

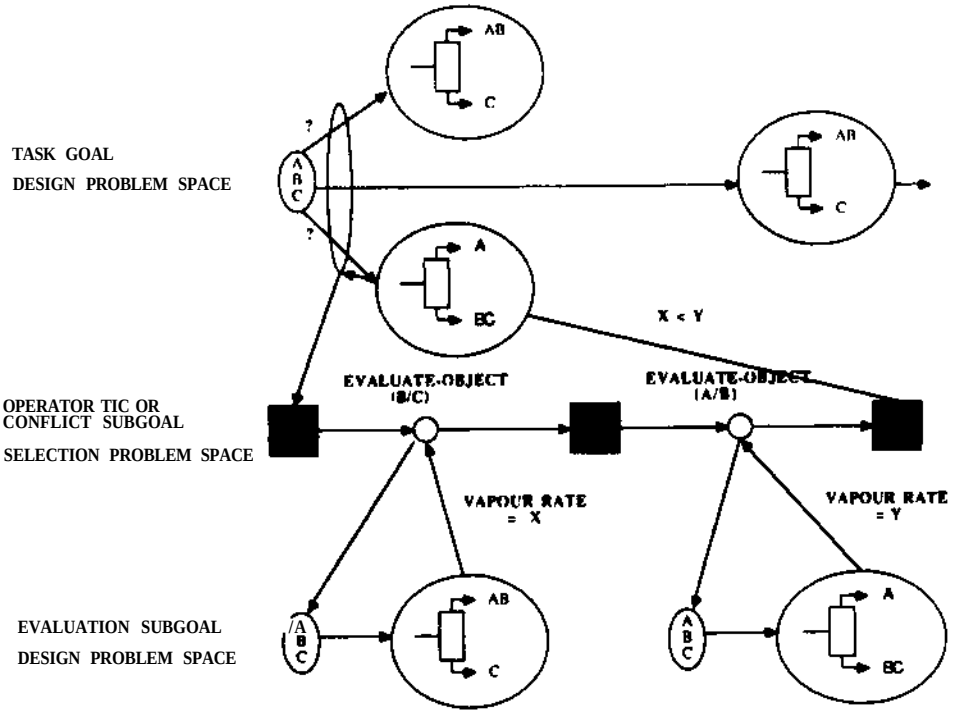

Figure 2: Selecting among Competing Splits in CPD-Soar

Figure 2 depicts the subgoaling CPD-Soar would undergo in the case of a tie or conflict impasse. Consider the simple, but representative, situation in which a stream consisting of three components, A, B and C, is to be split Suppose that CPD-Soar, on the basis of its current knowledge, i.e., its heuristics, is unable to decide between the two possible splits, A/B or B/C.

This could be due to one of two reasons. Either its knowledge indicates both splits are equally good, in which case a tie impasse will be encountered, or its knowledge is conflicting, with one piece of information stating A/B is better and another stating B/C is better, in which case a conflict impasse will arise.

In both cases, CPD-Soar subgoals into the **selection** space to generate the knowledge required to resolve the impasse. In the **selection** space, the ***evaluate-object*** operator is made acceptable for each of the tieing or conflicting objects; in this case, the two sequence-split operators. Since it does not matter in which order the splits are evaluated, the *evaluate-object* operators are made indifferent to each other. CPD-Soar evaluates the competing splits by trying each one out in turn and comparing the results. Suppose the B/C split is selected first to be evaluated. If CPD-Soar encounters an operator no-change impasse in trying to apply the *evaluate-object* operator, the **design** space is made acceptable. The B/C split is applied to the stream and the vapour flowrate of the resulting column computed. Suppose it is X in this case. This knowledge is passed back to the **selection** space. CPD-Soar next performs the same sequence of operations for the second split Suppose the vapour flowrate for the A/B split is Y. The two evaluations, i.e., flowrates, are then compared in the selection space and a better-than preference is generated for the split corresponding to the smaller one with respect to the other. Since in this case X is smaller than Y, the split B/C is chosen.

3J **Performance of CPD-Soar**

CPD-Soar is an attempt to emulate the behaviour of a human process designer working without the use of any external computational aids except a device to perform simple arithmetic calculations. Hence its performance should not be measured relative to man-machine computing systems where the machine is an extremely powerful number cruncher. With this in mind, it would probably be fair to state that CPD-Soar's performance is acceptable.

Two points will be made concerning the strategy used by CPD-Soar to resolve ties or conflicts among competing splits. The first concerns the one-step lookahead search and the second concerns the vapour flowrate evaluation function.

The one-step lookahead search employed was decided upon for the following reasons. Since the heuristics are employed to select the next step, i.e., operator, to be applied, using each operator in turn and selecting which resulted in the best state was deemed sufficient in providing a functional equivalence to the knowledge encoded in the heuristics. Of course, the decision resulting from such a limited lookahead is only optimal locally. Searches that look beyond the first step are certainly possible. One potential search would be a lookahead to the completion of a sequence. It should be noted however, that the deeper a search is, the more expensive it will be computationally. The main motivation, however, for resorting to the one-step lookahead in the current version of CPD-Soar was to get an appreciation of the sort of chunks that would be generated.

In many instances the vapour flowrate is a good indicates- of the cost, both construction and operating, of a column. Lower vapour rates result in smaller columns and lower utility usage. Hence, it was decided to use this parameter as being indicative of the quality of the state generated. However, this does not imply that the strategy employed by CPD-Soar is dependent upon this evaluation function. For the design of heat-integrated sequences, this will probably change.

The chunks generated in response to the above-described impasses are too specific, i.e., their condition elements have numeric attributes whose values are specific constants. An example chunk is presented in Figure 3[3]. Such a chunk can be extremely useful within the same problem-solving trial. A chunk acquired earlier on in the problem solving can fire during later stages of the same trial. Such a transfer can help in making much larger design problems (than are routinely attempted today) tractable. In process design problems, it is usually the case that many computations have to be repeated several times over during the same problem instance. Also, decisions among the same competing choices may also be repeated. An example from the domain of distillation-sequence design would be the calculation of the column parameters for a particular split A search down one branch of the tree may require the A/B split to be evaluated. However, further down the branch a decision may be made not to explore it any further and to switch the search to another branch. This new branch may now also require the A/B split to be evaluated. However, since the design system would have chunked away the results of the A/B evaluation carried out during the search of the previous branch, this evaluation will not have to be repeated. In process design problems, it is also often the case that decisions from among the same competing choices may also have to be remade several times within a single problem instance. For example, while searching a particular branch of a tree, a decision may have to be made between two competing splits, say, A/B and B/C, which in turn may require their full evaluations. Later, when some other branch of the tree is being searched, it is possible that a decision may again be required between A/B and B/C. This time however, the system would be able to make a decision straight away based on the knowledge it had acquired the previous time. The example chunk presented encodes such knowledge.

However, the power of a learning mechanism lies in its ability to acquire knowledge that can be employed in a situation that is not exactly the same as the situation it was acquired in, i.e., its ability to generalize. Chunks that are more general in their applicability than those currently learned by CPD-Soar are possible. One kind include those that would refer to intervals or ranges rather than specific values of numbers. Such chunks, it is conceived, would prescribe domains within which volatilities, for instance, or flowrates, would have to fall in order for the chunks to apply. However, before CPD-Soar can acquire such chunks, an understanding of how intervals of numbers may be learned and represented within Soar systems has to be acquired. The following section describes another system, Interval-Soar, that

[3]The syntax of the production presented here differs from those presented later since CPD-Soar was developed using Soar 4, whereas Interval-Soar employs a newer version, Soar 5

```
(sp p43 elaborate
    (goal <g1> ^problem-space { <> undecided <p1> }
     ^state { <> undecided <n1> } ^desired <d1>)
    (desired <d1> ^better lower)
    (operator <o2> ^name sequence-split ^split <s1>
     ^stream <s3>)
    (split <s1> ^hk <h1> ^lk <l1> ^relvol 3/2)
    (hk <h1> ^name b)
    (stream <s3> ^component <c1> { <> <c1> <c2> }
     { <> <c2> <> <c1> <c3> })
    (component <c1> ^name b ^flowrate 2)
    (lk <l1> ^name c)
    (component <c2> ^name c ^flowrate 3)
    (operator { <> <o2> <o1> } ^name sequence-split)
    (operator <o1> ^split { <> <s1> <s2> } ^stream <s3>)
    (split <s2> ^hk { <> <h1> <h2> } ^lk { <> <l1> <l2> }
     ^relvol 2)
    (hk <h2> ^name a)
    (component <c3> ^name a ^flowrate 1)
    (lk <l2> ^name b)
  -->
    (preference <o2> ^role operator ^value worse
     ^reference <o1> ^goal <g1>  ^problem-space <p1>
     ^state <n1>))


  If   there is a sequence-split operator <o2> that
       implements split <s1>
 and   split <s1> has light key C and heavy key B
 and   the ratio of volatilities between B and C is 1.5
 and   C has flowrate 3
 and   B has flowrate 2
 and   there is a sequence-split operator <o1> that implements
       split <s2>
 and   split <s2> has light key B and heavy key A
 and   the ratio of volatilities between A and B is 2
 and   A has flowrate 1

then   create a worse preference for operator <o2> with
       respect to operator <o1>
```

Figure 3: Example of a Chunk Learned by CPD-Soar and its English Description.

possesses the ability to learn ranges for relatively simple problems. It is one of a series of systems we are developing within the Soar framework to explore the issues involved in understanding and developing systems that can perform useful learning in highly quantitative domains.

Our experiences with OPD-Soar have indicated that Soar as a problem-solving engine seems to provide all that has thus far been sought in an architecture for design. As noted earlier, the performance of CPD-Soar, as a system that attempts to emulate a human designer, is acceptable and its performance will certainly be better than a corresponding non-learning system.

4 Interval-Soar

Interval-Soar is a system that can learn the intersection point between two arbitrary and unknown functions, fl and f2. This learning is performed as a welcome side-effect of performing another task. The task consists of applying one of the two functions to each element in a set of data points and computing the result. The data point, a scalar denoted by x, is then labelled with the function that was applied to it When all data points have been labelled, the task is considered accomplished.

Each function is characterized by a parameter called its bound. For function fl, this is ub-ol (upper bound of operator 1) and for function f2, it is Ib-o2 (lower bound of operator 2)[4]. The bounds of the operators thus demarcate the intervals or regions that the operators should be applied in. The value at which the two bounds, ub-ol and Ib-o2, are equal, will correspond to the intersection or crossover point between the two functions.

If a data point falls within a particular interval, then the operator to which that interval corresponds can be selected to apply to the data point. If a data point does not fall within an interval, then both operators will have to be fully evaluated in order to make a decision about which to select. If, after a full evaluation, *opl* is selected as the operator to apply, the data point will be learned as the new value of ub-ol. Conversely, if *opl* is selected, the value of Ib-o2 will be updated to the data point. By such a process of refining the values of the bounds, it is possible for Interval-Soar to incrementally converge to the intersection point of the two intervals.

To perform the above-described task is not as simple as it may seem to be. The main reason for this is that Interval-Soar must carry out this task without any knowledge of the functions themselves[5]. Thus, it should be emphasized, it is not a question of solving two equations in two unknowns. Instead, it is a problem of determining the intersection using knowledge of a sample set of data points only. One assumption that Interval-Soar currently makes is that the functions only intersect at a single point It is expected that future versions of the system will relax this assumption. In order to perform this task, as will become clearer later, the system must possess the ability to recognize and recall declarative knowledge.

[4]The lower bound of operator 1 is assumed to be minus infinity and the upper bound of operator two is assumed to be plus infinity.

[5]The example problem described later shows the two functions since they are used for the purposes of illustration. However, Interval-Soar does not have accesss to this knowledge per se.

However, since chunking is the only learning mechanism within Soar, this memorizing of declarative knowledge must be performed using it, and this is not so straightforward.

The problem spaces in Interval-Soar can be categorized into three major groups: the task spaces, the function spaces and the memory space. Figure 4 shows the decomposition of the system into its problem spaces.

4.1 Task Spaces

There are three task spaces in Interval-Soar the interval space, the selection space and the refine space.

The interval space is the top-level space. It contains three operators: *select-x, opl* and *op2. Select-x* selects a data point to be classified from among all those still unlabelled. *Opl* and *op2* implement functions f 1 and f2 respectively.

The selection space is chosen in response to a tie between operators *opl* and *opl* in the interval space. It has four operators: *memory, x-lte-ub-ol*$_y$ *x-gte-lb-o2* and *refine-interval. Memory* is used to retrieve the current values of the bounds, ub-ol and Ib-o2. *X-Ue-ub-ol* and *X'gte-lb-o2* are comparison operators. The first compares x to ub-ol and if it is less than or equal to the bound, returns the value true. The second compares x to Ib-o2 and in this case returns true if x is greater than or equal to the bound. The operator *refine-interval* is used to refine the values of the bounds on the intervals. It does this by carrying out a full evaluation of the operators *opl* and *op2,* comparing the evaluations and updating the bound of the operator that has the higher evaluation.

The refine space, which implements the *refine-interval* operator, contains five operators: *opl, op2*$_9$*fl-eq'f2,fl-gt-f2* and *memory. Fl-eq-f2* returns a value of true if fl is equal to f2. Likewise, *fl-gt-/2* returns a value of true if f 1 is greater than f2. In contrast to the *memory* operator in the selection space, the *memory* operator in the refine space associates the value of a bound with its corresponding cue, i.e., it learns a new bound value.

4.2 Memory Space

The chunking mechanism within the Soar architecture has been demonstrated to be adequate for the acquisition of procedural knowledge[7]. On the other hand, the acquisition of declarative knowledge[8] is not so straightforward. Although the chunking mechanism can be used to acquire such knowledge, the system must first perform some deliberate processing in

---

[7] Procedural knowledge includes knowledge about which actions the system can perform, when certain actions should be preferred over others and how to carry out the actions.

[8] Declarative knowledge includes facts. It is knowledge about what is true in the world.

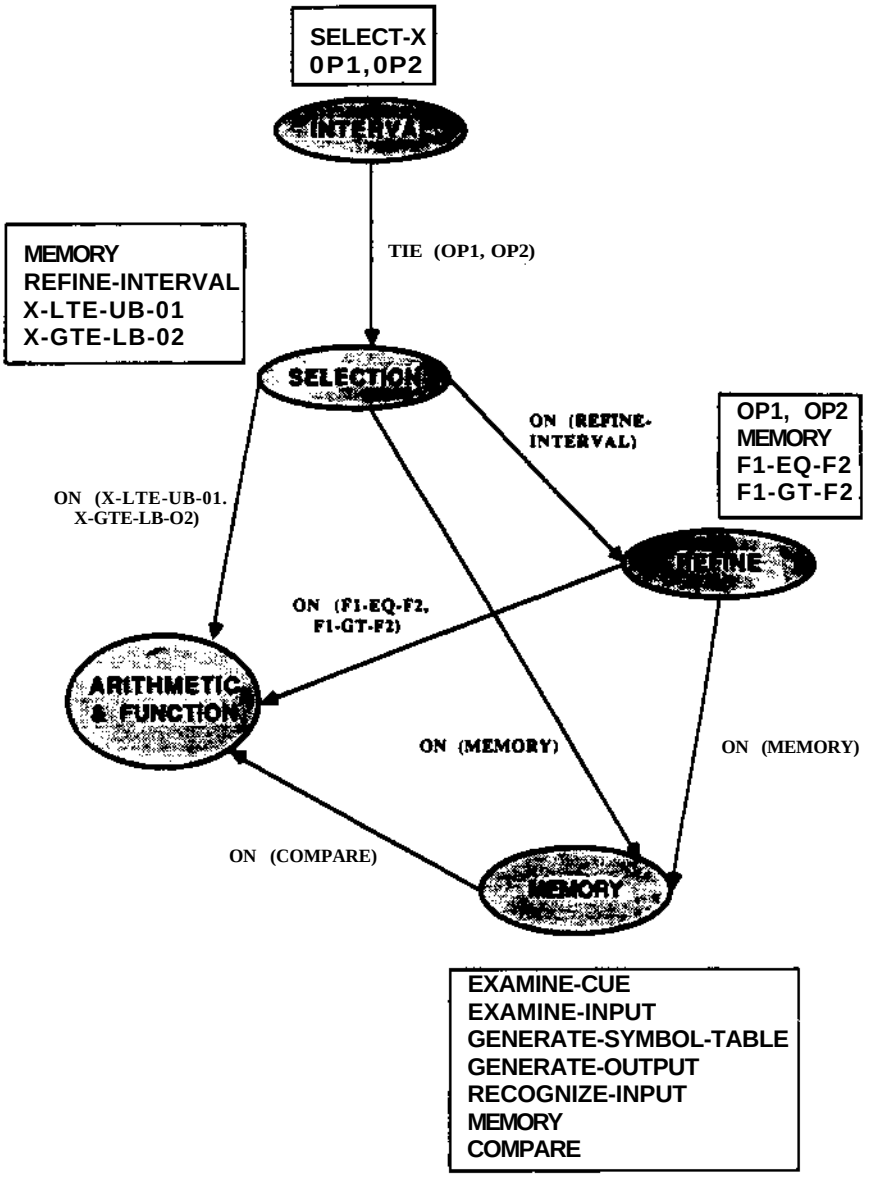

Figure 4: Hierarchy of Problem Spaces in Interval-Soar[6]

[6]Each problem space is depicted by an oval and the operators it contains are listed in the box next to it Impasses are noted next to the directed lines linking problem spaces. (ON refen to an Operator No-change impasse).

order to do so. This processing occurs in the memory space, which implements the *memory* operator. An involved discussion of representing, storing, retrieving, using and acquiring different forms of knowledge, including both procedural and declarative, has been provided by Rosenbloom *et al* [Rosenbloom *et al* 89],

The *memory* operator provides Interval-Soar with the means to memorize and retrieve declarative knowledge. It provides an ability to learn to recognize and learn to recall objects. The operator takes two arguments: an input object (the object to be learned) and a cue object. The cue constrains the situations in which the input object is to be retrieved. When the *memory* operator is applied, all objects that were previously associated with the given cue are recalled. If the input object is not among those retrieved, then it is learned (and will thus be retrieved the next time the *memory* operator is applied with the same cue). The absence of a cue is effectively taken to be the cue if the memory operator is applied without one. The operator will only perform retrieval of earlier memorized objects if it applied without an input object. However, if no objects had been previously associated with the cue, then nothing is retrieved and the operator is simply terminated.

The following example will hopefully serve to provide a clearer description of the functioning of the *memory* operator. Suppose we wish to associate the objects "Fido" and "Bozo" with the cue "dog." This means that whenever the system is presented with the cue "dog," the objects "Fido" and "Bozo" should be recalled. This association, which is a form of memorization, is carried out by selecting and applying the *memory* operator, in this case, twice. Suppose the first time the operator is applied with the cue "dog" and the input object (or object to be learned) "Fido." The result of applying the *memory* operator will be a chunk that delivers, i.e., retrieves, the object "Fido" on any future occasion that the operator is applied with the cue "dog." Suppose the operator is applied for a second time. However, this time we wish to associate the object "Bozo" with the cue "dog." This application will result in the object "Fido" being retrieved (since it was previously associated with the given cue) and the object "Bozo" being memorized, i.e., a chunk being created that will, on future occasions, deliver the object "Bozo" when the cue "dog" is presented. If the *memory* operator is applied for a third time with the cue "dog" and no object to be learned, then only retrieval of the objects "Fido" and "Bozo" will occur. An application of the *memory* operator with the cue "cat" will not retrieve anything since no objects have been associated with that cue. In Interval-Soar, the cues are "ub-ol" and "Ib-o2^n and the learned objects are the values of the bounds.

The application of the *memory* operator results in the learning of two kinds of chunks: recognition chunks and recall chunks. The recognition chunks allow the system to determine if it has seen an object before and the recall chunks allow it to generate a representation of an object seen before.

The *memory* operator is implemented as the memory space. This problem space contains seven operators: *examine-input, examine-cue$_y$ generate-symbol-table, generate-output, recognize-input, memory* and *compare.* The *examine-input* and *examine-cue* operators cycle

through all the symbols the input and cue objects are respectively constructed of so that tests of these symbols appear in the recognition chunks learned. *Gene rate-symbol-table* creates a symbol table relating the input symbols to the output symbols, which *generate-output* uses to construct the output object from. *Recognize-input* recognizes the input object and the input-cue pair. The *memory* operator augments the parent *memory* operator (which was implemented as the memory space) with the recalled object The *compare* operator compares the previously learned bound (which has now been retrieved) with the current bound to be learned. If they are not equal (which will always be the case since the interval is being refined), a reject preference is generated for the old bound The memory space is complex and subtle and the above is a very basic description of the functions of its operators. A more detailed description of the space has been provided by Rosenbloom [Rosenbloom 89].

4J Function Spaces

The function spaces contain knowledge about performing basic logical, arithmetic and control functions. This knowledge allows Interval-Soar to symbolically execute the mathematical functions it needs, such as computing f 1 and f2, comparing f 1 to f2, comparing x to ub-ol and ub-o2 and comparing the new value of a bound with an old one, all without recourse to an external computing device. A detailed description of the function spaces has been given by Rosenbloom & Lee [Rosenbloom & Lee 89].

4.4 Example Task

The functioning of Interval-Soar will now be illustrated by a simple example. Consider the two functions, fl = 7 - x and f2 = x + 1, and consider three data points: x = 1.3, x = 3.0 and x = S.8. As described earlier, the task is to apply one of the functions to each of the data points, compute the results and label the points with the names of the operators corresponding to the functions that were applied to them. As a by-product of this problem solving, the system must learn the values of two bounds, ub-ol and Ib-o2. These parameters demarcate the intervals where the functions should be selected. If a data point has a value less than or equal to ub-ol, f 1 should be applied. If it has a value greater than or equal to Ib-o2, f2 should be applied.

Problem solving begins in the interval space. The initial state consists of a set of (three in this case) unlabelled data points. The desired state is one in which each data point has had its function value computed and is labelled. The operator *select-x* is first applied to choose a data point to be worked on. Since the order in which the data points are selected is irrelevant, they are all made indifferent to each other. Once a point has been selected (suppose in this case it is x = 5.8), operators *opl* and *op2* are proposed to apply. *Opl* implements fl and *op2* implements f2. Since both operators are equally acceptable at this stage, a tie impasse results.

To resolve the tie between *opl* and *opl* ŕ the interval space, a subgoal is created and the selection space is chosen. In the selection pace, Interval-Soar first tries to retrieve any existing bounds on the tieing operators. It does this by proposing the *memory* operator twice,

one with the cue ub-ol and the other with the cue Ib-o2. Since the order in which the bounds are retrieved is irrelevant, the two memory operators are made indifferent to each other. However, nothing is retrieved because no values have yet been associated with the cues since this is the first time Interval-Soar is solving the problem.

Thus, to make a decision that resolves the impasse, a full evaluation of the tieing operators must be made. To do this, Interval-Soar selects the *refine-interval* operator. If there is an operator no-change impasse in attempting to apply the *refine-interval* operator, the refine space is selected. The schemes used to evaluate the operators *opl* and *opl* are just the functions themselves. Thus, *opl* and *op2* are first applied in random order to the data point In this case fl = 1.2 and f2 = 6.8. Next, the comparison operators *fl-eq-J2* and *fl-gt-fl* are selected and applied in turn to determine the relative magnitude of fl with respect to f2. *Fl-eq-fl* returns a value of false (indicating the two parameters are not equal) *and fl-gt-gl* returns a value of false (indicating that f 1 is not greater than f2). Before this knowledge is passed back to the higher-level spaces, the value of Ib-o2 (since f2 is greater than fl) is memorized as 5.8. Interval-Soar carries this out by selecting and applying the *memory* operator with the cue object as Ib-o2 and the learned object as 5.8. The knowledge that f2 is greater than fl is now passed back to the higher spaces to resolve the initial tie between *opl* and *op2*. Interval-Soar thus applies *op2* to the data point x = 5.8, which is consequently labelled op2. The state of affairs at this stage of the problem solving is depicted in Figure 5.

The entire problem-solving behaviour is now repeated for another data point There are a few differences since Interval-Soar uses some of the knowledge it had chunked away when running the first point. Suppose the point x = 3.0 is selected this time. The selection space is again chosen in response to a tie between *opl* and *opl* in the interval space. In the selection space, the *memory* operators first apply to retrieve any bounds, i.e., any numbers associated with the cues ub-ol and Ib-o2. Since 5.8 has been associated with Ib-o2, it is recalled instantly. The comparison operator *x-gte-lb-ol* is next selected and applied to determine the relative magnitude of the data point with respect to the bound. Since x is not greater than or equal to Ib-o2 in this case, a value of false is returned. This knowledge does not allow a decision to be made between the competing operators; hence, the *refine-interval* operator is selected to compute a full evaluation once again. *Opl* and *op2* are applied in random order in the refine space. In this case, both f 1 and f2 are determined to be 4.0. The operator *fl-eq~f2* is next applied and returns a value of true. Again, before this knowledge is passed back to the higher spaces to resolve the tie, the values of the bounds are updated by applying the *memory* operator. The value of ub-ol is memorized to be 3.0 by associating the number 3.0 with the cue ub-ol. In the case of Ib-o2, the process is slightly different Since the number 5.8 is already associated with the cue Ib-o2 (from the previous run), the number associated with the cue is updated to 3.0[9]. This updating is performed in the memory space (which is selected in response to a no-change impasse for the *memory* operator) by applying the *compare* operator.

*In Soar, this is equivalent to having a chunk that generates a reject preference for 5.8 and another chunk that generates an acceptable preference for 3.0.

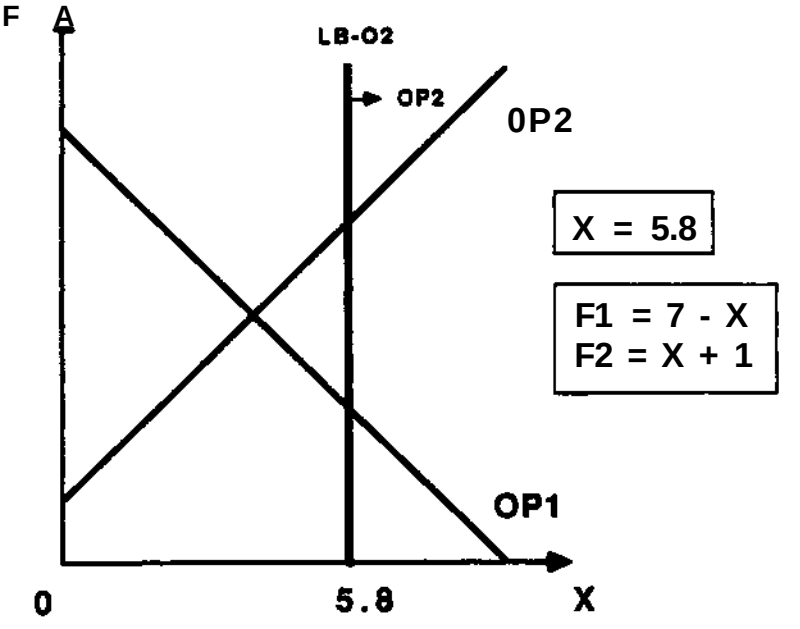

Figure 5: Location of Bounds after First Data Point is Employed

This operator compares the new value of the bound (the number to be learned) with its old value. If they are not equal (which, as noted earlier, will always be the case since the interval is being refined), a reject preference is generated for the old bound. After the memorization process is completed, the knowledge that fl is equal to f2 is passed up to the higher spaces. In this situation, *opl* and *opl* will be made indifferent to each other and one will be picked at random. Figure 6 depicts the problem solving situation at this stage.

The final point from the set to be labelled is x = 1.3. By now the sequence of steps taken to achieve this should hopefully be clear. In the selection space, *memory* operators are first applied to retrieve the current values of the bounds. In this case, both ub-ol and Ib-o2 are associated with the number 3.0. Next, the comparison operators, *x-lte-ub-ol* and *x-gte-lb-o2* are selected to apply. The first returns a value of true since x is less than ub-ol, while the second returns a value of false since x is less than Ib-o2. Since Interval-Soar possesses the knowledge that if a data point is less than the bound ub-ol, operator *opl* should be selected, a better-than preference is generated for *opl* with respect to *op2.* Hence* in this case, the tie impasse in the interval space can be resolved without resorting to a full evaluation of the competing operators, as was done during the two previous trials. The situation at this stage is

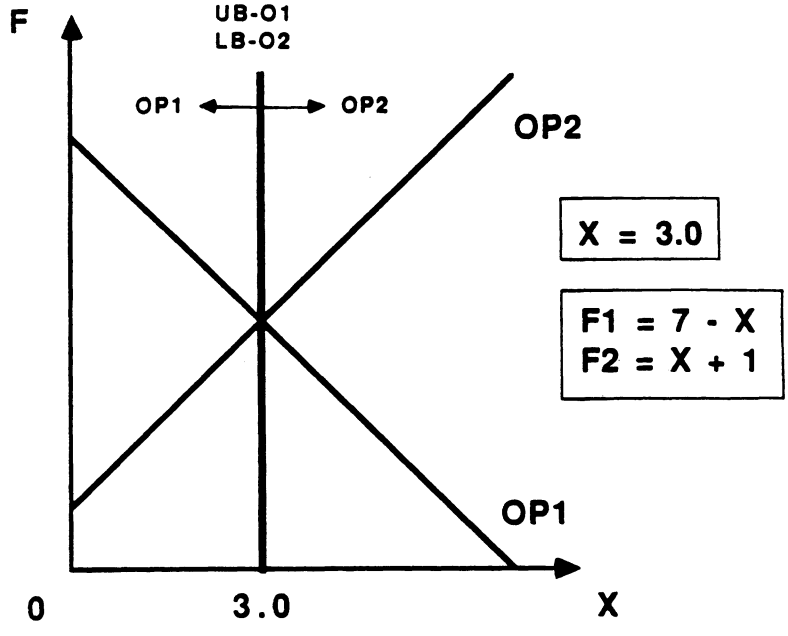

Figure 6: Location of Bounds after Second Data Point is Employed

depicted in Figure 7.

It should be noted that operator no-change impasses are encountered when attempting to apply ***op1***, ***op2***, the comparison operators (***f1-eq-f2***, ***f1-gt-f2***, ***x-lte-ub-o1***, ***x-gte-lb-o2*** and ***compare***) and the ***memory*** operator. The **memory** space is selected in the case of the ***memory*** operator and the function spaces are selected for the others.

4.5 Performance of Interval-Soar

To illustrate the performance of Interval-Soar, the results from running the system using three different sets of data points will be presented. Each set consists of three points: set 1 is (1.3, 3.0, 5.8), set 2 is (1.8, 3.7, 6.6) and set 3 is (2.4, 4.5, 6.9).

Across-trial transfer occurs when chunks acquired when solving one instance of a problem apply when another instance of the same problem is executed. Table 1 illustrates the effects of across-trial transfer. The results presented there include the changes in decision cycle numbers for each test case as well as the average over all three sets. As can be seen, the

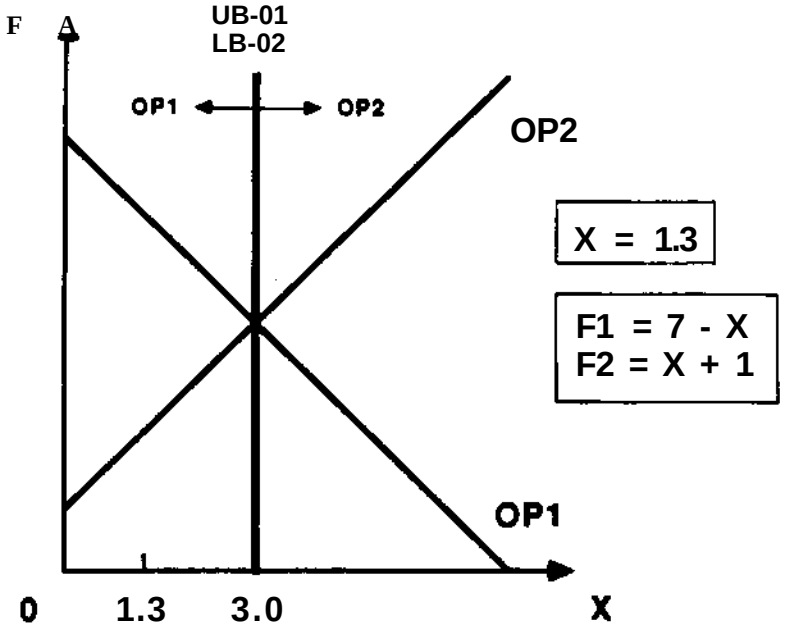

Figure 7: Lxxation of Bounds after Third Data Point is Employed

benefits of learning are encouraging. For the first trial, the average number of decision cycles required was 355. For trial two, this dropped to 73, a percentage drop of 79.4 and for trial three, this further dropped to 9 for a total percentage drop of 97.5 over the first trial.

Trial Case	1	2	3
1	317	57	9
2	374	81	9
3	374	81	9
Ave	355	73	9

Table 1: Effects of Aero ->-Trial Transfer of Chunks: Changes in Numbers of Decision Cycles

Table 2 shows the number of productions learned by the system over the course of each

of the three trials. In all cases the system begins with 934 productions. A Soar system that learns on all goals during a trial will normally not acquire additional knowledge during subsequent trials. This is because the system will have learned all that it can during the first trial. The behaviour of Interval-Soar however, is an interesting example of how a system that learns on all goals during a first trial can learn additional knowledge during a second trial. This occurs in Interval-Soar since knowledge acquired during the first trial causes it to carry out a different problem-solving process in the second trial. This new process creates a different goal hierarchy, thus allowing the system to acquire knowledge that was not acquired through the original problem-solving process. To illustrate this, consider the data points in example set 1. At the end of the first trial, Interval-Soar learns that the values of ub-ol and Ib-o2 are both 3. During the second trial, this knowledge is brought to bear. When Interval-Soar attempts to decide which function to apply to 5.8, the first point in the set, it compares this number with the retrieved bounds instead of carrying out a complete evaluation as it did during the first trial. This comparison process requires Interval-Soar to subgoal into the arithmetic and function spaces (rather than the refine space) and the chunking that takes place over these spaces thus allows the system to acquire additional knowledge during a second trial. For case 1, the system acquires 65 chunks during the first trial and 10 chunks during the second trial. No chunks are acquired during the third trial since the system has learned all that it can. During this trial, no subgoaling occurs since productions learned in the previous trials fire to prevent all impasses.

Trial Case	1	2	3
1	65	10	0
2	76	15	0
3	76	15	0

Table 2: Numbers of Productions Learned during Different Trials

Table 3 illustrates the effects of across-task transfer. This occurs when chunks learned when solving a problem in a particular domain apply during the solution of another problem within the same domain. In the case of Interval-Soar, chunks acquired when using the data from set 1, for instance, fire when performing the task using the data from set 2 or set 3. Again, the benefits of learning are encouraging. Each data set was run independently with the chunks acquired during the running of the other two data sets and in each case, the number of decision cycles required was less than the number needed when no imported chunks were used. The percentage decrease in decision cycles ranged from 12.6 (when running set 1 with chunks learned during the running of set 3) to 71.4 (when running set 2 with chunks learned during the running of set 1). The average decrease over all 6 runs was 38.1%.

Case	No. Decision Cycles	% Decrease in DC
1 with no imported chunks	317	
1 with chunks imported from 2	252	20.5
1 with chunks imported from 3	277	12.6
2 with no imported chunks	374	
2 with chunks imported from 1	107	71.4
2 with chunks imported from 3	257	31.3
3 with no imported chunks	374	
3 with chunks imported from 1	169	54.8
3 with chunks imported from 2	232	38.0

Table 3: Effects of Across-Task Transfer of Chunks

4.6 Analysis of Knowledge Learned by **Interval-Soar**

As described earlier, chunks acquired during problem solving prevent the system from subgoaling should the same or similar situations arise in the future. Most of the chunks learned in Interval-Soar are operator-implementation productions. These are productions that fire to directly implement an operator in particular situations. Before learning, such an operator, because of its complexity, would require subgoaling in order to be applied. Table 4 is a summary of the number of operator-implementation productions learned for the different operators in Interval-Soar.

Operator	No. Implementation Chunks Learned
opl	7
op2	5
x-lte-ub-ol	4
x-gte-lb'02	4
memory	3
fl-eq-fl	3
fl-gt-f2	4
compare	3

Table 4: Numbers of Operator-Implementation Chunks Learned for Different Operators

The *memory* operator-implementation chunks deserve special attention. As noted earlier, the purpose of applying the *memory* operator is to either recall or memorize declarative knowledge. These chunks allow the system to recognize the input object, i.e., the object to be learned, to recognize the combination of cue and input objects and to retrieve any previously learned objects associated with the given cue. Typical examples of these chunks are presented in Figure 8.

```
;; Recognize the combination of cue and input objects if the
;; cue is Ib-o2 and the object to be learned is the number 5.
(sp p293 elaborate
   (goal <gl> ^operator <ol>)
   (operator <ol> ^name memory ^cue <c2> ^learn <yl>)
   (class <c2> ^name Ib-o2)
   (param <yl> ^value <il>)
   (integer <il> ^sign positive ^head <d> ^tail <d>)
   (column <cl> ^anchor head tail ^digit <dl>)
   (digit <dl> ^name 5)
 -->
   (operator <ol> "recognized cue-input £, cue-input +))

;; Recognize the object to be learned if it is the number 5.
(sp p294 elaborate
   (goal <gl> ^operator <ol>)
   (operator <ol> -^recognized input ^name memory
    ^learn <yl>)
   (param <yl> ^value <il>)
   (integer <il> ^sign positive ^head <d> ^tail <d>)
   (column <cl> ^anchor head tail ^digit <dl>)
   (digit <dl> ^name 5)
 -->
   (operator <ol> ^recognized input 6, input +))

;; Augment the memory operator with the recalled object, which
;; is the number 5.
(sp p295 elaborate
   (goal <gl> ^operator <ol>)
   (operator <ol> ^name memory ^cue <cl>)
   (class <cl> ^name Ib-o2)
 -->
   (integer <x3> ^sign positive + ^head <xl> + ^tail <xl> +)
   (digit <x2> ^naroe 5 +)
   (column <xl> ^digit <x2> +
    ^anchor head + head 6, tail 6, tail +)
   (param <x4> ^value <x3> +)
   (operator <ol> ^recalled <x4> =, <x4> +))
```

Figure 8: Example Memory Operator-Implementation Chunks

Chunks are also acquired that reject previously learned values of the bounds. An example of such a chunk is given in Figure 9.

```
;; Reject the recalled object if it is the number 5.
(sp p593 elaborate
   (goal <gl> ^operator <ol>)
   (operator <ol> "name memory ^cue <cl> ^recalled <x4>)
   (class <cl> ^name Ib-o2)
   (param <x4> ^value <xl>)
   (integer <xl> ^tail <x3>)
   (column <x3> ^anchor tail ^digit <x2>)
   (digit <x2> ^name 5)
-->
   (operator <ol> ^recalled <x4> - <x4> 8))
```

Figure 9: Example Chunk to Reject a (Previously Learned) Bound

Besides operator-implentation chunks, search-control productions are also acquired by Interval-Soar to resolve the tie impasses encountered between *opl* and *opl* in the Interval space. An example of such a chunk is presented in Figure 10.

```
;; Prefer opl over op2 for the data point x • 2.
(sp pi051 elaborate
   (goal <gl> ^operator <o2> + { <> <o2> <ol> } +)
   (operator <o2> ^naxne opl ^param <xl>)
   (operator <ol> ^name op2)
   (param <xl> ^value <il>)
   (integer <il> ^sign positive ^tail <d> ^head <cl>)
   (column <cl> ^anchor head ^digit <dl>)
   (digit <dl> ^name 2)
-->
   (goal <gl> ^operator <o2> > <ol>))
```

Figure 10: Example Search-Control Chunk

4.7 Potential Application to Chemical Process Design

The functionality of a system such as Interval-Soar could be employed in two areas to improve the performance of a design system such as CPD-Soar. One area, which was first alluded to in Section 3.3, is the learning of chunks whose condition elements refer to ranges of numbers rather than specific values. Such chunks could then be used in situations that are different from those under which they were created. This would be equivalent to learning an "approximation" of the model used to evaluate the partial or total design solutions generated.

A second area where a design system could employ the functionality of a system like Interval-Soar is in learning to discriminate among the use of competing models. The example used to illustrate the behaviour of Interval-Soar employs evaluation functions that are the same as the operators they are evaluating. This simple example was used since the intention was to convey an unambiguous description of the functioning of Interval-Soar. However, nothing precludes the system from having a different evaluation function (than the operator itself) or even a set of evaluation functions from which to select Furthermore, there is also no restriction on the use of an external device to compute the evaluation. Since the functions employed in the example are really simple, these can be computed by Interval-Soar itself using the knowledge it possesses about basic arithmetic.

In fact, the existence of a suite of models for evaluating designs, some of which could be computationally very expensive, accurately depicts the state of affairs in chemical process design. For example, in the area of distillation sequence design, models can range from simple list-splitters to those based on rigorous stage-by-stage calculations. Multiple models exist not only for the unit operations and equipment, but also for the materials themselves. For instance, a vapour could be analyzed using the ideal gas law or it could be subjected to extremely detailed molecular theory. Of course these are extremes, but the point trying to be made is that it is usually the case that choices exist. Thus, a critical design decision that often has to be made is what model to use. Factors such as resources available and quality of solution desired are usually taken into consideration when making such a decision. The use of a more detailed model can result in a better quality solution, but at a dearer price.

Useful knowledge to acquire is that which would allow a design agent to decide what situations warrant the use of what models. The ability to make such discriminations could enhance the performance of a design agent considerably. Valuable resources could be saved by selecting and employing simpler models in situations where the use of more complex models would have yielded the same decisions.

It is possible to conceive that a design system such as CPD-Soar could learn to discriminate among the use of multiple competing models if it were endowed with a functionality that was similar to the one possessed by Interval-Soar in learning the intersection point of two functions.

5 Conclusions

We have presented two Soar systems: CPD-Soar, for the design of distillation sequences and Interval-Soar, for learning the intersection point of two functions. With CPD-Soar, we have successfully demonstrated how a process design problem can be carried out within the Soar framework. This is the first application of a cognitive architecture to chemical process design problems.

With Interval-Soar, we have shown how the learning of declarative knowledge could be

used to discretize a space of real numbers into two intervals. This learning is performed as a beneficial by-product of performing another task, that of computing the function values of a set of data points.

We have also described how the functionality of a system such as Interval-Soar could be used by a system such as CPD-Soar either to acquire an approximate model of the domain or to discriminate among competing models. Such abilities, it is hypothesized, could significantly enhance the performance of the design system over its current levels.

6 Acknowledgements

We would like to thank David Steier for his extremely useful guidance in conducting this resea and Allen Newell for his occasional insightful comments. We would also like to thank Paul ienbloom for the use of the *memory* operator and, along with Soowon Lee, for the use of the arithmetic and function spaces. This work was supported by the Engineering Design Research Center at Carnegie Mellon University through a grant from the National Science Foundation.

References

[Douglas 88] Douglas, J. M.
Conceptual Design of Chemical Processes.
McGraw-Hill, San Francisco, CA, 1988.

[Laird 88] Laird, J.E.
Recovery from Incorrect Knowledge in Soar.
In *Proceedings of the National Conference on Artificial Intelligence*, pages 618-623. August, 1988.

[Laird *et at* 87] Laird, J. E., A. Newell & P. S. Rosenbloom.
SOAR: An Architecture for General Intelligence.
Artificial Intelligence 33(l):l-64,1987.

[Newell 89] Newell, A.
Unified Theories of Cognition.
Harvard University Press, Cambridge, MA, 1989.
In press.

[Rosenbloom 89] Rosenbloom, P. S.
A Memory Operator for Soar.
1989.
Unpublished code and working notes.

[Rosenbloom & Lee 89]
Rosenbloom, P. S. & S. Lee.
Soar Arithmetic and Functional Capability.
1989.
Unpublished code and working notes.

[Rosenbloom *et al* 85]
Rosenbloom, P. S., J. E. Laird, J. McDermott, A. Newell & E. Orciuch.
Rl-Soar: An Experiment in Knowledge-Intensive Programming in a Problem-Solving Architecture.
IEEE Trans. Patt. Anal. 75(5):561-569,1985.

[Rosenbloom *et al* 89]
Rosenbloom, P. S., A. Newell & J. E. Laird
Towards the Knowledge Level in Soar The Role of the Architecture in the Use of Knowledge.
In K. VanLehn (editor), *Architectures for Intelligence.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.
In press.

[Talukdar *et al* 88]
Talukdar, S., J. Rehg & A. Elfes.
Descriptive Models for Design Projects.
1988.
Unpublished paper.

[Westerberg 88] Westerberg, A. W.
Synthesis in Engineering Design.
In *Proceedings of Chemdata88.* Gothenburg, Sweden, June, 1988.