Supporting Chemical Process Design within an Integrated Architecture

Ajay Modi, Allen Newell, David Steier, Arthur Westerberg

EDRC 06-119-92

Supporting Chemical Process Design within an Integrated Architecture

AjaylLModi
Department of Chemical Engineering and Engineering Design Research Center

Allen Newell
School of Computer Science

David M. Steier
Engineering Design Research Center

Arthur W. Westerberg[1]
Department of Chemical Engineering and Engineering Design Research Center

Carnegie Mellon University, Pittsburgh, PA 15213

Abstract Process design is a complex activity that requires the design agent to possess characteristics currently missing in most artificial design systems. Soar is an integrated software architecture for knowledge-based problem solving, learning and interaction with external environments. We report on *tfto* systems developed using the architecture. The first, CPD-Soar, designs distillation sequences, while the second, Interval-Soar, performs simple arithmetic tasks. The structure and behaviour of both systems is described and discussed in depth. In so doing, it is depicted how design and design-related tasks can be cast within the Soar framework, hence demonstrating the functioning and strengths of the architecture's problem-solving mechanisms. The systems provide evidence for an hypothesis about learning; namely, that the knowledge learned by an agent while performing a task is strongly dependent upon the models the agent brings to the problem-solving experience. Specifically, it is shown that the richer the model an agent has of its evaluation ftinctions, the more general the knowledge it learns. In relation to this, we describe how the functionality of Interval-Soar can be used to farther improve upon the performance of CPD-Soar. We also discuss a number of other issues that arose when building the two systems.

1. Introduction

Given the significant role that design plays within the chemical processing industry, the development of computer-based support systems has long been an important objective within chemical engineering. However, most process design systems developed to date lack mechanisms allowing them to handle the inherent complexities of their domain. In particular, the ability to make decisions along the entire spectrum of contexts within which design problems are posed, the ability to bring multiple knowledge sources to bear in making these decisions and the

[1]To whom correspondence concerning this paper should be addressed.

ability to learn new knowledge, either from internal problem-solving experiences or from external sources, are seen as being especially significant, even necessary, for dealing with the extensive demands of process design domains.

We begin by introducing the Soar architecture as a vehicle for constructing process design systems with the above abilities, and then describe two systems, CPD-Soar and Interval-Soar, built using the architecture. While die former designs distillation sequences, die latter performs simple arithmetic tasks. The motivation for constructing each system as well as its domain, structure, operation and performance is presented in detail. The implications for process design of each system are also discussed A number of effective ways of further improving the abilities and performance of CPD-Soar are described These include endowing the system with knowledge to evaluate competing separation tasks using marginal prices as well as embedding within the system the functionality of Interval-Soar. The problem-space structure and expected performance of CPD2-Soar, the enhanced version of CPD-Soar, are postulated

2. Soar

Soar [Laird *et al* 87] is an integrated software architecture for knowledge-based problem solving, learning and interaction with external environments under development for several years. The architecture is of direct interest to a large group of researchers for a broad spectrum of reasons. Their research agendas are diverse and their academic backgrounds range from psychology and sociology to engineering and computer science.

The Soar system has been used to build systems capable of solving problems ranging from highly routine to extremely difficult, and its learning mechanism has been successfully applied in a wide variety of situations [Steier *et al* 87]. Depicted as a block diagram IA Figure 2-1, the architecture comprises a small number of distinct mechanisms that provide support for a large number of the capabilities deemed important for a problem-solving agent These mechanisms include problem spaces for conducting all search, production rules for all long-term knowledge, attribute-value objects for all short-term knowledge, preference-based decision making for all decisions, impasse-driven subgoaling for all goal generation, input/output functions for all interactions with the external world and chunking for all learning. The chapter only briefly describes the mechanisms; more detailed descriptions are provided by Laird *et al* [Laird *et al* 90].

CPD-Soar was developed using Soar 4, an earlier version of the architecture, while the current version, Soar 5, was used in creating Interval-Soar. The description of the architecture presented is of Soar 5 with some of the more important differences between Soar 4 and Soar 5 highlighted.

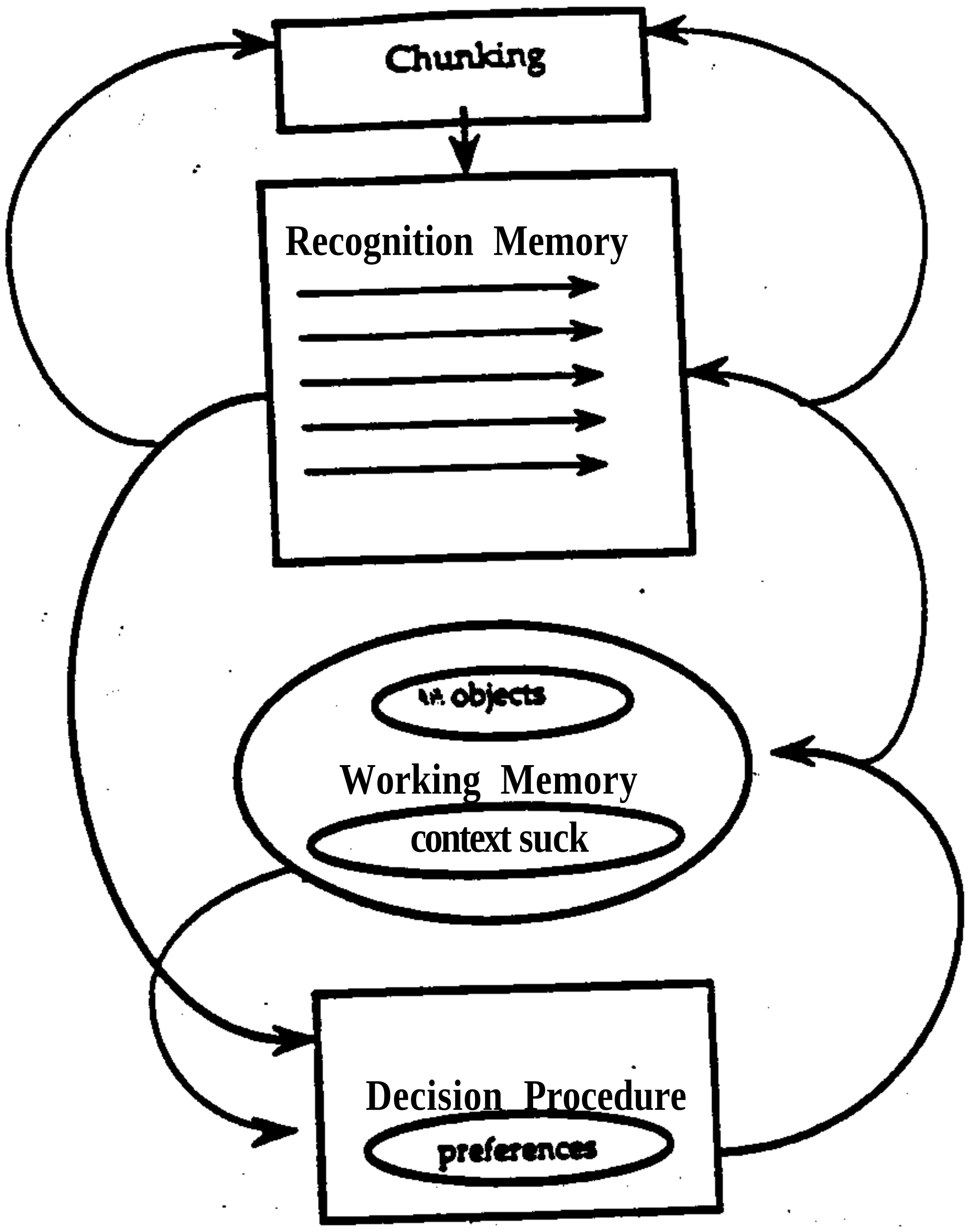


Figure 2-1: Architectural structure of Soar

2.1. Problem Spaces

* All tasks in Soar **are** performed as seaich 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. Figure 2-2 depicts an example problem space from the Blocks World task.

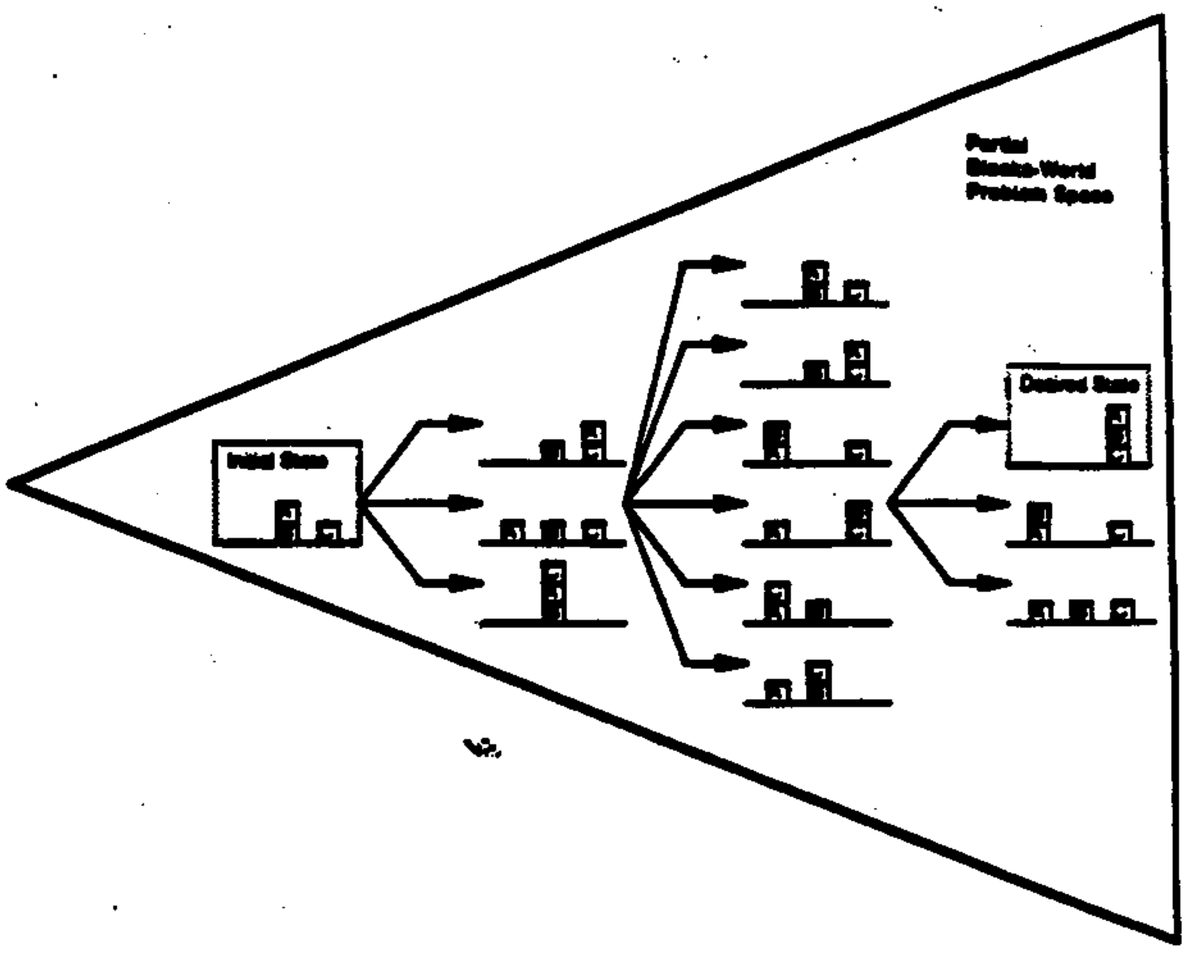


Figure 2-2: Example problem space
(from [Laird *et al* 90], Figure 2-1)

In Soar, multiple goals correspond to a task decomposition, each of which may require different problem spaces to be searched. The task is accompUshed when the top-level goal is attained. 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.

A major difference between Soar 4 and Soar 5 is in the way changes are made to an existing state. In Soar 4, the application of an operator results in the creation of a new state. All state modifications are made directly on the new state and any state contents not changed as a result of the operator application are copied from the old state. Multiple states may thus be maintained within a single problem space. In contrast, an operator is implemented in Soar 5 by making destructive changes to the existing state; thus, only a single state is maintained in each problem space.

2.2. Working Memory

All short-term or temporary knowledge, representing the current problem-solving situation, is stored in working memory as a set of objects. Each object, denoted by a unique symbol called its identifier, consists of a set of augmentations and a set of preferences for the augmentations. While augmentations describe the object in terms of a set of attributes, preferences determine the actual contents of working memory by asserting the relative or absolute worth of a specific value of an attribute. Since the values of attributes can be identifiers of other objects, complex frame-like representation structures may be created. Figure 2-3 depicts the general form of an augmentation, which consists of four symbols: a *class,* an *identifier,* an *attribute* (preceded by an ^{A}) and a *value. Of* the example augmentations, two share the same identifier (c4) and hence refer to the same object, and two belong to the same class (distillation-column).

```
GENERAL FORM OF AN AUGMENTATION:

         (class id "attribute value)


EXAMPLES OF AUGMENTATIONS:

         (component c4 Aname benzene)
         (component c4 Aflowrate 23.8)
         (distillation-column d8 Are£lux-ratio 2.85)
         (distillation-column d2 Areboiler r3)
         (reboiler r3 Atemperature 473)
```

Figure 2-3: Example Soar working-memory augmentations

2.3. Recognition or Long-Term Memory

All long-term knowledge in Soar is stored in an associative recognition memory, realised as a production system. Each production consists of a set of conditions and a set of actions. The conditions test working memory for the presence or absence of object augmentations whereas the actions add to it new augmentations and preferences. Productions encode all knowledge required to perform a task. This knowledge pertains 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.

Figure 2-4 presents an example Soar production, its English version, the working memory augmentations that instantiate the production, i.e., cause it to fire, and the augmentations and preferences added to working memory as a result of the production firing. The pluses in the working-memory elements denote acceptable preferences.

In Soar 4, the preference scheme is only used in determining the contents of context slots, i.e., problem spaces, states and operators. In Soar 5, preferences are used to determine the values of all augmentations, not just those pertaining to the context

2.4. Decision Cycles

All problem solving in Soar revolves aroung a number of decisions; what problem space should be searched to attain a 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 consisting 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 can continue for many elaboration cycles. It terminates when it runs to quiescence, i.e., when no more productions can fire. Elaboration results in both augmentations and preferences being added to working memory.

The selection of an object for a role is made during the second phase of the decision cycle, the decision procedure phase. Beginning 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 chosen. The selection of an object for a context slot signifies the end of a decision cycle and problem solving then proceeds with the elaboration phase of the next cycle. Figure 2-5 pictorially illustrates an example sequence of three decision cycles.

SOAR PRODUCTION:

```
 (sp interval*inst-select-x
     (goal <g> ^probiem-space <p> ^state <s>)
     (problem-space <p> ^name interval)
     (state <s> ^parameter <x>)
     (parameter <x> ^selected no ^name x)
-->
     (goal <g> ^operator <o> +)
     (operator <o> ^name select-x + ^parameter <x> +))
```

ENGLISH VERSION OF PRODUCTION:

```
  If  problem*space interval has been selected
 and  parameter x has not been selected

then  create an acceptable preference for operator select-x
 and  create an acceptable preference to augment operator
      select-x with parameter*x
```

AUGMENTATIONS INSTANTIATING PRODUCTION:

```
     (goal g2 ^probiem-space p3 ^state s2)
     (problem-space p3 ^name interval)
     (state s2 ^parameter x5)
     (parameter x5 ^selected no ^name x)
```

AUGMENTATIONS AND PREFERENCES ADDED BY PRODUCTION FIRING:

```
     (goal g2 ^operator o2 •)
     (operator o2 ^name select-x + ^parameter x5 +)
```

Figure 2-4: Example Soar production and working-memory contents before and after firing

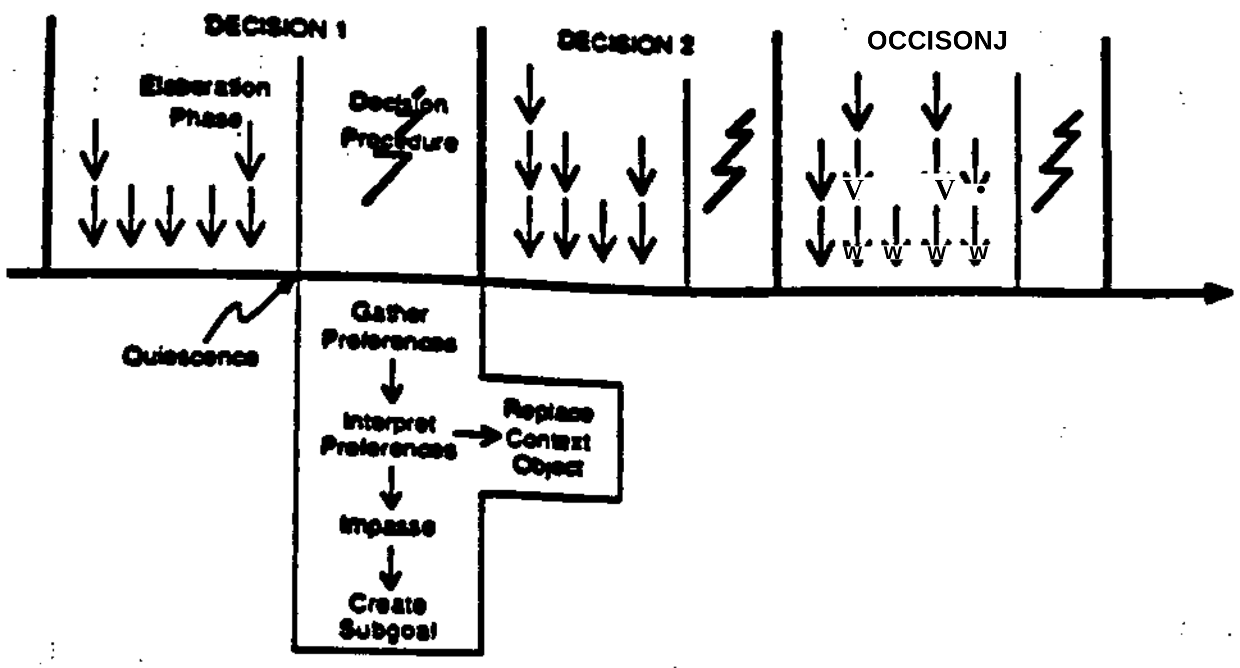


Figure 2-5: Example sequence of decision cycles
(from [Laird *et al* 87], Figure 2-6)

2.5- Subgoaling

A situation may arise in the problem solving when a unique decision cannot be made for any of the slots in the current context because of either incomplete or inconsistent information. Such a situation, known as an impasse, is dealt with by Soar by automatically subgoaling. Furthermore, subgoals may occur within subgoals, thus resulting in a hierarchy. The architecture recognises four kinds of impasses: tie, no-change, rejection and conflict A tie impasse occurs when several objects, for example operators, have been made acceptable, but not enough knowledge exists to select one. A no-change impasse arises if none of the context slots change value during a decision cycle. A rejection impasse occun if all the objects made acceptable for a context slot are also rejected A conflict impasse arises if the objects for a context slot have conflicting preferences, e.g., X is better than Y and Y is better than X. Impasses are resolved when preferences that allow Soar to select one of the candidate objects uniquely for a context slot are added to working memory.

Figure 2-6 depicts the preferences in working memory before and after an operator-tie impasse * has occurred

```
BEFORE SUBGOALING:

        (operator 01 acceptable)
        (operator 02 acceptable)
        (operator 03 acceptable)
        (operator 03 indifferent to operator 02)

        ~> Generate subgoal to resolve tie ixnpasae among
           operators 01, 02 and 03

AFTER SUBGOALING:

        (operator 01 better than operator 02)
        (operator 03 rejected)

        —> Operator Ol selected and impasse resolved
```

Figure 2-6: Example preferences in Soar before and after an operator-tie impasse

2.6. External Interaction

All interactions with the external environment occur via a single input/output (I/O) interface. The interface allows Soar systems to communicate with two kinds of functions, input functions and output functions, both written in Lisp. Input functions provide Soar with information about the outside world by creating preferences that result in changes to top-level-state augmentations. Output functions affect the external enviroment by responding to changes in top-level-state augmentations caused by production firings. An output function is triggered whenever a working-memory element that pattern matches the output function is created

The I/O interface is a feature of Soar 5. In Soar 4, all interactions with the external world are performed via lisp functions that are called as actions from the right-hand-sides of productions.

2.7. 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. Chunking operates by basing the actions of a new production on the results of the subgoal and the conditions on those aspects of the pre-impasse situation that were relevant to the generation of the results. Relevancy is determined by working backwards through the traces of the productions that fired in the subgoal to those working-memory elements, residing outside the subgoal, on which the subgoal's results are ultimately dependent These elements form the basis of the chunk's conditions. An important feature of the backtracing procedure is that productions that only generate search-control knowledge do not have their traces examined since these productions only affect the efficiency with which a goal is attained, and not the correctness of its results. Finally, the working-memory elements deemed relevant by the backtracing procedure are processed to determine the chunk's actual conditions and actions. This includes replacing some of the symbols in the working-memory elements by variables, a process known as variablization. Only symbols that are object identifiers are replaced by variables. All others are left as constants. This generalisation process ensures that the chunks learned will apply in future problem-solving situations that are not exactly the same as the ones they were created under, only similar. Laird *et al* [Laird *et al* 86] have presented a detailed description of chunking in Soar.

Figure 2-7 schematically illustrates the creation of an example chunk. The circles in the diagram represent working-memory elements while the intervals between the dashed lines represent decision cycles. Since the elements A, B and C represent the pre-impasse situation and the elements D and E are the results that resolve the impasse, the former constitute the conditions of the learned chunk and the latter its actions.

Like any other learning system, it is possibltffor Soar to acquire incorrect knowledge, which may result if the system either makes the wrong inference or receives the wrong information. Although productions encode all knowledge in Soar, errors only arise through incorrect decisions, Le., the wrong problem space, state and operator being selected. Consequently, recovering from erroneous knowledge in Soar does not involve modifying the productions that encode the knowledge, but rather the decisions that are the consequence of applying the knowledge. By learning productions that make the right decisions, Soar recovers from incorrect knowledge. Laird [Laird 88] has described a recovery-from-error technique that allows Soar systems to recover from any incorrect knowledge they may have captured in their long-term memories.

Rosenbloom and Laird [Rosenblootn & Laird 86] have described how chunking maps onto explanation-based learning [Mitchell *et al* 86], an analytic learning method.

3. CPD-Soar

3.1. Overview

CPD-Soar is a system developed within the Soar architecture that solves process design tasks. The system determines the sequence in which the splits should be applied to a feed stream in order to create the desired products. The reasons for developing CPD-Soar were simple: to understand how a process design task could be cast within the Soar framework and to observe

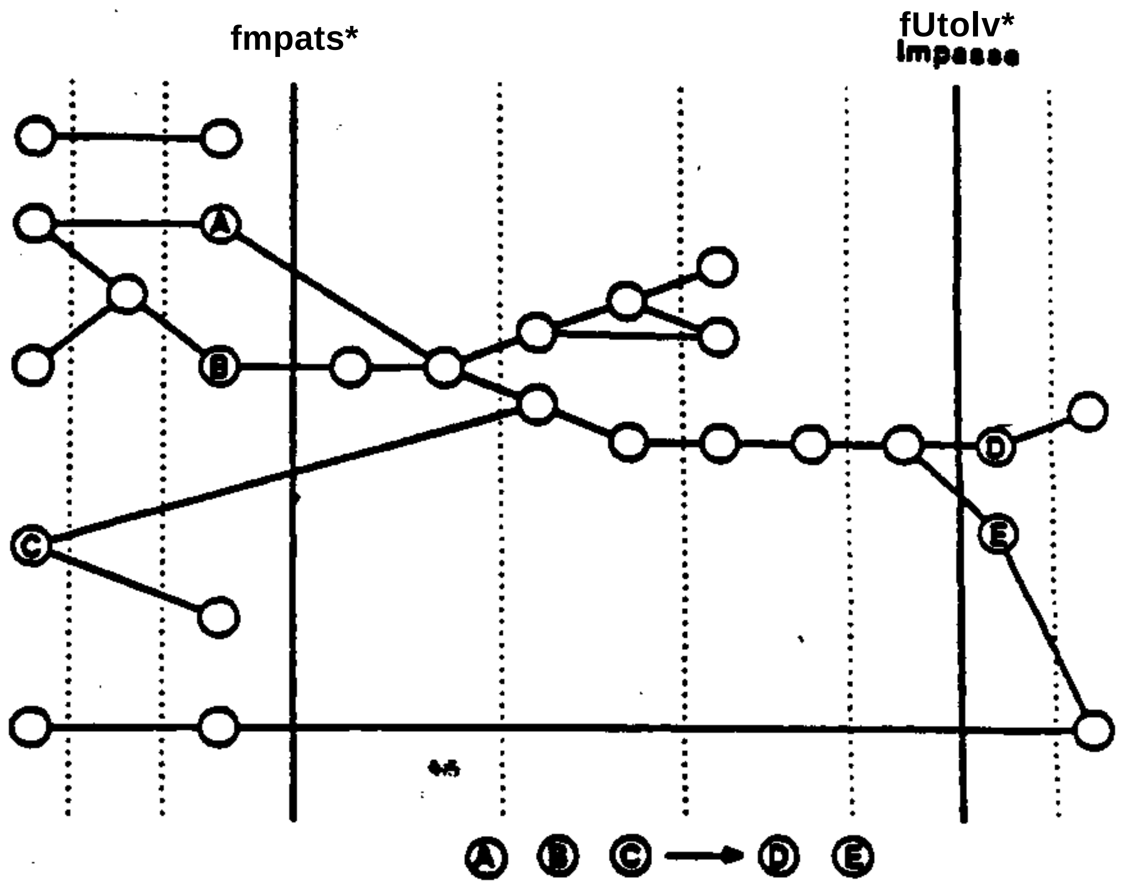


Figure 2-7: Chunk creation in Soar
(from [Newell 90], Figure 4-12)

how die resulting system would perform, in terms of both problem solving and learning.

Given a feed specification, CPD-Soar solves the task by first generating all the applicable splits and then sequencing them. It controls the search by applying a number of heuristics commonly used in distillation sequence design. It resolves ties and conflicts among the heuristics by performing a one-step lookahead search. It has been shown in Chapter CHAP3 that the marginal price of a separations task performs better as an evaluation function for controlling the search than the heuristics employed in CPD-Soar. However, because the function was not discovered until after CPD-Soar was created, it was not used in evaluating competing*task decisions in CPD-Soar.

3.2. Structure of CPD-Soar

CPD-Soar's knowledge is distributed among its domain spaces and the selection^2 space. Figure 3-1 depicts the decomposition of the system into its problem spaces and operators. Each problem space is depicted by an oval and the operators it contains are listed in the box next to it Impasses are denoted by directed lines linking problem spaces. ON refers to an Operator No-change impasse.

CPD-Soar is implemented by 168 (Soar 4) productions. Of these, 60 implement the selection space and other default search-control knowledge. The sizes of the remaining task spaces in CPD-Soar in terms of numbers of productions is summarised in Table 3-1.

3.2.1. Domain Spaces

There are seven domain spaces in CPD-Soar design, feed, order, split, sequence, update and output. Of these, all except the design space are operatoT-implementation spaces. The design space, which is the top-level space, has eight operators: *get-feed, order-components, link-components, make-splits, identify-forbidden, sequence-split, update-stream* and *write.*

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, *moke-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 its flowrate, its relative volatility and product in which it is desired.

The *order-components* operator 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 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 sputters. 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 whose components have already been 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 2V-1. Each split is characterised by a light and heavy key. The split problem space implements the *make-sphts* operator. Split contains a single operator, also called sptt, that generates a split and computes the ratio of the volatilities of the light and heavy keys. *Identify-forbidden* tags each split

•Boldface TlmesRoman is used to denoce problem spaces and *M&ee Unties* is used to denote operators.

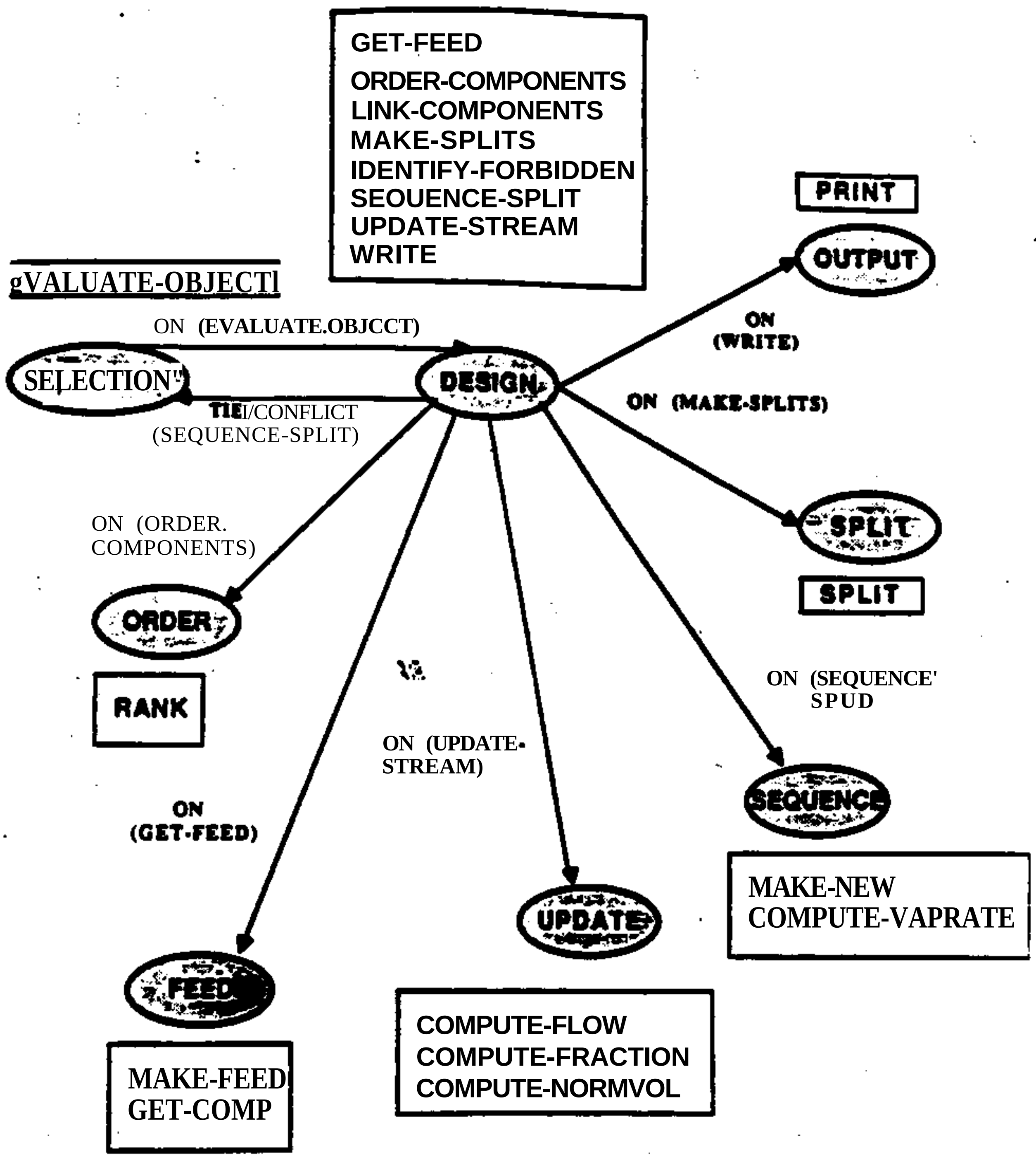


Figure 3-1: Problem-space structure of CPD-Soar

Problem Space	No. Productions
Design	49
Feed	9
Order	**8**
Update	14
Split	6
Sequence	15
Output	7

Table 3-1: Sizes of domain spaces in CFD-Soar

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*****comjtihe-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. *Write* is implemented as the problem-space **output,** which contains a single operator called *print* that outputs a split to the screen.

The operators described thus far are all used by GPD-Soar to perform routine book-keeping and input/output functions. However, the key function to be performed in solving the design problem is the ranking of the allowable splits in the order in which they are to be applied. This function is performed by the *sequence-split* operator, which is implemented as a problem space called **sequence** that contains two operators, ***make-new*** and *compute-vaprate.* The *make-new* operator 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. The *compute-vaprate* operator computes the vapour flowrate in the column using a simplified function proposed by Douglas [Douglas 88].

The *sequence-split* operator is made acceptable for all unranked allowable splits that may be applied to streams that have neither been split nor are 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. If the application of the heuristics fails to result in a unique split choice, a one-step lookahcad strategy is employed to make a decision. This involves selecting the split that results in the column with the lowest vapour flowrate.

3.2.2. Selection Space

The selection space contains Soar's default knowledge for resolving multi-choice impasses, i.e., ties or conflicts among competing problem spaces, states or operators. It contains one operator, *evaluate-object*$_y$ whose function is to compute an evaluation for a competing object. The following section describes the use of the selection space to resolve ties or conflicts among competing *sequence-split* operators.

3.3. Illustration of CPD-Soar Executing

To portray the search performed by CPD-Soar, consider the simple but representative situation in which a stream consisting of three components, A, B and C, is to be separated This situation is depicted in Figure 3-2.

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 indecision 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 object in the tie or conflict impasse, 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 flowrate with respect to the other. Supposing that X is smaller than Y in this case, the split B/C is chosen.

To illustrate die behaviour of CPD-Soar, die problem-solving trace for an example eight-component problem is presented below. This version of CPD-Soar is only endowed with one search-control heuristic, select easiest separation next; hence, the splits are applied in order of decreasing ratio of key-component relative volatilities. Splits with the equal key-component

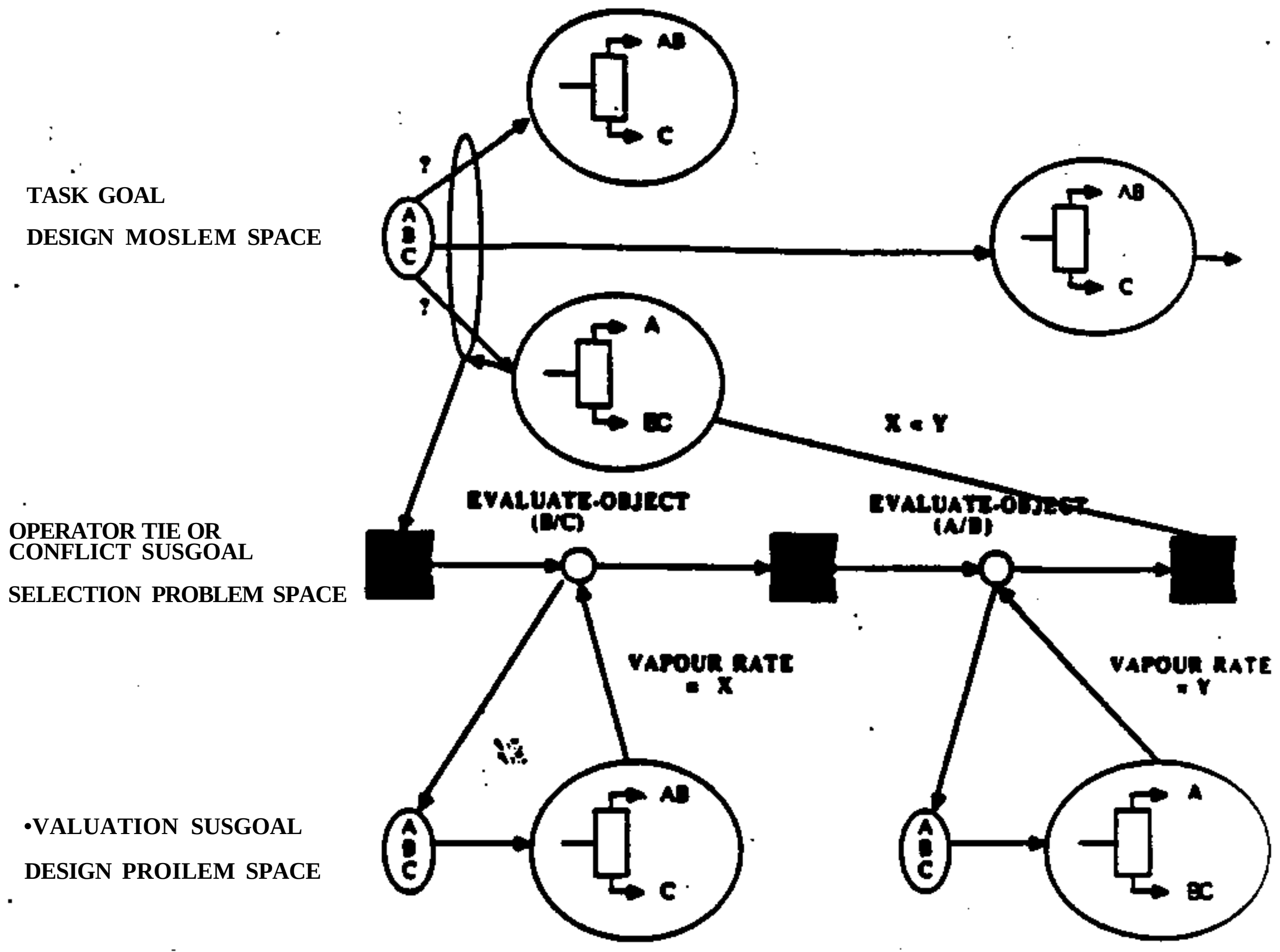


Figure 3-2: Selecting among competing splits in CPD-Soar

relative-volatility ratios are made indifferent to each other.

The leftmost column indicates die decision cycle number. G.(for goal), P (for problem space), S (for state) and O (for operator) indicate the context slots and the identifiers adjacent to them, e.g., Gl or P2, indicate the specific objects that fill the slots together with their names in some cases, e.g., design or *get-feed.* Indentations in the trace indicate the creation of subgoal£ All statements in brackets are comments included to make the trace readable while all other statements are either output by the system or input by the user.

0 6: 61

```
1   P: P2 DESIGN
2   S: S4
3   O: 05 GET-FEED
4   ->G: Q€  (GET-FEED OPERATOR NO-CHANGE)
5      P: P7 FEED
6      S: S9

7      O: Oil MAKE-FEED Enter name of feed stream

feed       [The feed stream name has been entered]

 Enter number of components
8          [The number of components has been entered]

8      S: N13

9      O: 014 GET-COMP Enter component name

h          [The component name has been entered]

 Enter component volatility

3          [The component volatility has been entered]

 Enter component flowrate

1          [The component flowrate has been entered]
           [Name/ volatility and flowrate are entered for the
            remaining 7 components___]

10  S: N25
11  0: 026 ORDER-COMPONENTS
12  «-X3: G27  (ORDER-COMPONENTS OPERATOR NO-CHANGE)
13     P: P28 ORDER
14     S: S30

           [The components of the feed stream are ordered...]

31  S: N83
32  0: 084 LINK-COMPONENTS   [The components are linked]
33  S: N85
34  0: 086 DPDATE-STREAM
35  —X3: G87  (UPDATE-STREAM OPERATOR NO-CHANGE)
           [The stream is updated]
36     P: P88 UPDATE
37     S: S90
38     0: 091 COMPUTE-FLOW
39     S: N93
40     O: 094 COMPUTE-NORMVOL
```

```
41     S: N96
42     O: 097 COMPUTE-FRACTION
43  S: N99
44  O: O100 MAXE-SPLITS
45  —>G: 6101  (MAXE-SPLITS OPERATOR NO-CHANGE)
          [All possible splits are created]
46     P: P102 SPLIT
47     S: S104
48     O: O106 SPLIT
49  S: N129
50  O: 0131 SEQUENCE-SPLIT
          [A split is selected to apply]
51  —>G: 6137  (SEQUENCE-SPLIT OPERATOR NO-CHANGE)
52     P: P138 SEQUENCE
53     S: S140
54     O: 0141 MAKE-STREAM   [New streams are created]
55  S: N154
56  O: 0156 ORDER-COMPONENTS [The components are ordered...]
       [Similar steps to those described above are performed
        whenever a new stream is created; its components are
        ranked and linked and it is updated...]

233 S: N509
234 O: 0510 WRITE           %*
          [Since all splits are ranked, they are output]
235 —>G: 6511  (WRITE OPERATOR NO-CHANGE)
236    P: P512 OUTPUT
237    S: S514
238    0: 0515 PRINT
Split B/C of key-component volatility ratio 3   is 1
Split C/D of key-component volatility ratio 5/2 is 2
Split E/r of key-component volatility ratio 2 . is 3
Split r/6 of key-component volatility ratio 2   is 4
Split 6/H of key-component volatility ratio 2   is 5
Split A/B of key-component volatility ratio 2   is 6
Split D/E of key-component volatility ratio 3/2 is 7
goal D0-DESI6N achieved      [The problem has been solved]
"End — Explicit Halt"
```

3.4. Performance of CPD-Soar

As noted earlier, almost all of CPD-Soar's problem solving involves either routine calculations or the transfer of information into and out of the system. The only problematic situations encountered concern the ordering of the splits. Hence, the performance of the system is most usefully described with respect to its strategy in resolving ties or conflicts among competing splits. In regard to this, two points will be made: the first concerns the use of the single-step lookahead search and the second the use of vapour flowrate as an evaluation metric.

Recall that an important incentive in constructing GPD-Soar was to get an appreciation of the kind of chunks **that** would be learned in domains in which a significant portion of the problem solving involves numerical calculations. To achieve this objective, the lookahead search was deemed sufficient since chunks are learned by CPD-Soar whenever results are generated within subgoals.

Vapour flowrate was selected as an evaluation metric since in many contexts it is a good indicator of the cost, both construction and operating, of a column. Lower vapour rates result in smaller columns and lower utility usage. However, this in no way implies that CPD-Soar is restricted to using vapour flowrate as the evaluation metric. By endowing it with the appropriate knowledge, the system can certainly be made to apply alternative evaluation metrics.

To depict its learning performance, a typical chunk learned by CPD-Soar is presented in Figure 3-3[3] together with its English version. Although too specific, i.e., die condition elements have attributes whose values are numeric constants, such a chunk can be useful if the system solves its task by conducting an exhaustive search and the search space is represented as a network, i.e., the quantity of each species in the input to a distillation column is assumed to be either the same as in the initial process feed or zero. Under these conditions, a chunk acquired early in the problem solving can fire during later stages of the same problem.

In process design problems, it is often the case that many computations are repeated several times 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 is the calculation of the column parameters for a particular split A search down one branch of the graph 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 will have chunked away the results of the A/B evaluation that was performed 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 will have to be remade several times within a single problem instance. For example, while searching a particular branch of a graph, **a** decision may have to be made between two competing splits, say, A/B and B/C, which in turn may require a search. Later, when some other branch of the graph is being searched, it is possible that a decision may again be required between A/B and B/C. This time however, the system will be able to make a decision immediately based on the knowledge it had learned earlier. Such within-trial transfers of knowledge can help make tractable larger design problems than would be attempted without it

[3]Thc syntax of the production presented here is Soar 4 since this was the version of the architecture within which CPD-Soar was initially developed while the productions presented elsewhere are in Soar 5 syntax.

```
(•p p45 alaborata.
    (goal <gl> "problan-apaca ( o undacidad <pl> }
     Aatata < O undacidad <nl> } Adaairad <dl>)
    (dasirad <dl> Abattar lowar)
    (oparator <o2> Anana aaquanca-aplit A»plit <»1>
     Aatraam <«3>)
    (aplit <al> Ahk <hl> Alk <11> Aralvol 3/2)
    (Ik <11> Anana b)
    (atraan <a3> Acomponant <d> { o <cl> <c2> }
     ( o <c2> o <cl> <c3> ))
    (componant <cl> Anama b Aflowrata 2)
    (hk <hl> Anama c)
    (componant <c2> Anama c Aflowrata 3)
    (oparator { o <o2> <ol> } Anama aaquanca-aplit)
    (oparator <ol> Aaplit < o <al> <«2> } Aatraam <«3>)
    (aplit <«2> Ahk { o <hl> <h2> > Alk { o <11> <12> }
     Aralvol 2)
    (Ik <12> Anama a)
    (componant <c3> Anama a Aflowrata 1)
    (hk <h2> Anama b)
-->
    (prafaranca <o2> Arola oparator Avalua worsa
     Arafaranca <ol> Agoal <gl>  ^roblam-apaca <pl>
     Astata <n1>))
```

```
  If  aaquanca-aplit oparator o2 that inplamanta
      aplit al haa baan aada accaptabla
 and  aplit al haa light kay B and haavy kay C
 and  tha ratio of volatilitiaa batwaan B and C i« 1.5
 and  C haa flowrata 3
 and  B haa flowrata 2
 and  aaquanca-aplit oparator ol that implamanta
      aplit »2 haa baan mada accaptabla
 and  aplit «2 haa light kay A and haavy kay B
 and  tha ratio of volatilitiaa batwaan A and Bit 2
 and A haa f lowrata 1

than  craata a woraa prafaranca for oparator o2 with
      raapact to oparator ol
```

Figure 3-3: Example of a chunk leamed by CPD-Soar

3.5. Implications for Process Design

The development of GPD-Soar was intended to be an initial step towards understanding the implications of developing a process designer within an integrated architecture. Its instantiation was a valuable exercise for two main reasons: one, it presented evidence that the mechanisms present in Soar can provide process design systems with useful abilities and two, the act of creating it was helpful in distinguishing those aspects of the task domain that are well understood from those that are not

3.5.1. Usefulness of Soar's Mechanisms for Process Design Systems

As described in Chapter CHAP2, if process design systems are to be capable of tackling the complex and diverse demands of their domains, they require a wide range of abilities. Some of these capacities have been displayed by a number of design systems, mostly small shallow expert systems performing in limited domains. These include the ability to represent and manipulate symbolic situations, the ability to formulate tasks as search, the ability to decompose larger problems into groups of smaller subproblems and to organise these as subgoal hierarchies, and the ability (made possible by archiving the system's knowledge as a parallel if-then recognition memory) to bring all relevant knowledge to bear at each point of the problem-solving process. Other capacities, such as the ability to mix knowledge and search, operating as recognition-like systems in regions where the expertise exists, and searching where it is lacking, and die ability to learn, have not The development of CPD-Soar was important since it provided evidence that process design systems can be built in which not only the missing capabilities **are** exhibited, but that all the above-mentioned capacities can be tightly integrated within a single process designer.

The mixing of knowledge application and search that occurs in CPD-Soar plays an important role in allowing design decisions to be made at run time rather than at system-creation time. To illustrate die significance of run-time decision making, consider the multiplicity of heuristic rules that have been proposed for selecting the split to be applied to a process stream. Since most of these rules discriminate on the basis of different attributes, conflicts and ties among the competing splits are to be expected in many cases even after the rules have been applied. Most previous works in the field deal 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 the rules altogether. Instead, either exhaustive searches or heuristic evaluation functions involving partial searches are employed to rate the competing splits. However, most of these schemes **are** computationally expensive for even moderately sized problems. A third approach uses only a subset of ail the rules 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 rules could have applied

As will have been noted, in each approach decisions about how splits should be evaluated are made *a priori,* i.e., when the system is created As emphasized in Chapter CHAP2, this is tantamount to solving, at least partially, a design problem before it has even been posed. Hence, any design systems created on the basis of the above-mentioned approaches will also suffer any shortcomings resulting from *a priori* made decisions. In this regard, CPD-Soar's ability to deal with inconsistent and incomplete knowledge by subgoaling allows it to overcome the weaknesses of the other approaches. The power of using heuristic rules as a means of controlling the search is exploited, and only when the rules result in conflicts or ties, is the more expensive lookahead search resorted to.

All this is not to say that CPD-Soar, in the version described, has no decisions hardwired into it The system's solution strategy (heuristic search), its split evaluation metric (vapour flowrate) and its model for computing vapour flowrate, among other decisions, were all made when the system was created. However, as noted earlier, these restrictions are not a consequence of using Soar. It is certainly possible to pose the selection of solution strategies and evaluation metrics and models, for example, as decisions within CPD-Soar. The only limit to doing this is the current lack of understanding in making these decisions. Although the relevant design literature is flush with descriptions of different solution methods and models and evaluation functions, all performing well within their (usually unstated) contexts, there is an absence of any discussion of the conditions which govern their performance and hence their selection.

3.5.2. Utility of Creating CPD-Sdar in Understanding the Task

The act of creating CPD-Soar was itself a valuable exercise because it helped to better understand the task domain. In this regard, the development activity was useful in two significant ways. First, the task analysis performed to help articulate the domain knowledge was instrumental in the discovery of the marginal price concept for the separation system design problem. The usefulness of the concept has been described in Chapter CHAP3.

Second, the explicit mapping of task knowledge to problem spaces required to instantiate the system helped identify gaps in the current state of understanding of the task. The dearth of knowledge to select among competing models, evaluation functions and methods alluded to earlier is only one such hole. Two other important areas where it was discovered that a lack of understanding existed about the required knowledge, how the knowledge was to be integrated with the rest of the system's knowhow, and what was needed in the way of capacities to learn this knowledge, are discussed below. In light of our experiences with constructing CPD-Soar, it is conjectured that building process design systems within an architecture such as Soar that forces the explicit representation of all task-related knowledge is a powerful scheme for identifying where future design research should be concentrated.

Learning in Numerically-Intensive Domains: Although the use of chunking within CPD-Soar demonstrated how a process designer's problem-solving experiences can be captured for future use it also indicated that knowledge learned in operator-implementation and evaluation subgoals in numerically-intensive domains such as engineering design could be too specific. This is problematic since learning will only be truly useful if the captured knowledge can also be used in situations different from the ones it was acquired under. Observing CPD-Soar's learning behaviour was thus valuable since it explicitly pointed to the need for a solution to the too-specific-chunks **problem.**

Even though **some headway was** subsequently made in dealing with the problem of learning knowledge that is **too** specific, very little was known at the outset about either the cause of the problem or its remedy. Was it the particular task representation being used that resulted in the chunks being too specific? If so; what alternative representations would have to be employed? Or was it the inherent nature of chunking? Perhaps the abstraction processes embedded within the learning mechanism were singly inadequate for creating chunks of the right generality. If so, any additional abstractions needed would have to be provided *a priori* to the system or would have to be generated as part of the problem solving. However, the problem spaces within which these abstractions could be created and how these spaces should be integrated with the design-task spaces was largely unknown. Later work leading to the development of Interval-Soar provided some answers to these issues.

Interaction with External Software Systems: A recurring issue during the design of CPD-Soar was the question of how the total task labour was to be distributed Should CPD-Soar perform the task completely using only its own internal abilities? Or should part of the effort be shipped out to external systems? If so, what aspects of the task should these be, and to what external systems should they be sent? The answers to these and other related questions were and still are largely unknown. It was finally decided to have CPD-Soar perform the task internally, without recourse to external tools. The reason for this was simple: to avoid the complexities of Soar-tool interactions until a better understanding wfcs obtained of what was required to get CPD-Soar to use external tools effectively. Besides, such a decision would allow the learning and problem-solving behaviour of a Soar system performing autonomously within the process design domain to be witnessed, which, it should be recalled, was one of the primary aims of creating CPD-Soar in the first place.

Later, it was observed that although CPD-Soar could perform its task, it did so inefficiently, a symptom largely attributable to its non-use of external tools. Such tools will henceforth be referred to as *external software systems (ESSs),* since they are created and exist external to the agent, here CPD-Soar, that ultimately uses them. CPD-Soar's task environment is analogous to that of **a human designer** performing the same task without die benefit of external software systems, **except perhaps a** simple calculator. Such a task environment is not a true reflection of the current **state of affairs. Due** to the pervasive growth of research in process design towards specifying **and writing** software aids and tools, the nature of the process design environment has evolved from **one** inhabited by no software tools to one populated by many systems. Real process design environments are today rife with software tools; equation solvers, optimisation packages, process simulators, databases, etc. abound in number. These software systems have developed to the point where they now perform a significant portion of the complete design task. Hence, if artificial agents are to be created for solving process design problems, it is imperative that they be endowed with tool-using capacities.

However, as mentioned earlier, it is still largely unknown how artificial agents can be made to use ESSs. With the formation of the DESS (Interaction with External Software Systems) subgroup within die Soar community, preliminary steps have already been taken towards achieving this understanding. Newell and Steier [Newell & Steier 91] have described the subgroup's objectives, the empirical observations that lead to its formation and the activitcs that compose its research agenda.

The ESS environment is very relevant to CPD-Soar. Even in its current limited form, a large proportion of its operators, including *order-components, update-stream, compute-flow, compute-fraction, compute-normvol* and *computeraprate,* could be implemented as external knowledge sources. As the complexity and size of CPD-Soar[9]s task grows, one can envisage the system having to interact with larger and more sophisticated ESSs. Databases could be accessed to retrieve relevant component data, process simulators could be invoked to simulate columns and compute relevant parametric quantities such as reflux ratios and vapour flowrates, mathematical programming packages could be executed to optimise process variables and external memories could be used to store detailed design descriptions.

Getting artificial agents to interact with ESSs involves more than just the creation of the appropriate transducers and interfaces to facilitate the conversion and exchange of information from one entity to another. As Newell and Steier have stated, a commonly held belief, termed the *transduction fallacy* by them, is that the only capability required by an agent in working with an ESS is the transduction between the representations of the agent and the software system. For example, in the case of CPD-Soar interacting with an optimisation package, it could appear that all that was required was the conversion of the mathematical programming model to a language suitable for input to the optimisation package. This is clearly a fallacy, because there is certainly a fair amount of problem-solving activity involved in first formulating the task as a mathematical program before transducing it into the syntax required by the solver.

Newell and Steier have listed and discussed the performance capabilities required by an artificial agent in order to use an external software system. These include formulate-subtask, create-input, convert-output, interpret-result, operate-software-system and simulate-ESS. Formulate-subtask is the capability concerned with formulating a subtask as a computational problem and deciding whether to solve the problem using internal resources or to employ an ESS. Create-input and convert-output are the transduction capabilities. Interpret-result involves the use of the results created by the ESS in service of the original task. Operate-software-system deals with the operation and monitoring of the ESS under both normal and abnormal conditions and simulate-ESS involves the ability to create the same results as the ESS at either the same or different levels of abstraction and approximation. The capabilities and their interactions are depicted in Figure 3-4.

The development of CPD-Soar was useful for the role it played in identifying the need for research into understanding what was required to get artificial agents to interact with ESSs. The creation of the system provided evidence, one more data point, of the need for focussed effort in both defining a relevant research agenda and executing it

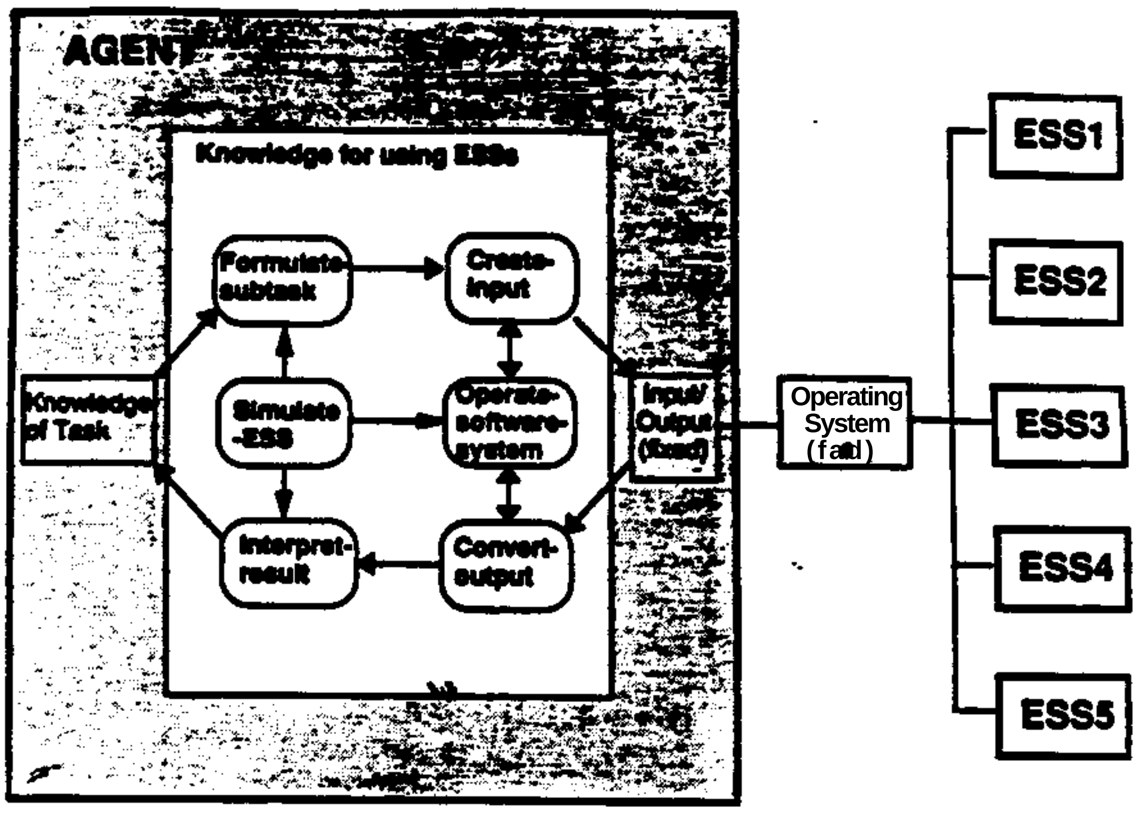


Figure 3-4: Capabilities for using external software systems
(from [Newell & Steier 91], Figure 2)

4. Interval-Soar

4.1. Overview

Interval-Soar is a system developed within die Soar architecture that performs a simple arithmetic task. Although die problem solved by Interval-Soar is elementary, namely, to determine which of two fimctions gives the larger response when applied to a variable, its creation was valuable in the attempt to understand what is required by an artificial agent to learn useful knowledge while performing in a quantitative domain. The development of Interval-Soar was a direct **response** to **the** observation that the chunks learned by CPD-Soax were too specific to be of general **use.**

As was seen earlier, when CPD-Soar selects among competing separation tasks by evaluating them via a lookahead search, all that it teams for a particular input (representing the flowrates and volatilities of the components involved) is that one task should be preferred over another for that input The chunk presented in Figure 3-3 illustrates this learning. The reason for the specificity in the learning is that all the system knows, albeit implicitly, about the evaluation employed is that it produces numeric values which are to be compared to make a unique choice. Hence, all that can, and does, get learned is knowledge of the kind depicted in Figure 3-3. If the system is to be capable of learning more general knowledge, then it must initially have more knowledge, i.e., a better model, of what is to be learned. For example, if the model possessed by the system is that the results of the evaluation also provide information about the region in which the evaluation function is applicable, then the system will learn about this region. Interval-Soar is an attempt to explore this hypothesis.

Two simple evaluation functions, one of which was monotone increasing and the other monotone decreasing, were selected for investigation within Interval-Soar. The system was implicitly provided with an abstract model of these functions, namely, that they intersected at a single point Then, if the hypothesis was valid, through their application, the system would leam information about the evaluation function***that was of more general use than the knowledge learned by CPD-Soar.

4.2. Task Performed by Interval-Soar

The task performed by Interval-Soar consists of selecting one of two functions, fl and f2, to apply to each element of a set of values of a scalar variable x. The function to be chosen for a particular value of x, say $\bar{x}$, is the one which gives the greater result Hence, if $f1(\bar{j}) < f2(J)$, f2 will be selected as the function to be applied. When a result has been computed for a particular x value, it is labelled with the function that was applied to it When all the variable values have been labelled, the task is considered accomplished. As a consequence of performing its task, Interval-Soar learns to converge to the intersection point of the two functions.

Each function is characterized by a parameter called its bound. For function f1, 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 in which the operators should be applied. The value at which the two bounds, ub-ol and Ib-o2, are equal, corresponds to the intersection of the two functions.

•The lower bound of operator 1 is assumed to be minus infinity aid the upper bound of operator two is assumed to be plus infinity.

If the value of % falls within a particular interval, then the operator to which that interval corresponds will be selected to apply to the data point If, however, the data point does not fall within an interval, then a lookahead search in which both operators are applied will be performed to make a decision about which operator to select If, after a full evaluation, *opl* is selected as the operator to apply, die data point will be learned as die new value of ub-ol. Conversely, if *opl* is selected, the value of Ib-o2 will be updated to the x value. By such a process of refining the values of the bounds, Interval-Soar incrementally converges to the intersection point of the two intervals.

Since Interval-Soar performs this task without any knowledge of the functions themselves[3], the issue is not one of solving two equations in two unknowns. Instead, it is a problem of determining the intersection point only using knowledge of the sample data set Although in its current version, Interval-Soar assumes that the functions only intersect at a single point, it is expected that this assumption could be relaxed in future versions of the system. In order to perform its task, as will become clearer later, the system requires the ability to recognize and recall declarative knowledge. However, since chunking is the only learning mechanism within Soar, this memorizing of declarative knowledge must be performed using chunking, a task not as straightforward as learning procedural knowledge.

43. Structure of Interval-Soar

The knowledge in Interval-Soar is districted among several problem spaces: the interval space, the selection space, the refine space, the function spaces and the memory space. Figure 4-1 shows the decomposition of the system into its problem spaces.

Interval-Soar is implemented by 934 (Soar 5) productions. These include the productions that encode Soar's default search-control knowledge. Table 4-1 summarises the numbers of productions required to implement the different spaces. As is seen, the function spaces required by far the largest number of productions.

4 J.L Interval Space

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

4*3.2.* Selection **Space**

The selection space contains Soar's default knowledge for resolving multi-choice impasses that arise when several competing alternatives exist In Interval-Soar, the space is used to resolve ties between the operators *opl* and *opl* in the interval space. To accomplish this, the selection space has been augmented with four additional operators: *memory, x-lte-ub-ol, x-gte-lb-ol* and

[5]Although the example problem described later depicts die forms of the functions for the purposes of illustration, Interval-Soar does not have access to this knowledge *per se.*

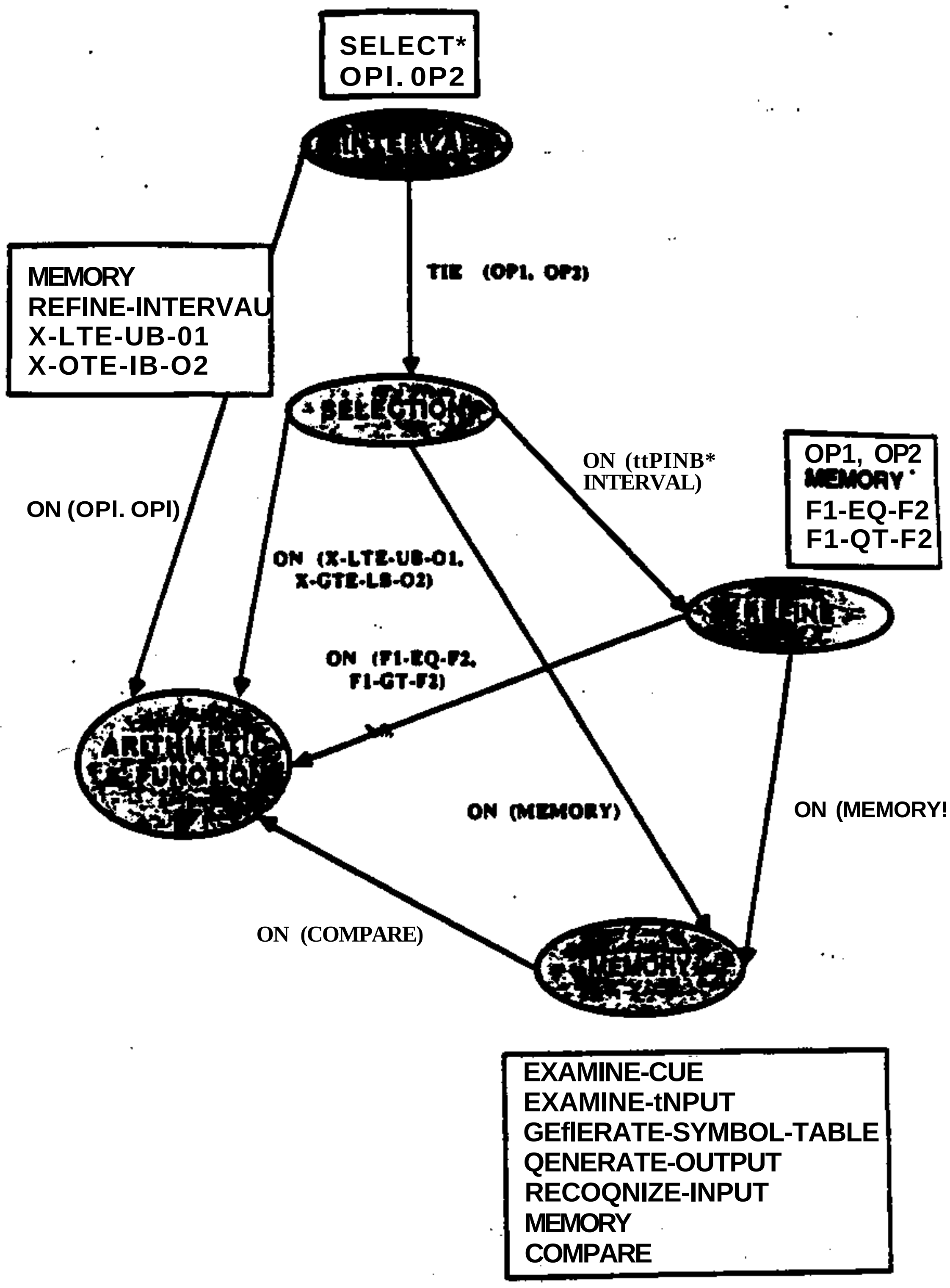


Figure 4-1: Problem-space structure of Interval-Soar

Problem Space	No. Productions
Interval	11
Selection	23
Refine	19
Memory	57
Function	810

Table 4-1: Sizes of problem spaces in Interval-Soar

refine-interval. Memory retrieves the current values of the bounds, ub-ol and Ib-o2. *X-Ite-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* refines the values of the bounds on the intervals. It does this by conducting a lookahead search on the operators *opl* and *op2$* comparing their evaluations and updating the bound of the operator that has the higher evaluation.

43.3. Refine Space **

The refine space, which implements the *refine-interval* operator, contains five operators: *opl, op2% fl-eq-f2, fl»gt*J2* 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 fl 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, Le., it learns a new bound value.

43.4. Memory Space

Although the learning of procedural knowledge in Soar only requires the application of chunking, the acquisition of declarative knowledge[6] also requires a search by the system. This additional deliberate processing occurs in the memory space, which implements the *memory* operator developed by Rosenbloom [Rosenbloom 89].

The *memory* operator provides Interval-Soar with the means to memorize and retrieve declarative knowledge. It provides the system with object recognition and recall abilities. The operator takes two arguments: an input object (the object to be learned) and a cue object The cue

^Declarative knowledge includes facts. It is knowledge about what is true in the world. Procedural knowledge, on the other hand, includes knowledge about which actions the system can perform, when certain actions should be preferred over others and how to carry out the actions. An involved discussion of representing, storing, retrieving, using and acquiring different forms of knowledge, including both procedural and declarative, is provided by Rosenbloom *et al* [Rosenbloom *et al* 89].

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 die 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 is 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.

Each application of the *memory* operator results in the learning of two recognition chunks and a recall chunk. The recognition chunks allow the system to determine if it has seen the cue and input objects before and the recall chunk allows it to generate a representation of an object seen before. By learning such chunks, Interval-Soar can recognise and retrieve previously encountered objects without having to perform a search.

Suppose it is desired 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." As a consequence of applying the *memory* operator, a chunk will be learned that delivers, Le., retrieves, the object "Fido" on any future occasion that the operator is applied with the cue "dog." *In* other words, the chunk will have the symbol "dog" as a condition and the symbol "Fido" as an action. 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 learned with the symbol "dog" as a condition and the symbol "Bozo" as an action. Thus, when the cue "dog" is encountered on future occasions, the firing of the chunk will deliver the object "Bozo" into the system's working memory. 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" and the learned objects are the values of the bounds.

Not only for its crucial role within Interval-Soar, but also because of the interesting way in which its implementation allows the chunking mechanism to acquire the recall production, the functioning of *memory* operator is significant to this work and hence is described in greater depth. However, before resorting to this, the following pertinent aspects of chunking should be recalled. Chunking operates by summarizing the processing that leads to the results of subgoals, on which a chunk's actions are based. By backtracing through the production traces to the working-memory elements that were ultimately relevant to the creation of the subgoal's results, the chunk's conditions are determined. For the functioning of the *memory* operator, the most important feature of the chunking mechanism is that productions that only generate search-control knowledge* Le.f desirability preferences, do not have their traces examined as pan of the backtracing process since these productions only affect the efficiency with which a goal is attained, and not the correctness of its results.

The key to memorising an object so that it can be recalled on presentation of a cue is to learn a chunk that can generate the object when the need arises. In Soar, this can be easily accomplished by simply making a copy of the object within a subgoal. Since this copy is a subgoal result, it will be incorporated as an action in the chunk that is learned. However, since the creation of the copy is based on an examination of the original object, the conditions of die chunk will test for the existence of the latter before generating the former. Such a chunk is clearly useless since it requires the object to be recalled to be already available. The *memory* operator overcomes this problem by creating the recalled object in two phases. First, it exhaustively generates all the possible objects that can be constructed from the symbols the system has already learned to recall. Next, the created objects are examined and the one determined to be equivalent to the original object is selected. By making a copy via such a scheme, die generation of the recalled object within the subgoal is treated as the creation of search-control knowledge. Since working-memory elements that embody such knowledge do not get backtraced from when the condition elements of a chunk are determined, the original object does not get incorporated into the chunk.

To ground this more concretely, consider the use of the *memory* operator in learning to recall the object "Fido" on presentation of the cue "(fog.*[9] In order to highlight only the important features, the description of the representation and manipulation of the input object in working memory has been abstracted from the actual processing that occurs. Suppose the operator is represented by:

```
(operator q6 "name memory ^learn 17 ^cue c8)
(object c8 ^name dog)
(object 17 ^name Fido)
```

If a no-change impasse occurs in the attempt to apply the *memory* operator, the memory space is selected and its initial state, with identifier *sll,* is augmented with the input object as well as the symbols of which the object is composed The objects the system has already seen before, and from which the recalled object will be generated, are represented as known-symbol attributes of the state. In its current incarnation, the *memory* has to be provided with these symbols *a priori,* i.e., when the system is created Suppose for this example the known symbols are *dog, name, object, Fido, Polly* and *parrot.* Hence:

```
(state all "'input 17
           "input-augmentation a21
           "known-symbol dog name object Fido
                         Polly parrot)
(augmentation a21 "class object "id 17
                  "attribute name "value rido)
```

```
(object 17 Aname rido)
```

All the possible values that each *class, id, attribute* and *value* field of the input object can acquire are next given acceptable preferences. For this example, each field has seven potential values; six representing the known symbols and an extra one representing a newly generated *id;* in this case, *x!8.* Hence the following state augmentions, representing components of the recalled-object, are given acceptable preferences (denoted by +) and placed in working memory:

```
(atate all Adass dog +)
(state all Adaas nan* +)
(state all Adass object +)
(state all Aclaaa rido +)
(state all Aclaaa Polly +)
(state sll Adass parrot +)
(state all Adaaa xl8 +)
(state sll Aid dog +)
•
•
(state sll Aid xl8 +)
(state all "attribute dog +)
•
•
(state sll Aattribute*xl8 +)
(state sll Avalue dog +)
•
•
(state all Avalue xl8 +)
```

Next, a symbol table is generated that links each input symbol to a known symbol. If an input symbol is an *id,* e.g., *17,* as in this case, it is linked to the earlier generated id, *x!8.* At this point the contents of working memory will be:

```
(atate all Ainput 17
           Ainput-augmentation a21
           Afcnoim-aymbol dog name object rido
                          Polly parrot
           Ainput-symbol object name rido 17
           * symbol-table s28)
(augmentation a21 Aclaaa object Aid 17
                  "attribute name Avalue rido)
(object 17 Aname rido)
(symbol-table s28 A17 xl8 Aobject object
                    name name
                  A               APido rido)
```

The values of the attributes of the input augmentation, *all,* are then compared to the attributes of the symbol table and best preferences (denoted by >) are generated for those state augmentations representing the components of the recalled object for which acceptable preferences were earlier created. Hence:

```
(atate til ^daaa object >)
(atate all ^id xl8 >)
(atata all ^attribute nama >)
(atata all ^value rido >)
```

Finally, the output object is created using the objects for which the best preferences were created. Working memory thus becomes:

```
(atate all ^input 17
           ^input-augmentation a21
           ^known-symbol dog name object rido
                         Polly parrot
           ^input-symbol object name rido 17
           ^symbol-table s28)
(augmentation a21 ^class object ^id 17
                  ^attribute name ^value Fido)
(object 17 ^name rido)
(symbol-table s28 ^17 xl8 ^object object
                  *n«pe name ^Fido rido)
(object x!8 ^name rido)
```

Figure 4-2 depicts the recall chunk learned for this example. Since desirability preferences were used to create the copy of the object to be learned from symbols already known, no test of the original object appears as a condition of the chunk. If the recalled object had been created by simply copying the original object in the subgoal, a chunk would have been learned in which a test of the original object would have appeared as a condition, thus indicating that for an object to be recalled, it must already be known!

The illustration just presented of the functioning of the *memory* operator only depicted the important actions. As developed by Rosenbloom, the *memory* operator, implemented as the memory space, contains six operators that produce the described learning capability. These include *examine-input, examine-cue, generate-symbol-table, generate-output, recognize-input* and *memory.* The *examine-input* and *examine-cue* operators cycle through all the symbols from which the input and cue objects are respectively constructed so that tests of these symbols appear iii the recognition chunks learned. *Generate-symbol-table* creates the symbol table relating the input symbols to the output symbols from which *generate-output* constructs the output object *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

```
(sp P38
   (goal <gl> Ast*t* <al> Aop«rator <ql>)
   (operator <ql> Anaa* memory Acum <cl>)
   (object <cl> Anmm* dog)
-->
   (operator <ql> Arecalled <xl> &, <xl> •)
   (object <xl> Aname rido •))

  If   the memory operator has been selected to apply
 and   the cue is dog

then   augment the memory operator with rido, the
       object to be recalled
```

Figure 4-2: Example recall chunk learned on application of the *memory* operator

In Interval-Soar, the memory space has been augmented with an additional operator, *compare*. This operator is needed to determine if th^ purrcnt bound is equal to the old bound 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 and consequently a chunk is also learned that automatically rejects the old bound on future occasions. If the memory space were not augmented with the *compare* operator, every time a bound value were required, the application of the operator would retrieve all the old values, not just the most recent one. Learning to reject an old object is an important feature of Interval-Soar that was not illustrated by the example of the functioning of the *memory* operator presented above. If the object "Bozo" had also been associated with the cue "dog" in the animal example, the *memory* operator would have retrieved both "Fido" and "Bozo" on presentation of the cue "dog."

43.5. Function Spaces

The function spaces, developed by Rosenbloom and Lee, contain knowledge about performing basic logical, arithmetic and control functions. This knowledge allows Interval-Soar to execute the mathematical functions it needs symbolically, such as computing fl and f2, comparing fl to £2, comparing x to ub-ol and Ib-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 is given by Rosenbloom and Lee [Rosenbloom & Lee 89].

4.4. Example of Interval-Soar Executing

This section illustrates the functioning of Interval-Soar with a simple example. Consider the two functions, f l = 7 - x a n d £ 2 » x + l , and the three data points: x « 1.3, x • 3.0 and x « 5.8. 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, fl should be applied. If it has a value greater than or equal to Ib-o2, *(2* 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 on which to work. 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 *opl* are proposed to apply. *Opl* implements fl and *opl* implements f2. Since both operators are equally acceptable at this stage, a tie impasse results.

To resolve the tie between *opl* and *opl* in the interval space, a subgoal is created and the selection space is chosen. In the selection space, 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 H&&2. 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 performing the task.

Thus, to make a decision that resolves the impasse, a lookahead evaluation of the tieing operators must be performed. To do this, an acceptable preference is generated for the *refine-interval* operator, which is then selected. If there is an operator no-change impasse in attempting to apply the *rtfine-interval* operator, the refine space is made acceptable and selected. The schemes used to evaluate the operators *opl* and *opl* are just the functions themselves. Thus, *opl* and *opl* are first applied in random order to the data point The function spaces are used to implement the operators *opl* and *opl.* In this case fl = 1.2 and f2 » 6.8. Next, the comparison operators *fl-eq-fl* and *fl-gt-fl* are selected and applied in turn to determine the relative magnitude of f1 with respect to £2. *Fl-eq-jl* 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 step 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 f1 is now passed back to the higher spaces to resolve the initial tie between *opl* and *opl.* Interval-Soar thus applies *opl* to the data point x = 5.8, which is consequently labelled op2. In this case no subgoaling into the function spaces is required to implement operator *opl.*

Chunks that were teamed earlier during the lookahead search on *opl* and *opl* fire to directly apply function £2. The state of affairs at this stage of the problem solving is depicted in Figure 4-3.

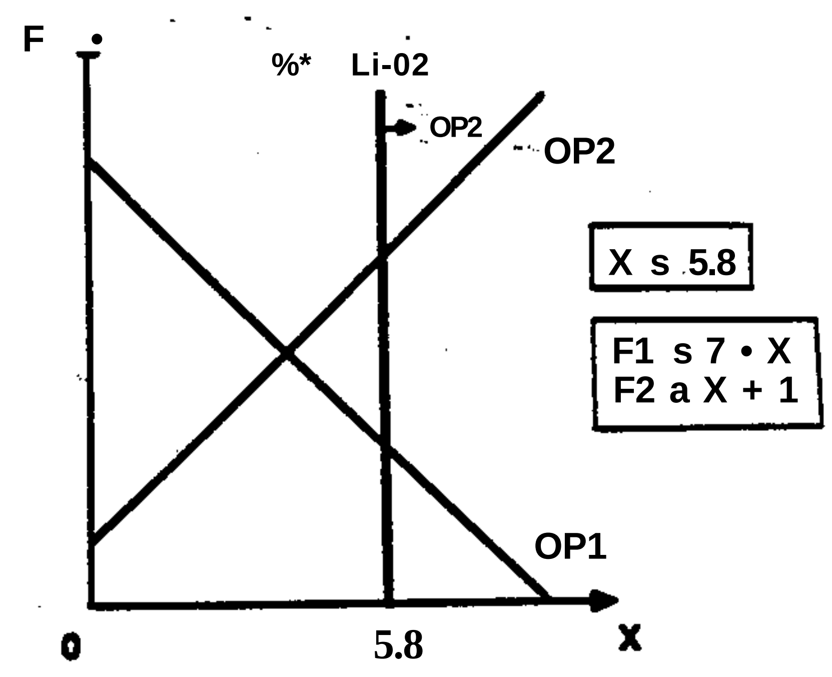


Figure 4-3: Location of bounds after first data point is employed

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 *op2* 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*gU4b-o2* 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^lb-02 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 *refitu-interval* operator is selected to compute a full evaluation once again. *Opl* and *opl* 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-/2* 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[7]. 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. 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 *(2* 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 4-4 depicts the problem solving situation at this stage.

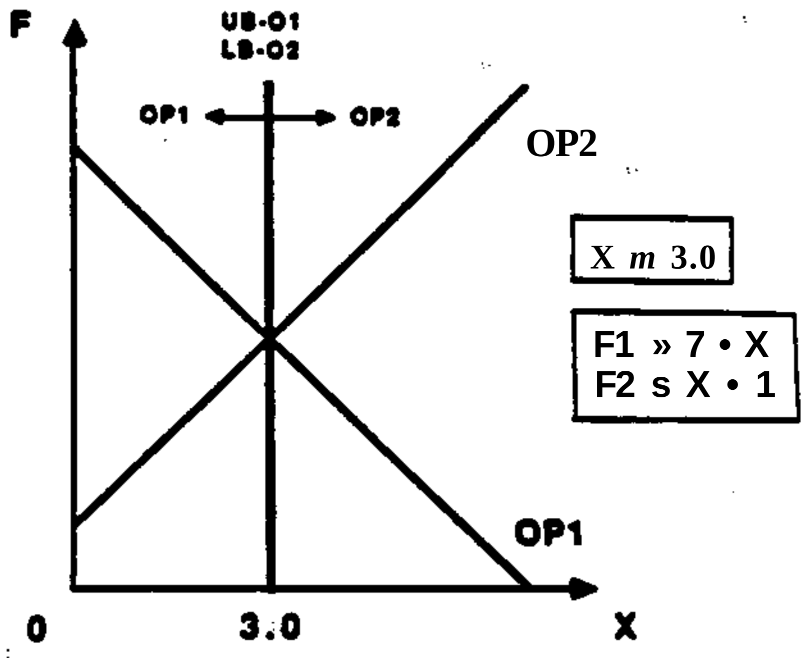
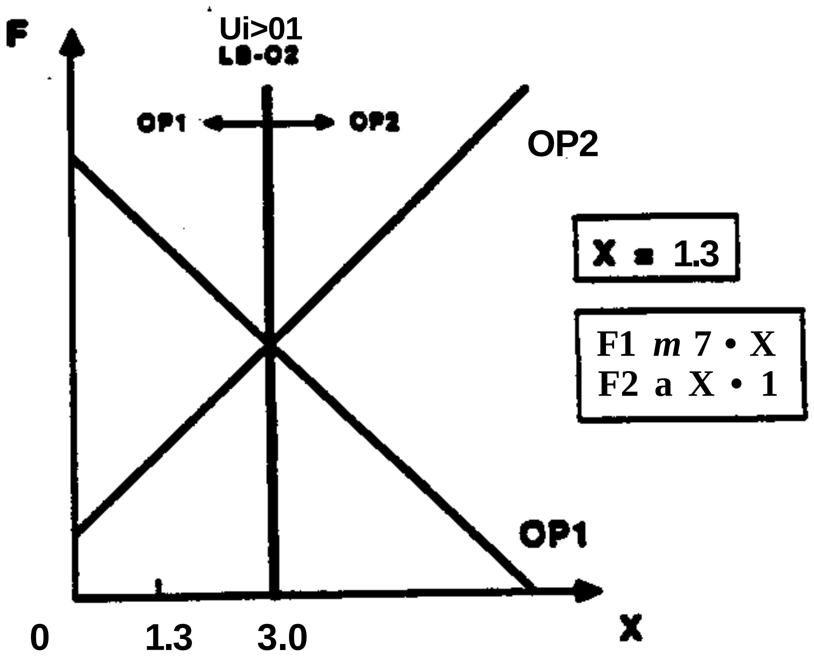


Figure 4-4: Location of bounds after second data point is employed

The final point from the set to be labelled is x * 1.3. By now the sequence of steps taken to

[7]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.

achieve this should hopefully be clear. In the selection space, memory operators are first applied to retrieve the cunent values of the bounds. In this case, both uK>l and UK>2 are associated with the number 3.0. Next, the comparison operators, *x*e-ulH>l* and x-tfe-fc-o* are selected to apply Ttie first returns a value of true since x is less than ubol, while the second returns a value offiserince x is less than Ib-o2. Since Interval-Soar possesses the knowledge that if a data £ T i s less than the bound ubol, operator *opl* should be selected, a better-Aan preference is ed for *opl* with respect to *op2*. Hence, in this case, the tie impasse in the interval space ^^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 depicted in Figure 4-5.



Figured: Location of bounds after third data point is employed

It should be noted that operator no-change impasses are encountered when attempting to apply the *memory* operator, *opl, opl* and the comparison operators *(fl-eq~f2, fl-gt-JZ, xJte-ub-ol, x-gte4b-o2* and *compare)*. 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 **are** now 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, as distinct from across-task transfer, occurs when chunks acquired when solving a problem apply when the same problem, i.e., one involving the *same* pair of functions as well as the *same* set of data points, is repeated at a later time. Here Table 4-2 illustrates the effects of across-trial transfer for the task at hand Each data set, each representing a different problem, was presented to Interval-Soar three times in succession. In each trial the system began solving its task with the knowledge it was endowed with at creation time as well as the knowledge it had acquired in all prior problem-solving trials. Thus, at the start of the second trial, the system had access to its innate knowledge as well as that learned during the first trial, and at the start of the third trial, the system's knowledge base also included the knowledge learned during the second trial. The results displayed in the table depict the effects of learning on the system's performance. For each test case, the changes in decision cycle numbers from one trial to the next are presented. As can be seen, the 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. Since most of the productions learned in Interval-Soar were arithmetic-operator-implementation chunks,^ major part of the reduction in decision cycles was due to the system not having to subgoal into the function spaces.

It should be noted that case 1 required fewer decision cycles than either cases 2 or 3 since the set of data points to be labelled in case 1 contained the actual intersection point of the functions.

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

Table 4-2: Effects of across-trial transfer of chunks: changes in numbers of decision cycles

Table 4-3 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 nonnally not acquire additional knowledge during subsequent trials 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 learning 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 die 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 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 die 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	vr65	10	0
2	76	15	0
3	76	15	0

Table 4-3: Numbers of Productions Learned during Different Trials

Table 4-4 illustrates the effects of across-task transfer, which occurs when chunks learned while solving a problem in a particular domain apply during the solution of another problem, i.e., one involving a *different* set of data points but the *same* pair of functions. It will be reemphasised that this phenomenon differs from the across-trial transfer described earlier in which the effects of learning were examined across repeated presentations of the same problem, involving both the same functions and the same data points. In the case of Interval-Soar, chunks acquired when using the data points from set 1, for instance, fire when performing the task using the data points 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 4-4: Effects of across-task transfer of chunks

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 implement an operator directly in particular situations. Before learning, such an operator, because of its complexity, would require subgoaling in order to be applied. Table 4-5 is a summary of the number of operator-implementation productions learned for the different operators in Interval-Soar.

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 Figures 4-6 and 4-7. Chunks are also acquit that reject previously learned values of the bounds. An example of such a chunk is given in Figure 4-8.

Besides operator-implentation chunks, search-control productions are also learned by Interval-Soar to resolve the tie impasses encountered between *opl* and *op2* in the interval space. An example of such a chunk is presented in Figure 4-9.

4.6. Implications for Process Design

An important contribution of Interval-Soar was the demonstration that when performing in a quantitative domain, an artificial agent must bring additional background knowledge to the learning experience in order to learn useful knowledge. In other words, what a system learns

Operator	No. Implementation Chunks Learned
opl	7
op2	5
x-lte-ub-ol	4
x-gte-lb-o2	4
memory	3
fl-eq-fl	3
fl-gt'f2	4
compare	3

Table 4-5: Numbers of operator-implementation chunks learned for different operators

from the application of an evaluation is dependent upon what model, Le_M knowledge, about the evaluation the system brings to the problem-solving experience. By endowing Interval-Soar with a model, even implicitly, of the evaluation functions it employs, it was depicted that the system could be made to learn useful knowledge about those functions.

The implications of Interval-Soar are also important for process design. To ground this more explicitly, recall again the basic problem-solving behaviour of CPD-Soar, it selects among multiple separation tasks by evaluating the competing choices via a lookahead search. The evaluation function applied computes the vapour flowrates of the separation tasks and the task with the lowest flowrate is ultimately selected. Since all that the system knows, albeit implicitly, is that the evaluation function only produces numeric values, and that these values are to be compared when making a decision, then, as has already been seen, all that will get learned is that when a particular input is presented to the evaluation function, the relative preference of one separation task over another should be delivered. Utilizing the insight obtained through the development of Interval-Soar, it is conceivable that an agent such as CPD-Soar can be made to learn more general knowledge if it were supplied with a richer model of its evaluation function than it currently has. This model, like the model embedded within Interval-Soar, could be that the evaluations provide additional information about the system's function, e.g., in what region it was applicable. Then, when the agent applied the function to select among the competing separation tasks, the knowledge learned would be about this region and hence be of more general use.

It is often the case in process design domains that multiple functions are available for evaluating candidate designs. As described in Chapter CHAP2, the differences among the underlying models and equations on which the functions are based can be characterised along a number of

```
(sp p294 elaborate
    (goal <gl> ^operator <ol>)
    (operator <ol> -^recognized input ^name memory
    ^learn <yl>)
    (pmram <yl> ^value <il>)
    (integer <il> ^sign positive ^head <cl> ^tail <cl>)
    (column <cl> ^anchor head tail ^digit <dl>)
    (digit <dl> ^name 5)
-->
    (operator <ol> ^recognized input 6, input +))
```

```
  If  the memory operator has been selected to apply
 and  the input object has not yet been recognised
 and  the object to be learned is the number 5

then  create an operator augmentation indicating that
      the input object has been recognised
```

```
(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 <cl> ^tail <cl>)
    (column <cl> ^anchor head tail ^digit <dl>)
    (digit <dl> ^name 5)
-->
    (operator <ol> ^recognixed cue-input &, cue-input +))
```

```
  If  the memory operator has been selected to apply
 and  the cue object is Ib-o2
 and  the object to be learned is the number 5

tfcen  create an operator augmentation indicating that
      the combination of cue and input objects has been
      recognised
```

Figure 4-6: Example recognition chunks learned by Interval-Soar

```
(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> ^name 5 +)
    (column <xl> ^digit <x2> +
     ^anchor head + head 4, tail 6, tail +)
    (param <x4> ^value <x3> +)
    (operator <ol> ^recalled <x4> «, <x4> +))
```

```
  If  the memory operator has been selected to apply
 and  the cue object is Ib-o2

then  augment the memory operator with the number 5/ the
      object to be recalled
```

Figure 4-7: Example recall chunk learned by Interval-Soar

```
(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> 6))
```

```
  If  the memory operator has been selected to apply
 and  the cue object is Ib-o2
 and  the object recalled is the number 5

then  create a reject preference for the recalled object
```

Figure 4-8: Example chunk learned by Interval-Soar to reject a (previously learned) bound

```
(sp pl051 elaborate
    (goal <gl> "operator <o2> + { o <o2> <ol> } +)
    (operator <o2> ^name opl *param <xl>)
    (operator <ol> ^name op2)
    (paraa <xl> ^valua <il>)
    (integer <il> ^«ign poaitive ^tail <cl> ^haad <cl>)
    (column <el> ^anchor head ^digit <dl>)
    (digit <dl> ^name 2)
-->
    (goal <gl> ^operator <o2> > <ol>))


  Zf  operator opl has been made acceptable
 and  operator op2 haa been made acceptable
 and  data point x haa a value 2

then  create a better-than preference for operator opl
      with reapect to operator op2
```

Figure 4-9: Example search-control chunk learned by Interval-Soar

other dimensions besides their form: accuracy (how well is the real situation depicted?), precision (to what order of magnitude can the results be believed?), scope (under what conditions is the function applicable?) and efficiency (how expensive is the function to apply?). Interval-Soar only has knowledge about the form of its evaluation functions. However, if a design agent was also endowed with the knowledge that its use of an evaluation function also provided information about the accuracy, precision, scope and efficiency of the function, die learning performed by the agent will be much more richer than if its a *priori* knowledge were only that the use of the function provided information about its form. This kind of learning has added value since the system will now be capable of selecting among competing evaluation functions on the basis of such characteristics as accuracy, precision, scope and efficiency. Of course, given our initial foray into this domain, much work has still to be performed to obtain a better understanding of **what** is required of an artificial agent to perform this learning. The next section postulates the structure and expected performance of GPD2-Soar, an enhanced version of CPD-Soar whose implicit model of an evaluation function is that its application provides two kinds of information: information about the form of the function and information about the error in the result obtained The use of Interval-Soar's functionality in realising the system is discussed

Learning about the evaluation functions employed by a process designer also has implications for interacting with external software systems. As described earlier, many evaluation functions are implemented as independent software tools. Because these tools often contain the expertise and cumulative results of many person-years of research, their calculations cannot be performed internally by the **design agent** Hence, each time a candidate design needs to be evaluated, an external software **system** has to be invoked. However, given the inherent complexity of most ESSs and the combinatorial size of typical design spaces, calling upon an ESS to compute an evaluation frequently will be costly. Thus, there is an incentive for the agent to be capable of calculating its evaluations cheaply, even if they are much more approximate than those produced by the ESS. In this regard, Ac functionality of Interval-Soar is very relevant If the design agent were endowed with a model of the ESS, then, like Interval-Soar has, it could learn to create the same results, perhaps at a different level of approximation or abstraction, as the ESS. This capability of an agent of performing internally the same computations as a specific ESS has been termed simulate-ESS by Newell and Steier [Newell & Steier 91].

In concluding this section on Interval-Soar, a few final remarks will be made. The development of Interval-Soar was a preliminary attempt to understand what is required by an artificial agent to learn useful knowledge when performing in a quantitative domain. In process design domains, approximate models of the evaluation functions employed by the agent can be useful in making a search cheaper. By learning such models, the agent can by-pass the employment of expensive evaluation functions by simulating their operations internally. The internal generation of abstract or approximate models by a problem-solving agent however, has only recently been recognised as an area of much-needed research. Descriptions of current research efforts in this domain can be found in Ellman *et al* [EUman *et al* 90].

5. CPD2-Soar: A Postulated Extension to CPD-Soar

5.1. Overview

This section describes how the abilities and performance of CPD-Soar can further be improved. It presents a new problem-solving strategy for the design of distillation sequences and discusses the rationale behind the strategy. The strategy draws on the lessons learned from the development of the two earlier-described systems. To implement the suggested strategy, CPD2-Soar, an enhanced version of CPD-Soar, is postulated. Its problem-space structure and expected performance are described. The section concludes by discussing the implications of CPD2-Soar for process design.

5.2. **Basis for an Improved Design System**

As mentioned earlier, one aspect in the design of distillation sequences that is largely not understood are the conditions which govern the selection of an evaluation function. Although new evaluation functions and rules are often presented in the design literature, there is hardly ever any discussion regarding the context in which the proposed functions perform well and hence should be selected. In all known design systems developed to date, the evaluation functions employed by the system to analyse design decisions and control the search are *a priori* selected by the system's developer at system-creation time. Even CPD-Soar fits this mould. It always attempts to decide among a set of competing separations tasks by first applying us heuristic rules. If the application of the rules does not succeed in breaking the impasse, a one-step lookahead evaluation is carried out and the task with the smallest vapour rate is selected to apply. At no point does die system deliberate about its choice of evaluation function.

Since the choice of an evaluation function is context dependent, a system that is capable of selecting the function as part of its problem-solving activity will clearly perform better than one in which this decision has been hardwired into the system by its creator. This section attempts to make a first pass at describing how such a system could be constructed It describes how the selection of an evaluation function could be posed as a problem-solving activity within a design system. It also describes how the system could learn to improve its evaluation-function choosing ability by capitalising on its problem-solving experiences. The capacity to perform this learning draws on an important lesson learned from the development of Interval-Soar; namely, that the better the model a system has of its evaluation functions, the more general its learning will be.

The marginal price of a separations task is its change in price as a result of performing it in the absence of non-key components. Its use as an evaluation function for controlling the search in distillation sequence design problems has been described in depth in Chapter CHAP3. The ability to select among competing evaluation functions will be illustrated by focussing upon two variations of marginal price: the marginal vapour rate *(MV)* and die marginal total annualised cost *(MTAQ*. Although the performance of marginal price as an evaluation function has been seen to be excellent, it was not used to control the search in CPD-Soar since the function was discovered after the system was created v%

The large combinatorial problem resulting even when the feed mixture consists of only a modest number of components is a major motivation for using heuristic evaluation functions to tame the search. However, although the use of these functions may result in greater computational efficiency, it is usually at the expense of solution quality. The multiplicity of evaluation functions for distillation sequence design that have been reported in the design literature are specific points along a spectrum of evaluation schemes whose extremums represent no search and exhaustive search. The selection of an evaluation function thus properly involves a trade-off between the computational resources used and the solution quality obtained

The *MV* evaluatioo function is cheaper to compute than the *MTAC* evaluation function since computing costs first necessitates computing vapour flowrates. In turn, both these functions are cheaper than a search. On die other hand, the quality of the results delivered decreases from a search to the *MTAC* evaluation function to the *MV* evaluation function. Given the potential savings in computational effort, it is beneficial to use as cheap an evaluation function as possible when solving a design task. For example, there would be no incentive for a search to be performed if the use of the *MTAC* evaluation function resulted in die same solution. Likewise, employing the *MTAC* evaluation function when the *MV* function will also deliver the same result is clearly a waste of resources. Hence, learning the conditions under which an evaluation function will deliver decisions of a certain minimum quality will be useful. However, if the quality of a solution and the conditicms under which an evaluation function should be selected

to be iwsoned about, metrics must first be devised to represent them.

are

•n. „ « « difference between the evaluation of a decision and the equation ot me oest The percentage aincren^ ^"~ ..au *or adesign decision. In the distillation sequence decision possible is used to represent*° the co * s t rf ^ WKpnet ^ g ^from me task to the domain, the evaluation^" " £ ^•val ^^ ^^ could result in different tasks being

to repre the quality of the solution that will be created as a consequence [illegible] quality of a task-selection decision, termed the *task error*, is formally defined and illustrated in the next section.

When marginal price is used as an evaluation function, the task with the smallest marginal price [illegible] of the results of the test problems used •ce.wt«H to aoolv However, a preliminary c«uuu»-v m,a«t-, rHAM **is seiectca 10 APF¹/* * f'mction and reported upon in Chapter** unArj, in gauging the performance of: the ev ua OT hwhcft ^ reladve njagnitudes indicate that the function only delivers the same resul ^ ^ ^ ^^ of the tasks are all of the marginal prices of the competing UKKS g ^ ^ ^ on m sobscrvation

^ I ^ T S S S r«^"pn^of *e sksrel^-^

decision of the desired quality. If a bound can oe P ¹ ^ ^ & dedsion of a certain minimum prices of the competing tasks can jrow ° ^ n tfAe evaluation faction should be used, quality, then the bound could be used to aecta p ^^ nl2^,e t0 the marginal prices of the To measure the magnitude of the marginal pnee ^ ^ mtroduced. The marginal price other competing tasks, the notion of a *mar** ^^ cc of me task to the lowest marginal **ratio** of a task is the inverse of the nitioo ^ ^ ^ ^^ defincs> ^ Ulustrates with

IOW a bound on this quantity can be

£ | L d m order to ensure a mmimum solution qu^ity.

SAProblem-SolvlngStratwrfCPIM-Soar . few terms are first Before outlining the steps in CFK-Sou'* *V°o* lem-solving strategy, to a proce ^ ^ ^ ^ ^ exajnple,

defi the decision set for the stream ^ ^ " J ^ ^ ^ decision sets for a 4<omponem separation problem ar ^ZZ&ZSZ and {A/BC, AB/C}.

The marginal vapour ratio of a task; is defined as:

(1)

where MV</) is the marginal vapour rate of task; and *wnj)* • «f t «w i and MV^O) is the lowest marginal vapour rate

of all the tasks **besides** *j* in die decision set To illustrate this, consider the decision set (A/BCD, AB/CD.ABC/D} **and** suppose *MV(AB/CD) < MV(A/BCD) < MV(ABC/D)*. Then the marginal vapour ratios of the tasks will be given by:

$$MVR(A/BCD) = \frac{MV(AB/CD)}{MV(A/BCD)}$$

$$MVR(AB/CD) = \frac{MV(A/BCD)}{MV(ABICD)}$$

$$MVR(ABC/D) = \frac{MV\{ABICD)}{MV\{ABCID)}$$

The *MVs* of the tasks are repectively given by:.

MViAIBCD) « ViABICD) - V{AJB)

MV(AB/CD) » V{ABICD) - V(B/Q

MV(ABC/D) = V{ABCID) - V(C/D)

where *V(J)* is the vapour rate of tasky.

Similar to the marginal vapour ratio, the ~~marginal~~ cost ratio of task *j* is defined as:

$$MCR(i) = \frac{MTAC^{*}(j)}{MTAC(j)} \qquad (2)$$

MCm

where *MTAC(j)* is the *NTTAC* of task *j* and *MTACHj)* is the lowest *MTAC* of all the tasks besides *j* in the decision set

The *vapour bound* is a metric used to detennine if a competing separation task should be selected or not The bound has an *error level* associated with it Hence, B/ is the vapour bound at error level *e*. If a distillation sequence whose cost is not more than *e%* of die cost of the best solution attainable is desired, a bound value at error level *e* will be used in making the task-selection decisions. More specifically, the bound $B_v^{€}$ will be used in selecting a separation task from each of the problem's decision sets. Any tasky for which *MVR(J)* > B/ will be preferred-Analogous to the vapour bound, the *cost bound* can also be used to detennine if a separation task should be selected or not B_e* is the cost bound at error level *e*. When evaluating a set of separation tasks using the *MCR* evaluation function, any task *!* for which AfC/?(/) > B/will be preferable.

The error, £(/)> of a task is defined as:

$$E(j) = \frac{C(j) - C(b)}{C(b)} \qquad (3)$$

where *C(j)* is the evaluation of task *;*' and *C(b)* is the evaluation of the best decision in the decision set of which *j* is a member. The evaluation of a task is a total annualised cost of the sequence segment, as determined by a lookahead search, from the task to the end of the sequence. To illustrate the task error, consider the decision set {A/BCD, AB/CD, ABC/D}. If task A/BCD is the best decision, then:

$$E'_{V}AB'_{'}CD'_{'} , \frac{CWICD) \;-\; *ABCID)}{C(ABC/D)}$$

Task errors are employed by CPD2-Soar to learn bound values. Whenever the system performs a search to select a separation task, it uses the occasion to learn the values of the vapour and cost bounds at each error level. There are two circumstances in which a search will be performed; either no bound values at the desired error level are retrieved (because the system has not yet solved enough problems to have stored values at the desired error level), or the bounds are unsuccessful in resolving the impasse. The system can only learn the bound values at *a priori* defined error levels. There are no restrictions on what or how many levels can be selected. The only constraint is that they must be decided upon at the time the system is created.

Table 5-1 depicts the steps in CPD2-Soar's evaluation strategy. It outlines the use of marginal price ratios and bound values in selecting a separation task as well as the procedures used in learning a bound value for the first time and in updating it on subsequent occasions. In presenting the strategy, the following nomenclature is employed: {D} is the current decision set, *{Mf}* is the subset of {D} for which £(/) V *e* and *MVR(j)* > 1, *[Mf)* is the subset of {D} for which £(/) > *e* and *MCR(j)* > 1, {T} is the set of preferable tasks, {N} is the set of non-preferable tasks, r_v is the task in $\{My^e\}$ with the largest *MVR* and t_c is the task in $\{M_c^c\}$ with the largest *MCR.*

CPD2-Soar evaluates competing separation tasks using a three-pronged approach. The system attempts to evaluate the tasks using the marginal vapour ratio *(MVR)* evaluation function first since this function is cheaper to apply than either the marginal cost ratio *(MCR)* evaluation function or a total search. However, if it fails to result in a decision, the *MCR* evaluation function is employed If this evaluation function also fails to resolve the impasse, a search is performed.

Steps (1), (2) and (3) of the strategy are performed to evaluate the separation tasks using the *MVR* evaluation function. If a value for the vapour bound at the desired level *e* is retrieved, the *MVRs* of the tasks are compared to it Any task whose *MVR* is greater than the bound value is preferred In other words, the selection of such a task is likely to lead ultimately to a sequence whose cost is no greater than *e%* of the cost of the best sequence in the search space. Conversely, a task whose *MVR* is less than or equal to the bound value is considered non-preferable, i.e., its selection is not likely to lead to the design goal. All the preferred tasks, the members of {T}, are given better-than preferences with respect to the members in {N}, the set of non-preferable tasks. Since all the tasks in {T} are likely to lead to the design goal, they are all made indifferent to each other. *It should be noted that the bound* B/ *implicitly plays a dual role:*

1. Compute the marginal vapour ratio, MVR, for each task in {D}.
2. Retrieve B/, the vapour bound at the specified error level e. If the bound is not retrieved, go to step (4).
3. Place any task j for which MVR(j) > B_v* in {T}. Place all other tasks in {N}. If {T} is non-empty, go to step (12); else, continue.
4. Compute the marginal cost ratio, MCR, for each task in {D}.
5. Retrieve B_e«, the cost bound at the specified error level e. If the bound is not retrieved, go to step (7).
6. Place each task j for which MCR(j) > B/ in {T}. Place all other tasks in {N}. If {T} is non-empty, go to step (12); else, continue.
7. Perform a search on each member in {D}. In doing so, identify (T) and {N}.
8. Compute E(j) for each task j.
9. Identify the sets {M/} and {M/} for all e. A task j is a member of {M/} if E(j) > e and MVR(j) > 1. A task j is a member of {M/} if E(j) > e and MCR(j) > 1.
10. Identify the task t_v. Set B_v« := L2 • MVR(t_v) if B/ was not retrieved at step (1). If B/ was retrieved, perform the assignment only if 1.2 * MVR(t_y) > B/.
11. Identify the task t_c. Set B/ := 12 • MCR(t_e) if B_e« was not retrieved at step (4). If B/ was retrieved, perform the assignment only if 12 * MCR(t_c) > B_c«.
12. Generate "better-than" preferences for all tasks in {T} with respect to all tasks in {N}. Generate "indifferent" preferences for all tasks in {T} with respect to each other.

Table 5-1: CPD2-Soar's task-evaluation strategy

one, in deciding if a particular function should be employed to evaluate the tasks; and two, in ' selecting a task from among those competing for inclusion within the design solution.

The *MVR* evaluation function may fail to deliver a design decision for one of two reasons; either no bound value is retrieved because no value had been previously associated with the bound at the desired error level or no tasks were found to be preferred. In this case, the system resorts to evaluating the tasks using the *MCR* evaluation function. The procedure followed here, depicted by steps (4), (5) and (6), are exactly the same as for the *MVR* evaluation function.

In the situation where both the evaluation functions fail to resolve the impasse, the system resorts to a search, depicted by step (7) in Table 5-1. This entails creating the entire distillation sequence of which a task is a part and costing it It should be noted that the search is recursive. When decision points are encountered further downstream, the tasks are evaluated first by the *MVR* evaluation function, then by the *MCR* evaluation function if die preceding function fails, and finally by a search if both the functions fail.

The system attempts to learn the bound values on the culmination of a search since the information required to carry out the learning is generated during the search. The learning phase is depicted by steps (8), (9), (10) and (11). The error of each task in the decision set is first computed using equation 3. Next, the set {M/} is determined, the members of which have errors larger than the desired error level and marginal prices greater than one. An error greater than the desired error indicates that if one of the^ tasks is selected, it may not lead ultimately to a distillation sequence whose cost fell within the desired percentage of the cost of the best solution attainable. It should be recalled that the value of a bound at a particular error level denotes how large the relative magnitudes of the tasks' marginal vapour rates should be in order for the *MVR* evaluation function to deliver preferred decisions. If the *MVR* of a task is greater than one, it means that it has the smallest *MV* in the decision set and hence should have been selected by the *MV* evaluation function. However, if it is not chosen, then the size of its *MVR* is an indication of how large the bound should be. Hence, the value of the bound *B*< should be at least as large as the task (r_v) with the largest *MVR* in the set {M/}. Thus, if no bound value already exists, B_v« is set to *l.2MVR(t}*. If a bound value already exists, the assignment is only performed if *l.2MVR(Q* > £/. The value of *B** is learned in a similar manner.

It should be noted that the bound values are set at a value 20% higher than the task (in {M/} or {M •}) with the highest marginal price. The 20% acts as a buffer zone and allows the system to overcome errors in the bound values. If a bound value is set to the highest marginal price, the system will have no way of checking in the future if the bound value just learned is safe enough to be used in making a decision. By setting the bound value to be slightly higher than the highest marginal price, the system will automatically check the value of the bound by conducting a search over those tasks whose marginal prices are close to the bound value. In memorizing the bound values in such a manner, it is expected that these values will converge to a steady point for each error level. Also, since marginal cost is a better (in terms of solution quality) evaluation function than marginal vapour rate, it is expected that for a given error level, the cost bound will be lower than the vapour bound, indicating that on average, the former will more often be selected than the latter.

The problem-solving strategy employed by CPD2-Soar can be summarised as follows. At first, the system attempts to evaluate competing tasks by comparing their marginal vapour ratios to the vapour-bound value at the desired error level Any task whose *MVR* is greater than the bound value is likely to lead to the design goal and hence is preferred If the use of the vapour bound is unsuccessful in resolving the impasse, the cost bound is used If this test also fails, a search is performed Knowledge generated during a search is used by the system to refine the values of the bounds.

5.4. Structure of CPD2-Soar

This section describes the problem spaces within which task-evaluation strategy described in the previous section could be implemented. The problem spaces in CPD2-Soar are essentially a combination of those in CPD-Soar and Interval-Soar. The system's knowledge is distributed among several problem spaces: the domain spaces, the selection space, the refine space, the function spaces and the memory space. Figure 5-1 depicts the major problem spaces in CPD2-Soar.

5.4.1. Domain Spaces

The top-level space in CPD2-Soar, Resign, has eight operators: *identify-forbidden, link-components, get-feed, order-components, make-splits, sequence-split, update-stream* and *write.* The latter six are implemented as the feed, order, split, update, output and sequence spaces respectively. Feed interacts with the user to obtain the feed specifications, order ranks the components in a stream in descending order according to volatility, split generates all the possible sharp splits that can be applied to the feed stream, update computes the mole fractions of a stream's components, normalises their volatilities and computes the total flowrate of the stream and write outputs the results on completion of the design. Sequence ranks all the allowable splits in the order they are to be applied It contains three operators: *make-new, compute-vaprate* and *compute-tac.* The first applies a split to a stream to generate the resulting column and product streams while the second computes the vapour flowrate of the column. The third operator, *compute-tac,* is a new addition to the sequence space. It does not exist in CPD-Soar. *Compute-tac* calculates the total annualised cost of a distillation column. The domain spaces in CPD2-Soar are depicted in Figure 5-2.

5.4.2. Selection Space

As in other Soar systems, the selection space is used to resolve multi-choice impasses. In CPD2-Soar, the space is chosen in response to ties and conflicts among competing *sequence-split* operators in the design space. In CPD2-Soar, the selection space has eight operators: *memory, evaluate-object, compute-mv, compute-mtac, compute-mvr, compute-mcr, refine-interval* and *compare.* The *memory* operator retrieves the current value of a vapour or cost bound at a specified error level. *Evaluate-object* computes an evaluation for a competing *sequence-split*

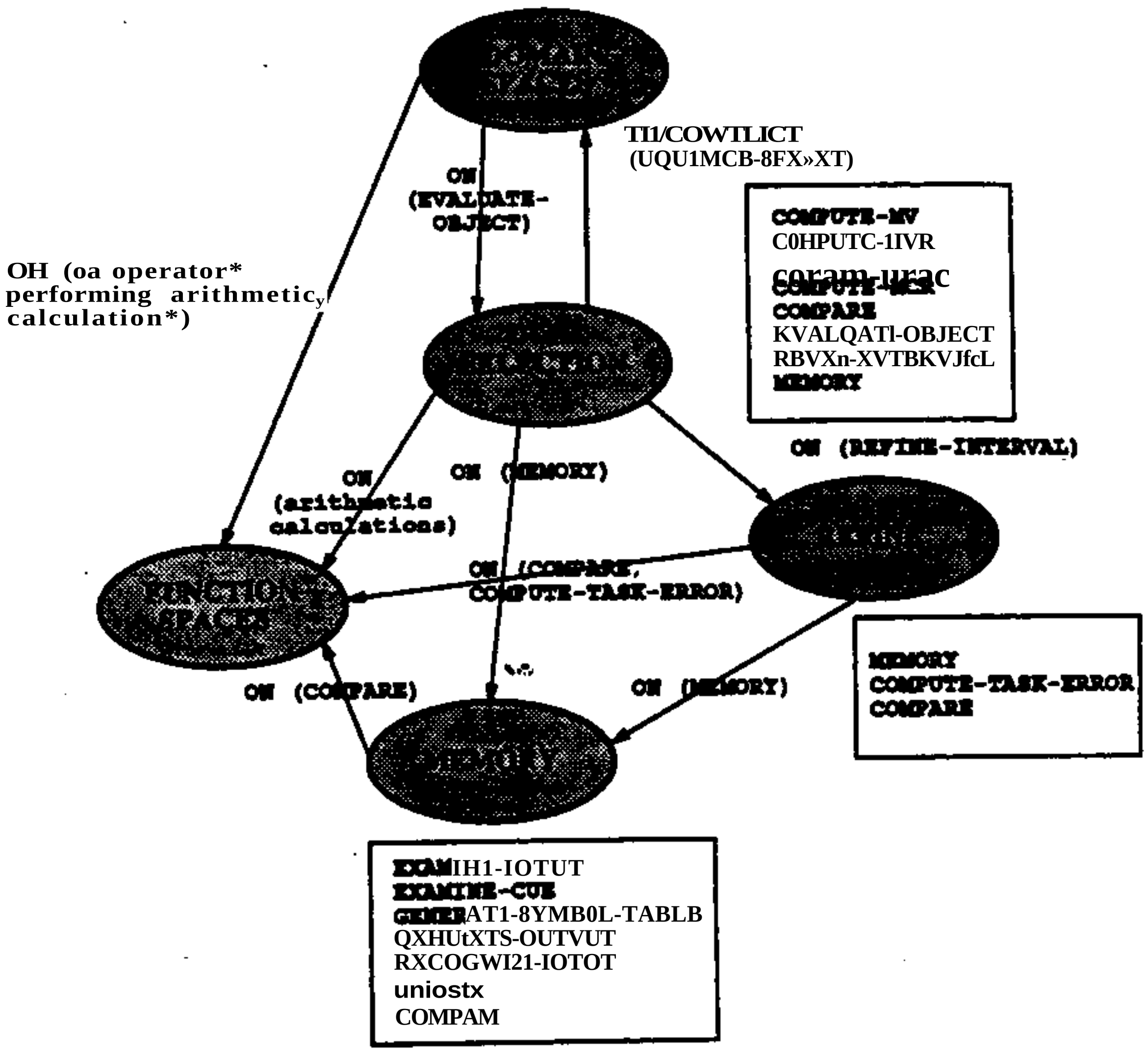


Figure 5-1: Problem-space structure of CPD2-Soar

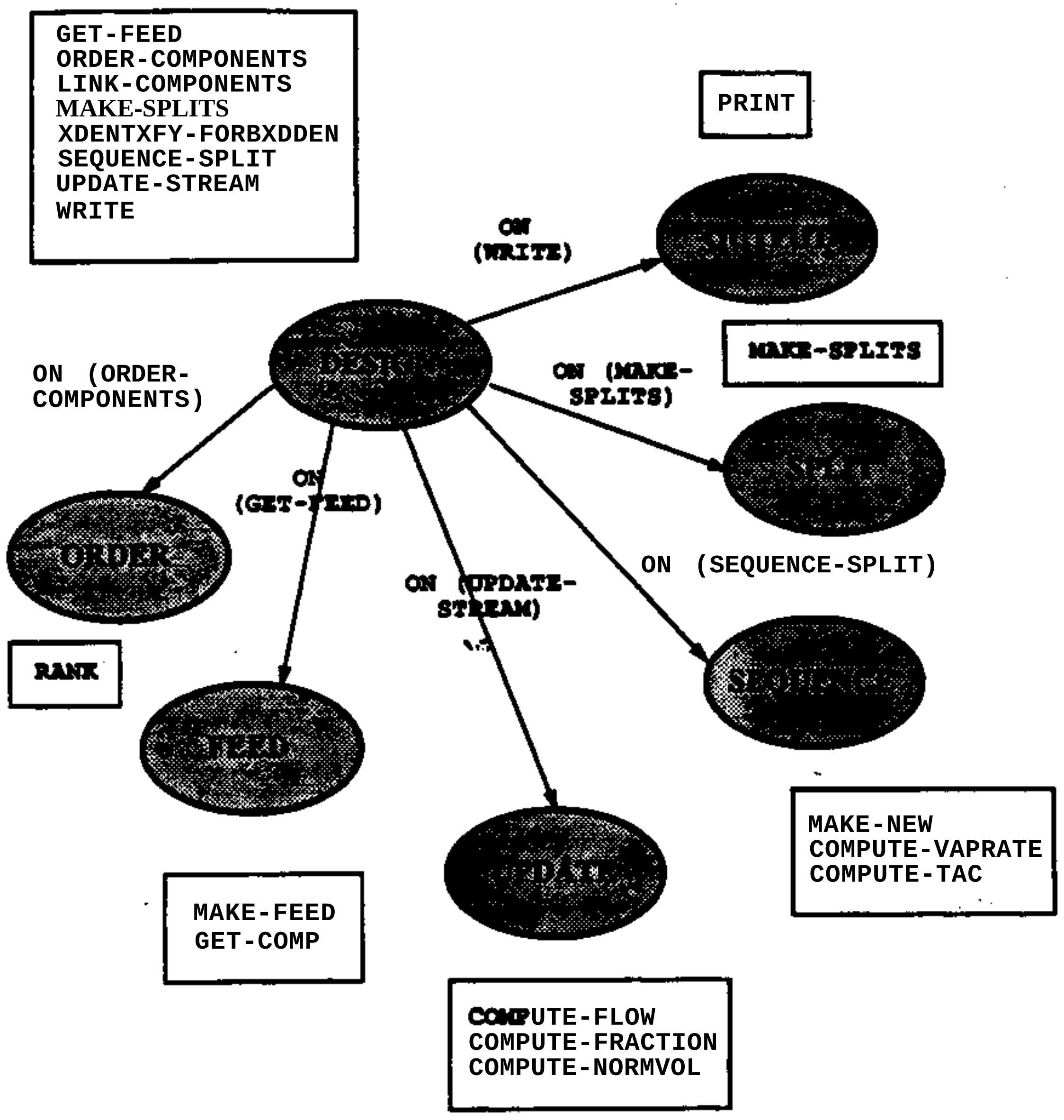


Figure 5-2: Domain problem spaces in CPD2-Soar

operator. The evaluation metric used in CPD2-Soar is the total annualised cost of the complete sequence of which the competing separation task is a part. The operators *compute-mv, compute-mtac, compute-mvr* and *compute-mcr* respectively calculate the marginal vapour rate, marginal total annualised cost, marginal vapour ratio and marginal cost ratio for a task. The *refine-interval* operator updates the value of a bound. The *compare* operator compares two numbers and determines their relative magnitudes with respect to each other.

5.4.3. Refine Space

The refine space implements the *refine-interval* operator. It contains three operators: *memory, compute-task-error* and *compare.* In contrast to its functioning in the selection space, the *memory* operator in the refine space associates the value of a bound with its corresponding cue, i.e., its application results in the learning of a new bound value. *Compute-task-error* calculates the error of a task.

5.4.4. Memory Space

The memory space implements the *memory* operator. Its functioning was described earlier in the section on Interval-Soar. In CPD2-Soar, the cues for the memory operator are the names of the bounds, either "vapour" or "cost," and their error levels. The objects to be learned are the values of the bounds.

5.4.5. Function Spaces

The function spaces provide CPD2-Soar with an elementary mathematical capability. These spaces contain knowledge that allows the system to cany out basic logical, arithmetic and control functions. Their structure and performance is described at length by Rosenbloom and Lee [Rosenbloom & Lee 89].

5.5. Illustration of CPD2-Soar Executing

This section uses an example to illustrate the major features of CFD2-Soar's problem-solving strategy. Assume that the system has been programmed to learn the bound values at the 10% error level. So as to make the example easier to follow, the example only illustrates the learning of the vapour bound. The learning of the cost bounds would be performed in exactly the same way using the corresponding cost-bound operators.

The feed mixture consists of four components: A, B, C and D. Tables 5-2 and 5-3 summarise some relevant information concerning the example problem. The first table presents the vapour rates, marginal vapour rates, marginal vapour ratios, total annualised costs, evaluations and errors of all the tasks in the search space. Thesecond table presents the costs of all the sequences in the search space and the percentages by which these costs deviate from the cost of the cheapest sequence. The information provided in the tables is intended to facilitate the verbal illustration of the search CPD2-Soar undergoes in solving the example problem. The description that follows only attempts to highlight the important features of the problem-solving activity. For example, even though the application of operators to order components or compute mole fractions may not be mentioned, it should be assumed that they are indeed performed.

Task	V (kmol/s)	MV (kmol/s)	MVR	TAC ($)	C ($)	E (%)
A/BCD	260	25	1.80	400,000	750,000	11.1
AB/CD	345	45	0.56.	250,000	675,000	0
ABC/D	210	50	0.50	500,000	850.000	25.9
A/BC	250	15	2.67	250,000	350,000	0
AB/C	340	40	0.38	230,000	455,000	23.1
A/B	235	-	-	225,000	-	-
B/CD	315	15	2.0	200,000	400,000	14.3
BC/D	190	30	0.5	250,000	350,000	0
B/C	300	-	-	100,000	-	-
C/D	160	-	-	200,000	-	-

Table 5-2: Data for example problem to illustrate the functioning of CPD2-Soar

Sequence	First Task	Second Task	Third Task	Cost ($)	Deviation (%)
1	A/BCD	B/CD	C/D	800,000	18.5
2	A/BCD	BC/D	B/C	750,000	11.1
3	AB/CD	A/B	C/D	675,000	0
4	**ABC/D**	A/BC	B/C	850,000	25.9
5	ABC/D	AB/C	A/B	955,000	41.5

Table 5-3: Cost of sequences in example-problem search space

To resolve the tie among the three *sequence-split* operators, which represent the A/BCD, AB/CD, and ABC/D tasks, the system subgoals into the selection space. In the selection space, the *memory* operator is made acceptable and selected The operator is applied to retrieve the current value of the vapour bound at the 10% error level. Since no value has previously been associated with the bound at the desired error level, the retrieval operation is unsuccessful. Hence the system attempts to resolve the tie impasse by performing a lookahead search.

To carry out the lookahead search, *evaluate-object* operators are made acceptable for each of the tieing objects, in this case, the three *sequence-split* operators. The application of these operators will yield an evaluation for each of the tieing tasks. Since the evaluation computed by the *evaluate-object* operator is the cost of the distillation-sequence segment from the task to the end of the sequence, a recursive problem-solving process is set into motion by the application of the *evaluate-object* operator.

Since the order in which the task evaluations are calculated does not matter, the *evaluate-object* operators are made indifferent to each other. Suppose the task A/BCD is chosen to be evaluated first In response to the no-change impasse that arises in trying to apply the *evaluate-object* operator, the design space is made acceptable. The task is applied to the stream and the vapour flowrate and total annualised cost of the resulting column computed. Since the bottoms stream from the distillation column is a non-product stream, two *sequence-split* operators, representing the tasks B/CD and BC/D, are proposed. A tie impasse arises, and the selection space is proposed and selected to resolve it.

To evaluate the competing tasks, the 3-stage process described above is repeated. The *memory* operator is first applied to retrieve the vapour bound. Since no bound is retrieved, it is next applied to retrieve the cost bound. Again, since no bound is retrieved, a lookahead search is performed to evaluate each of the competing tasks. Suppose the task B/CD is chosen to be evaluated first In this case the bottoms stream of the column generated has only one possible task applicable to it The C/D task is made preferable is selected immediately. Its application results in the streams (C) and (D) being created. Since these are pure-component streams, the lookahead search on the task B/CD terminates. The same process is now repeated for the BC/D task. *C(B/CD* the evaluation for task B/CD, is found to be $400,000 and *C(BC/D)* is determined to be $350,000.

At the end of the lookahead search to resolve the tie between the tasks B/CD and BC/D, which together constitute the current decision set {D}, the system attempts to learn values for the bounds. The *rejine-interval* operator is selected and the system subgoals into the refine space where the *compute-task-error* operator is applied twice to determine the error of the tasks. *E(BC/D)* is calculated to be 0% and *E(B/CD)* is calculated to be 14.3%. Since *MVR(B/CD) m* 2.0 > 1 and *E(B/CD)* > 10%, fl_v^{10} is assigned the value 2.4 (= 1.2 * 2.0) because $\{D_v^{10}\}$ • {B/CD}. Figure 5-3 pictorially depicts this situation.

At this point the system has generated all the information it requires to compute an evaluation for the A/BCD task, which is $750,000. The entire recursive problem-solving process just described is now repeated to evaluate the AB/CD and ABC/D tasks. For the AB/CD task, the evaluation is straightforward since no decision points are encountered further downstream. The system subgoals into the design space and applies the AB/CD task. The resulting streams (AB) and (CD) are then further split into pure components. *C(AB/CD)* is determined to be $675,000.

In evaluating the ABC/D task, an impasse between the tasks A/BC and AB/C is encountered.

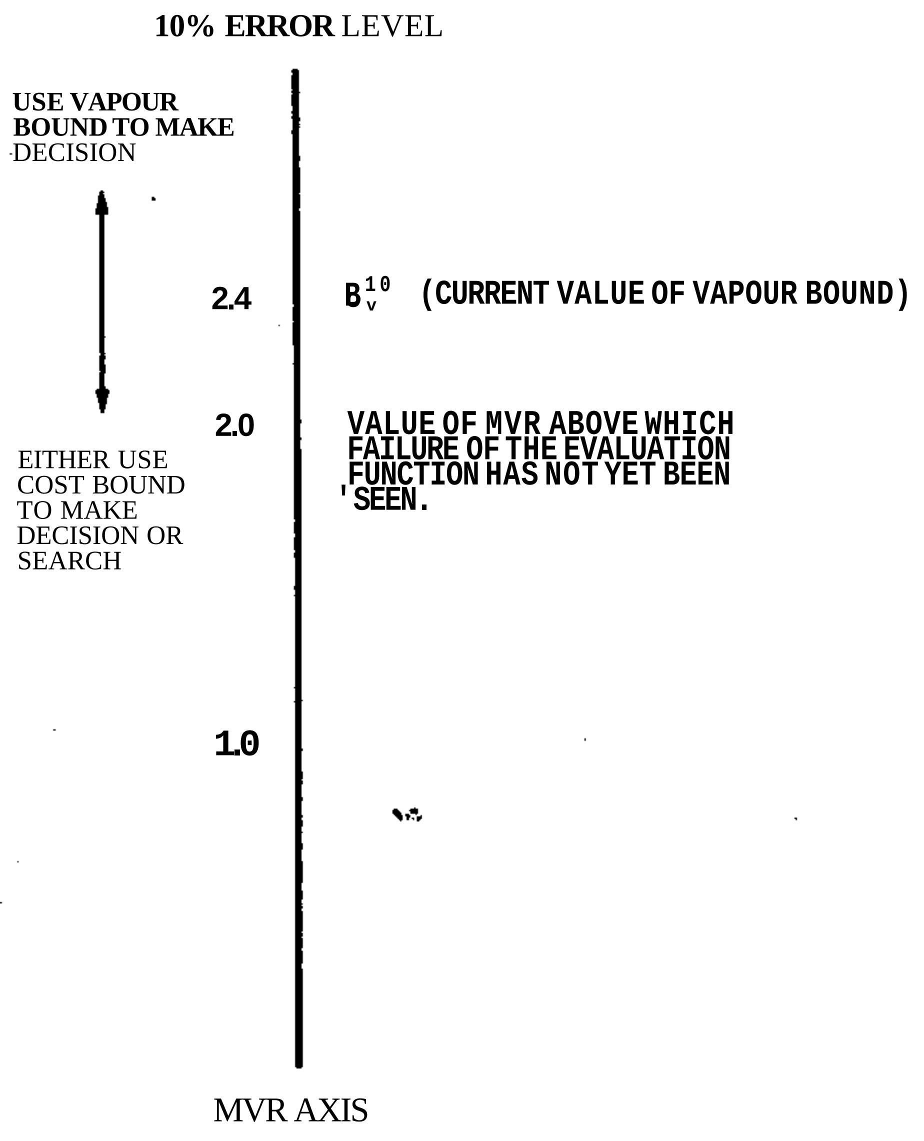

10% ERROR LEVEL
USE VAPOUR BOUND TO MAKE DECISION
2.4
B_v^{10} (CURRENT VALUE OF VAPOUR BOUND)
2.0
VALUE OF MVR ABOVE WHICH FAILURE OF THE EVALUATION FUNCTION HAS NOT YET BEEN SEEN.
EITHER USE COST BOUND TO MAKE DECISION OR SEARCH
1.0
MVR AXIS

Figure 5-3: Location of bound learned by CPD2-Soar in example problem

This time the *memory* operator is successful at retrieving a value for the vapour bound at the 10% error leveL The *compute»mv* and *compute-mvr* operators are then applied in succession to compute the marginal vapour rates and marginal vapour ratios of the tasks. The quantities arc summarised in Table 5-2. Since *MVR(AIBC)* > fl_v^{10}, the task A/BC is preferred and hence given a better-than preference than its competitor AB/C The impasse is resolved and the chosen task applied Finally, the task A/B is applied to the stream (AB) to complete the information needed to compute an evaluation for the task ABG/D, which is $850,000.

Since the evaluation process for the three initial competing tasks is now complete, the system attempts once **again** to leam values for the bounds. In response to a no-change impasse on the ***refine-interval*** **operator,** the system selects the **refine** space. Here the *compute-task-error* operator is applied thrice to calculate the errors of the tasks; *E(A/BCD), E(AB/CD* and *E(ABC/D)* are computed to be 11.1%, 0% and 25.9% respectively. In this case, D_v^{10} ={ A/BCD}. However, since *\2MVR{AIBCD)* < fl_v^{10}, the value of fl_v^{10} is not changed.

5.6. Expected Performance of CPD2-Soar

Although no data has explicitly been presented regarding the utility of the bounds, it is expected that the bounds will be useful in preventing expensive lookahead searches in many cases. The reason for this is that marginal price ratios, rather than straightforward marginal prices, are used in determining the bound values. The ratios are loaded metrics; they encapsulate a large quantity of domain knowledge. They indicate how small the marginal prices of the tasks in a decision set can become relative to each other before the marginal price evaluation function fails to deliver the correct decision.

A task with a marginal price ratio greater than one indicates that it has the smallest marginal price of all the tasks in the decision set Hence, if the marginal price evaluation function always gave the correct decision, the bound value would always be one irrespective of how close the magnitudes of the marginal prices of the competing tasks became. If a situation arose in which a task with a marginal price ratio smaller thaftvone was also the correct decision, Le., the one with the smallest evaluation, it would be a signal to the system that the marginal prices of the competing tasks were too close to each other and hence the bound on the ratio should be refined upwards.

The bound values are also a function of the task errors. The larger the magnitudes of the marginal prices relative to each other, the smaller the errors in the solutions generated. Hence, the larger the marginal price ratios or bound values, the smaller the expected errors. Figure 5-4 indicates how the vapour and cost bounds are expected to vary for different desired error levels. Since marginal cost is a better evaluation function than marginal vapour rate, it is expected that for a given error level, the cost bound will be lower than the vapour bound, indicating that on average, the former will be successful more often than the latter.

It should be noted that in the form described, CPD2-Soar takes no account of the overhead involved in computing the marginal price ratios and invoking the *memory* operator to retrieve and store bound values. Currently, it is assumed that the use of marginal price ratios is cheaper than conducting a search. However, there is certainly a cost involved in performing the operations required to use the ratios, and in reality, this cost must be compared to the cost of performing a search to determine which solution strategy is indeed cheaper.

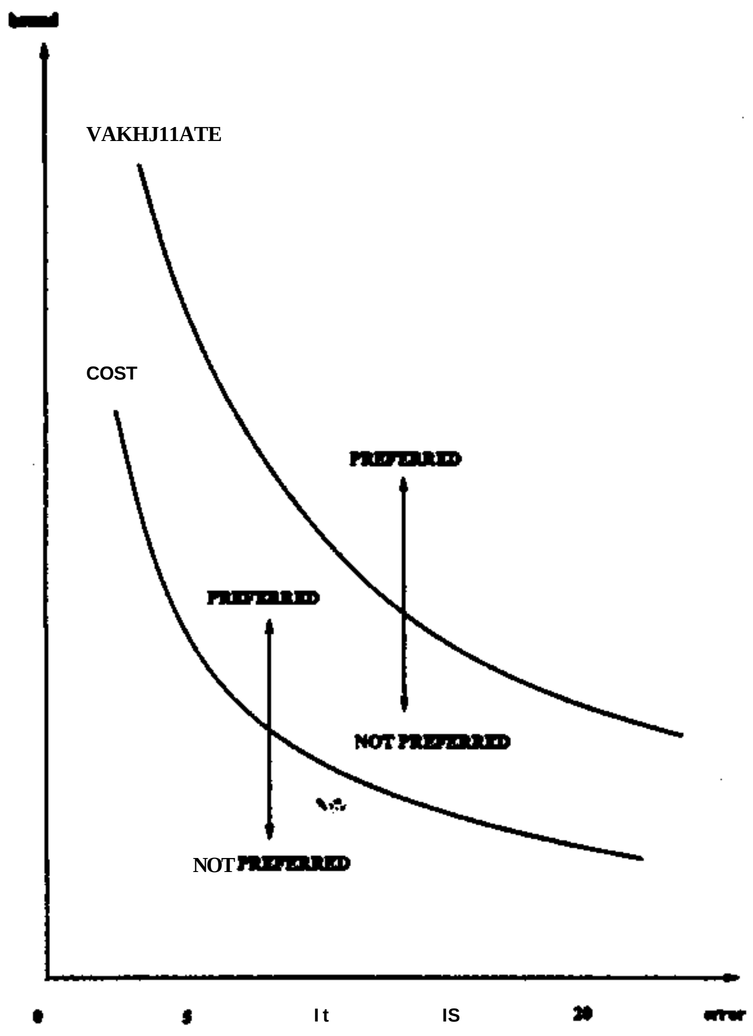


Figure 5*4: Expected variation of vapour and cost bounds with error levels

5.7. Implications for Process Design

A primary motivation for describing CPD2-Soar was to show that the functionality of a system like Interval-Soar could be fruitfully utilised within a process design system. Interval-Soar provided evidence for an hypothesis about learning; namely, that the knowledge learned by an agent is dependent upon the the models the agent brings to the problem-solving experience. The purpose of CFD2-Soar was to demonstrate for the process design domain what Interval-Soar does for an arithmetic domain; that the richer the model an agent has of its evaluation functions, the more general the knowledge it learns.

CPD2-Soar is **valuable** for two reasons: first, it recognises that the selection of a function to **evaluate competing designs** is an important, but often ignored, subproblem within a design task; and second, it **depicts** how **a** learning ability could be used by a design system to generate a design solution **that met a** minimum quality criterion without resorting to a search. These factors are further discussed below.

Learning to Select among Multiple Evaluation Functions: The selection of a function to evaluate competing designs is an important, but often ignored, subproblem within a design task. Almost all existing design systems have this decision hardwired into them when they are developed As described in Chapter CHAP2, this is tantamount to solving a design problem before it has even been posed. The selection of an evaluation function can only properly be made when the context is known within which the task is to be solved. CPD2-Soar is based on the recognition that process design is also an activity, and not just the product of an activity.

CPD2-Soar depicts how a design system can be provided with the ability to select among multiple competing functions. It depicts how the selection of a evaluation function can be posed as a problem-solving activity within a design system. It also indicates how a system could learn to improve its evaluation-function choosing ability by abstracting from its problem-solving experiences.

Generating Design Solutions that Satisfy a Minimum Quality Criterion: The goal pursued by almost all existing systems when soking a process design problem is to determine the cheapest design from among all the solutions embedded within the search space. However, a more useful goal would be to generate a solution whose cost was no greater than a certain percentage of the cost of the best solution. By stating the design goal in such a fashion, two advantages accrue: one, relaxing the condition that the best solution is required ensures that the otherwise expensive search needed to solve the task is avoided; and two, adding the condition that the cost of the solution generated be not more than a certain distance from the best solution attainable ensures that a certain minimum solution quality is maintained. At a first glance, it often seems that a potentially unsolvable conflict exists within the stated design objective. To determine if a particular solution lies within a certain horizon from the best solution, the best solution must first be found. But finding it requires an exhaustive search of the design space, precisely what **the** revised design objective is intended to avoid.

CPD2-Soar depicts how the apparent conflict in the design goal can be overcome by capitalizing on a system's learning. To determine if a design solution lies within a certain distance of the best solution, the system uses knowledge learned from earlier problem-solving experiences. In so doing, the system avoids a search of the design space. In CPD2-Soar, the vapour and cost bound metrics play an important role in realizing this ability.

A definition of a design goal that includes a bound on the solution quality also has a merit in addition to the ones described earlier. In so stating the goal, an attempt is also made to account for the inherent impreciseness of the models and evaluation functions used to analyse designs. In other words* the usefulness of all evaluation metrics in comparing the costs of competing designs only extends to a certain **degree.** To illustrate this, consider a column-costing model whose preciseness is **hundreds** of thousands of dollars. Then, a column whose cost was predicted to be $345,873 by the model should really be equivalent in quality to a column whose cost was calculated to be $278,132. If the preciseness of die model is taken into consideration, the cost of both columns can only be believed to be $300,000 with any certainty. Thus, any additional effort expended in finding the solution that cost $278,132 is wasted since the seemingly better solution is not really better if the preciseness of the models is also taken into account

6* Summary

The Soar architecture was introduced as a vehicle for developing design systems with capabilities seen to be important for* chemical process domains but missing in most existing design systems. Two systems developed within the Soar framework, CPD-Soar and Interval-Soar, were reported upon. The tasks, problem-space structure, operation and performance of each system was described in depth. The implications for process design of the systems were also discussed.

The construction of CPD-Soar and Interval-Soar was a valuable exercise for two main reasons: one, it presented evidence that the mechanisms present in Soar can provide design systems with useful abilities, and two, the act of creating the systems was helpful in identifying those aspects of the task domain that are well understoo&and those that are not Selecting among competing evaluation functions and design methods, learning approximate and/or abstract models of complex evaluation functions and interacting with external software systems were three areas identified as not being well understood.

The systems also provided evidence for the hypothesis that the learning carried out by a system is related to the model the system brings to the problem-solving experience. This hypothesis was proposed in response to CFD-Soar's learning behaviour, in particular, its learning of very specific knowledge. Interval-Soar verified that if an agent was provided with a richer model, in this case, of its evaluation function, the learning carried out by the agent would be more general and thus more uscfuL

Finally, a number of effective ways of further improving the abilities and performance of CPD-Soar, utilising **die** lessons learned earlier, were described. The problem-space structure and expected performance of CPD2-Soar, the enhanced version of CPD-Soar, was postulated.

1: Acknowledgements

We would like to thank Paul Rosenbloom for the use of die *memory* operator and, along with Soowon Lee, for the use of the function spaces. This work was supported by the Engineering Design Research Center, a National Science Foundation Engineering Research Center at Carnegie Mellon University.

List of Symbols

B_c^*	Cost bound at error level *e*
B_v^*	Vapour bound at error level *e*
$C(j)$	Evaluation of task *j* [$]
{D}	Decision Set
	Error of task *j*
M_c^*	Subset of tasks *j* from {D} for which $E(j) > e$ and $MCR(j) > 1$
	Subset of tasks *j* from {D} for which $E(j) > e$ and $MVR(j) > 1$
$MCR(j)$	Marginal cost ratio of task *j*
$MTAC(j)$	Marginal total annualised cost of task *j* [$]
$MTAC^*(j)$	Lowest *MTAC* of all tasks besides *j* in the decision set of which *j* is a member [$]
$MV(j)$	Marginal vapour rate of task *j* [kmol/hr]
$MV^*(j)$	Lowest *MV* of all tasks besides *j* in the decision set of which *j* is a member [kmol/hr]
$MVR(j)$	Marginal vapour ratio of task *j*
{N}	Set of unacceptable tasks
{T}	Set of acceptable tasks

Arguments, Subscripts and Superscripts

b	Refers to the task in a decision set with the best evaluation
e	Error level on a bound or the desired error for a problem
j	Refers to a task
t_c	Refers to the task with the largest *MCR* in M_c^*
t_v	Refers to the task with the largest *MVR* in M_v^*

References

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

[Ellmanera/90] Ellman, T., R. Keller & J. Mostow (editors).
Automatic Generation of Approximations and Abstractions - Working Notes of the AGAA-90 Workshop, Boston, MA.
American Association of Artificial Intelligence, 1990.

[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 *etal* 86] Laird, J. E., P. S. Rosenbloom & A. NewelL
Chunking in Soar The anatomy of a general learning mechanism.
Machine Learning 1(1):11-46,1986.

[Laird *et al* 87] Laird, J. E., A. Newell & P. S. Rosenbloom.
Soar: An architecture for general intelligence.
Artificial Intelligence 33(l):l-64,1987.

[Laird *et al* 90] Laird, J. E., C. B. Congdon, E. Altmann & K. Swedlow.
Soar User's Manual: Version 52.
Technical Report CMU-CS-90-179, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1990.

[Mitchell *et al* 86] Mitchell, T. M., R. M. Keller & S. T. Kedar-CabeUi.
Explanation-based generalization: A unifying view.
Machine Learning l(l):47-86,1986.

[Newell 90] Newell, A.
Unified Theories of Cognition.
Harvard University Press, Cambridge, MA, 1990.

[Newell ASteier 91]
Newell, A. & D. Steier.
Intelligent Control of External Software Systems.
Technical Report EDRC-05-55-91, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA, 1991.

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

[Rosenbloom & Laird 86]
Rosenbloom, P. S. & J. E. Laird.
Mapping explanation-based generalization onto Soar.
In *Proceedings of AAAI-86,* pages 561-567. 1986.

[Rosenbloom&Lce89]
Rosenbloom, P. S. & S. Lee.
Soar arithmetic and functional capability.
1989.
Unpublished code and working notes.

[Rosenbloom *etal* 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.

[Steier *et al* 87] Steier, D. M., J. E. Laird, A. Newell, P. S. Rosenbloom, R. A. Flynn, A. Golding, T. A. Polk, O. G. Shivers, A. Unruh & G. R. Yost
Varieties of learning in Soar.
In *Proceedings of the Fourth International Workshop on Machine Learning,* pages 300-311. 1987.