The Knowledge Engineering Review

Generic tasks as building blocks for knowledge-based systems: the diagnosis and routine design examples

B. CHANDRASEKARAN

Laboratory for Artificial Intelligence Research, Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210

Abstract

The level of abstraction of much of the work in knowledge-based systems (the rule, frame, logic level) is too low to provide a rich enough vocabulary for knowledge and control. I provide an overview of a framework called the Generic Task approach that proposes that knowledge systems should be built out of building blocks, each of which is appropriate for a basic type of problem solving. Each generic task uses forms of knowledge and control strategies that are characteristic to it, and are in general conceptually closer to domain knowledge. This facilitates knowledge acquisition and can produce a more perspicuous explanation of problem solving. The relationship of the constructs at the generic task level to the rule-frame level is analogous to that between high-level programming languages and assembly languages in computer science. I describe a set of generic tasks that have been found particularly useful in constructing diagnostic, design and planning systems. In particular, I describe two tools, CSRL and DSPL, that are useful for building classification-based diagnostic systems and skeletal planning systems respectively, and a high level toolbox that is under construction called the Generic Task toolbox.

1 Need for task-specific tools

The current generation of knowledge-based system (KBS) languages—those that are based on rules, frames, or logic—do not distinguish between different types of knowledge-based reasoning. For example, one would expect that the task of designing a car would require significantly different reasoning strategies than the task of diagnosing a malfunction in a car. However, these methodologies apply the same strategy (fire the rules whose conditions match, run resolution engine on all propositions etc.) to both design and diagnosis, as well as any other task. Because of this, it has been argued that these methodologies, although useful, are rather low-level with respect to modelling the needed task-level behaviour. In essence, these systems resemble an assembly language for writing KBSs. While they are obviously useful, clearly approaches that more directly address the higher level issues of knowledge-based reasoning are needed for the next generation of AI system development.

One example of a higher level approach is the *generic task* (GT) (Chandrasekaran, 1983, 1986, 1987). The aim here is to identify "building blocks" of reasoning strategies such that each of the types is both generic and widely useful as components of complex reasoning tasks. We have identified a number of such generic strategies, which together capture the functionality of a large portion of current expert systems. Each generic task* is characterized by:

* Each GT is a *strategy* from the viewpoint of the problem which it is helping to solve. It is a *task* from the viewpoint of the functionality that is to be achieved by the GT. It is in this sense that a GT is both a strategy for a task and a task in itself. This distinction is in fact much more general: e.g. diagnosis is a strategy in making patients feel better (i.e. one way to organize therapeutic actions is to base them on causes, finding which is, in fact, diagnosis), but it is a task which many expert systems are designed to solve.

1 The kinds of information it takes as input for the task and the information produced as a result of performing the task. This defines the functionality of the task.
2 A way to represent and organize the knowledge that is needed to perform the generic task. This includes a vocabulary of knowledge types, i.e. knowledge constructs that the representation language should have.
3 The process (algorithm, control, problem solving) that the task uses. This provides a vocabulary for inference and control for the task.

The GT framework proposes that, as each task and its associated structure are identified, languages be developed that encode both the problem solving strategy and knowledge that is appropriate for solving problems of that type. These languages facilitate KBS development by giving the knowledge engineer access to tools which work closer to the level of the problem, not the level of the implementation language such as rules or frames. However, for non-trivial problems it may be necessary to decompose it into subproblems such that each matches the functionality of some GT. For example, we will show how certain kinds of diagnostic problems can be decomposed into a number of GTs. This way of building complex KBSs also means that knowledge engineering environments should provide a *toolset* rather than a single tool.

This style of supporting higher-level generic computational activities with appropriate constructs is well-known in computer science in general; high-level programming languages are attempts to provide the programmer with constructs for a variety of common functions. It is items 2 and 3 above, *viz knowledge* and *inference*, that make the above specification different from the standard high-level language constructs in computer science and make it particularly appropriate for knowledge-based problem solving.

2 Some generic tasks and their specifications

While the approach has relevance to AI in general, as a practical matter, our work has concentrated on information-processing strategies useful for building systems for knowledge-rich problem solving, or so-called expert systems. Such systems emphasize the role of large amounts of domain knowledge compiled for specific problem solving tasks that characterize routine human expert behaviour. Without intending any kind of completeness, we list here some of the tasks that we have found to be very useful in building practical knowledge-based systems. As we will see, a variety of diagnostic and routine design and planning problems come under this category. The description of the tasks in this section is necessarily cryptic and somewhat oversimplified, and is provided mainly for a quick overview and comparison. Hierarchical classification, hypothesis matching, and plan selection and refinement are described in some detail in later sections of the paper, and abductive assembly described in somewhat less detail. Please see the citations for details on the rest of the generic tasks.

Each description is organized by the function of the task, the tool in our GT toolset for the task, the knowledge and inference types that the tool supports, and other relevant annotations. In each case, the GT tool commits itself to *one* way of achieving the functionality. Also, in each case the control behaviour described is the default behaviour, and it should be thought of as describing a family of control behaviours.

2.a Hierarchical classification

Task specification

Input: given a situation description in terms of features. Output: classify it, as specifically as possible, in a classification hierarchy. (Multiple classifications, where different classes characterize different parts of the situation description, are also possible.)

GT tool

CSRL (Bylander and Mittal, 1986) (Conceptual Structures Representation Language).

How CSRL works:
Forms of knowledge
Classification hierarchy, access to knowledge that helps decide about how well the data match the classificatory concepts (see Hypothesis Matching, below)
Inference and control
(Simplified) *Establish* nodes, if successful *refine* the concept by considering children, if unsuccessful, *reject* node, and *prune* subtree.
Example use
Medical diagnosis can often be viewed as partly a classification problem (Gomez and Chandrasekaran, 1981). Problems may be classified into types which may then suggest methods of solutions. The diagnostic portion of MYCIN (Shortliffe, 1976) (see Clancey, 1985) and PROSPECTOR (Duda, Gaschnig and Hart, 1980) can be viewed as classification problem solving.

2.b Hypothesis matching

Task specification
Input: given a concept (a hypothesis) and set of data (features) that describe the problem state. Output: decide how well the concept matches the situation. The task is a form of *recognition* (Josephson et al., 1988).
GT tool
HYPER (HYPothesis matchER) (Johnson, 1986).
How HYPER works:
Forms of knowledge
An hierarchy of evidence abstractions, lowest level at the level of data and the highest level at the level of the concept. Each node abstracts from its children into evidence about a higher level feature. In the particular version considered in our work, the abstractions are qualitative degrees of confidence. (See detailed description of HYPER, section 3.c.3).
Inference and control
At each level a degree of confidence in the presence of the feature is computed from the features that constitute evidence for it, and this is performed recursively until a degree of confidence for the concept is computed. The basic theory is that recognition of a complex concept is performed by hierarchically computing intermediate abstractions from raw data.
Example use
Samuel's *signature tables* perform this kind of abstraction. Many forms of *recognition* can be performed by means of this strategy. For example, the concept may be a disease and the data may be patient data relevant to the disease, and we wish to know what the likelihood of the disease is. Bylander and Johnson (1987) discuss a class of strategies called *structured matching* (of which the HYPER strategy is a particular example) and show how ubiquitous it is in knowledge-based reasoning.

2.c Knowledge-directed information passing

Task specification
Input: Given attributes of some data entities. Output: determine the attributes of other data of interest, but not directly known, but can be inferred from the available data.
GT tools
IDABLE (Intelligent DAta Base LanguagE).
How IDABLE works:
Forms of knowledge
Data concepts organized as a frame hierarchy of types and subtypes of data objects in the domain.

Data attributes are slots with default values, default methods of computation of values, or explicit procedural attachments which specify how to compute data attributes from related data.

Inference and control

Data queries result in the default value being chosen if no information is available, or the knowledge-intensive procedures to be invoked for inference. The inference procedures themselves can be inherited from parent concepts as needed.

Example use

A diagnostic system may use a knowledge-directed database of this type for converting from sensor or chart values into data of direct relevance to diagnosis. Clancey's data abstraction component of heuristic classification (Clancey, 1985) can be achieved by this functionality.

2.d Synthesis by plan selection and refinement

Task specification

Designing an object (device, program, plan) by hierarchical planning. Input: given specifications of the object to be designed. Output: generate design of an object (device, program, plan) meeting the specifications.

GT tool

DSPL (Design Specialists and Plans Language) Brown, 1984; Brown and Chandrasekaran, 1986.

How DSPL works:

Forms of knowledge

Hierarchical structure of the object to be designed known in advance (making it routine design). For each node in the hierarchy, precompiled design plans are known for making design choices. Failure handling knowledge available, and some parts of the plans are constraint satisfaction knowledge, i.e. knowledge of constraints to be met by the design parameters.

Inference and control

Top down control is typically used. Design plans are chosen, choices made at that level of abstraction, and design *refined* by calling on plans for the children (i.e. components). Plan failures are passed up until failure handling knowledge is available to fix the design or choose alternate plan.

Example systems

The task performed by the expert system MOLGEN (Friedland, 1979) and R1 (McDermott, 1982) can be viewed in this way. A variant of this task has also been called *skeletal planning* in the literature.

2.e Abductive hypothesis assembly

Task specification

Input: Given a situation description and a set of hypotheses each explaining some aspects of the situation and each with some plausibility value. Output: Construct a composite hypothesis that is the best explanation of the situation, i.e. explains all the data parsimoniously and as well as possible.

GT tool

PEIRCE (named after C. B. Peirce, who first described the form of inference known as abduction) (Punch, Tanner and Josephson, 1986).

How PEIRCE works:

Forms of knowledge

Causal or other relationships between hypotheses (such as incompatibility, special case of), relative significance of data describing the situation.

Inference and control

Assembly and criticism alternate. At each stage during assembly the problem solving is driven by an attempt to explain the most significant datum remaining unexplained. The best hypothesis that

offers to explain is added to the composite hypothesis. During criticism, explanatory superfluous parts are removed. This loops until all the data are explained or no hypotheses are left.

Example use

This task is a subtask in diagnostic reasoning as well as in theory formation in science. Portions of INTERNIST (Miller, Pople and Myers, 1984) and DENDRAL (Buchanan, Sutherland and Feigenbaum, 1969) systems perform this task.

2.f Some implications of GTs

The following general points about GTs are worth noting at this point.

1 As mentioned, a number of well-known expert systems can be thought of as decomposable into subtasks, each of which corresponds to one of the above tasks. R1 performs a simplified type of plan selection and refinement. Mycin performs classification and data abstraction (one of the capabilities of our knowledge-directed information passing) in the diagnostic part, and plan selection (in the therapy part). Additional examples can be given. Note, however, that in all these instances, we are only pointing out that these systems perform these tasks, but not necessarily in the manner that we propose that the tasks be performed. Our claim will be that once we understand the knowledge requirements and the inference strategies for each of the tasks, we can use methods that are more natural for the tasks.

2 We have mentioned diagnosis a number of times in the above description as a problem that uses one or more of the above generic tasks, e.g. classification and hypothesis assembly. Note that, correspondingly, we do not have a generic task called diagnosis in the above list. The reason for this is that, while diagnosis is "generic" in the sense that the problem occurs in a number of domains and there are similarities in the methods that are domain-independent, it is still a "compound" task in the sense that a number of distinct types of knowledge and inferences are used in the process of doing diagnosis. Thus the above list of tasks can be used as natural building blocks for putting together a diagnostic problem solver. (We shall soon describe how this can be done.) This illustrates an additional constraint in our sense of generic task. The task needs to have a coherence and simplicity to it in that it ought to be characterizable by a simple type of knowledge and a family of inference types. This is what makes them "building blocks".

3 Functional modularity is an important consequence of this point of view. As we have shown in a number of papers (Bylander, Smith and Svirbely, 1986; Chandrasekaran, 1983, 1986, 1987; Chandrasekaran, Tanner and Josephson, 1988) this functional modularity makes system building and debugging easier, and the task-specific knowledge and control constructs help in knowledge acquisition and explanation.

4 The above list is not meant to be a complete list of generic tasks useful in knowledge-based problem solving. In fact, quite a large part of AI—from weak methods to qualitative simulation to scripts and plans—can be thought of as attempts to identify interesting problems of some generality, the kind of knowledge required for them and the kinds of inferences useful to perform them. Thus, in qualitative reasoning, the generic problem considered is one where given the structure of a system, the task is to derive the system's behaviour in a qualitative way. The research program then identifies the knowledge and inference for the task. In the appropriate context, each of them can be thought of as possible generic tasks. Our goal in the development of the generic task theory and the toolset has been to produce a methodology and a technology that helps in the analysis, design, construction and debugging of practical knowledge systems and thus we concentrated on the generic tasks that we felt would be most useful at this stage in the development of the technology. Research is underway in our laboratory on other such generic tasks, which would cover phenomena in deep models such as structure to behaviour reasoning and functional reasoning.

In the next sections I will describe how certain kinds of diagnostic and design problems can be built in the GT framework.

3 Diagnostic reasoning

3.a Information processing in diagnosis

Abstractly, diagnosis is the problem of finding a cause or set of causes that "best explain" a set of observations of a system, some of them indicating behavioural abnormality. In most non-trivial cases, the process is a form of *abductive* reasoning, i.e. the diagnostic conclusion is not deductively provable, but is the hypothesis that makes best sense taking all the information into account.

We can identify a class of systems that we call *compiled knowledge systems* for diagnosis. These systems have knowledge that is needed for diagnosis precompiled. This would include, at a minimum:

- knowledge of possible causes in terms of which the diagnostic answer will need to be given;
- knowledge that helps map from observations to possible causes, i.e. evaluate how likely a given cause or subset of causes might be given the set of observations.

Many of the well-known diagnostic expert systems, e.g. Mycin (Shortliffe, 1976), Internist (Miller, Pople and Myers, 1984), MDX (Sticklen, 1987), set-covering and Bayesian diagnostic systems have this knowledge compiled in the knowledge base. Where these systems differ is in the *form* this knowledge takes, in the way the actual inference processes work, and also in the control of reasoning. Such compiled knowledge systems concentrate their problem solving behaviour only on the specific diagnostic problem at hand, rather than in activities that produce the needed knowledge. These systems ought to be contrasted with diagnostic systems which do not have the needed diagnostic knowledge in a readily usable form (or the diagnostic knowledge is incomplete), but must acquire them by other kinds of problem solving, e.g. by *deriving* it from structural models of the device under diagnosis, or from analysis of past diagnostic cases involving the device. See (Chandrasekaran, Smith and Sticklen, 1987) for a discussion of the general issues surrounding the use of deep models and (Sticklen, Chandrasekaran and Josephson, 1985) for a discussion of how such model-based reasoning and compiled reasoning can be interlaced. The diagnostic architecture that I will be discussing in the following pages is a compiled knowledge architecture.

Diagnosis can be computationally complex: even with compiled knowledge, all subsets of hypotheses may in principle need to be evaluated and compared. This is mainly due to two reasons: one, hypotheses may interact, i.e. two hypotheses together may account for more or less observations than the union of the sets of observations that they explain individually; and two, different subsets may explain the same sets of data and principles of parsimony will need to be brought in to choose the better explanation. All diagnostic systems, be they formal, such as set-covering and Bayesian approaches, or "heuristic", such as Mycin, either squarely face this problem and end up with computationally intractable algorithms, or make more or less realistic assumptions about the domain that help them cut down the exhaustive search through the space of hypotheses combinations. (For example, its domain is such that Mycin can implicitly make assumptions of no interaction between diagnostic hypotheses.)

Our GT architecture broadly decomposes the problem into a *classificatory* one, which generates highly plausible diagnostic hypotheses, which are then used by an *abductive assembly* component to produce the best composite hypothesis for the problem. The overall decomposition above brings significant computational advantages, since the assembly process now only needs to work with a much smaller number of initial hypotheses, with the option to seek out less plausible hypotheses as needed for explanatory completeness. This, in conjunction with the computational efficiencies that the proposed architectures for classification and abductive assembly individually possess, makes the above architecture computationally attractive whenever knowledge is available in appropriate forms: e.g. hierarchies for classification and explicit knowledge about causal and logical interactions among diagnostic hypotheses for the abductive assembly component.

3.b A GT architecture for diagnosis

The architecture has four components: *hierarchical classification*, *hypothesis matchers*, *abductive assembly* and *knowledge-directed data abstraction and inference*. The hierarchical classifier navigates the space of malfunctions organized as one or more hierarchies. The hierarchical organization permits a quick determination of the plausible hypotheses with minimal search through the space of all possible hypotheses. The result of the classification process is a small set of highly plausible hypotheses.

The classifier itself needs a mechanism to evaluate the degree of plausibility of each of the hypotheses. The knowledge necessary to evaluate the plausibility of a classificatory hypothesis can be localized to each hypothesis in the context of the hierarchy. This can be done by a *hypothesis matching* component which evaluates any given hypothesis in the classification hierarchy by matching the data with expectations for the concept and which outputs a qualitative degree of confidence in the hypothesis (and the observations the hypothesis can explain). Thus the classifier, in conjunction with the hypothesis matchers for each of the concepts, can output plausible diagnostic hypotheses along with the data each hypothesis can explain.

The output of the classifier goes to an *abductive hypothesis assembly* component, which puts together a subset of these hypotheses as a composite hypothesis that best explains the data. This process must consider the interactions that occur among the causes that correspond to the hypotheses in order to ensure internal coherence among combinations of hypotheses. The knowledge concerning hypothesis interactions can be explicitly represented for each hypothesis, thus being consulted only if that hypothesis is included in the composite hypothesis. This process must also assemble a diagnosis which meets various criteria of parsimony, completeteness and plausibility. In the more complex uses of this architecture, the classification hierarchy may be asked to refine originally less plausible hypotheses if the explanatory power of the best composite hypothesis so far assembled is insufficient to cover all the observations that need explanation.

In many diagnostic problems the level of abstraction of the data which are available may be different from that required for the concept matcher, or additional inferences from available data may be needed to generate data that the diagnostic concept matcher can recognize as relevant. A necessary addition to this architecture is a database concept which uses domain knowledge to make the inferences and abstractions. In the proposed architecture, the hypothesis matcher can communicate with a system for *data retrieval/abstraction/inference*, whose task is a form of *knowledge-directed information passing* component which can convert data at a low level of abstraction into diagnostically significant data.

The important point is that each of the modules above is generic:

- Each is a strategy independent of diagnosis and can be used in a number of other high level tasks. The abductive assembler, e.g. can easily accept input from a plan recognizer so that it assembles a best explanation of various sightings in a battle situation. The data abstractor can be just as easily used by a therapy planner, and so on.
- Each strategy, as we shall see, uses characteristic knowledge and inference, making it possible to focus the problem solving effort in a manner appropriate for the task.

The functionality of Clancey's *heuristic classification*, consisting of the subtasks of *data abstraction*, *heuristic matching* and *refinement* can be achieved by three modules in our architecture: the hypothesis-matcher performs the heuristic matching task, the database component performs, among others, the data abstraction task, and the classifier performs the refinement task. The architecture has the additional capability of handling multiple malfunctions because of the abductive assembly component.

In sum the task of diagnosis can often be handled by the building blocks in the architecture diagramed in Fig. 1. Thus, if the appropriate knowledge is available, a compiled knowledge diagnostic system can be built using the CSRL, HYPER, PEIRCE and IDABLE tools.

Gomez and Chandrasekaran (1981) pointed out the importance of classification for diagnosis

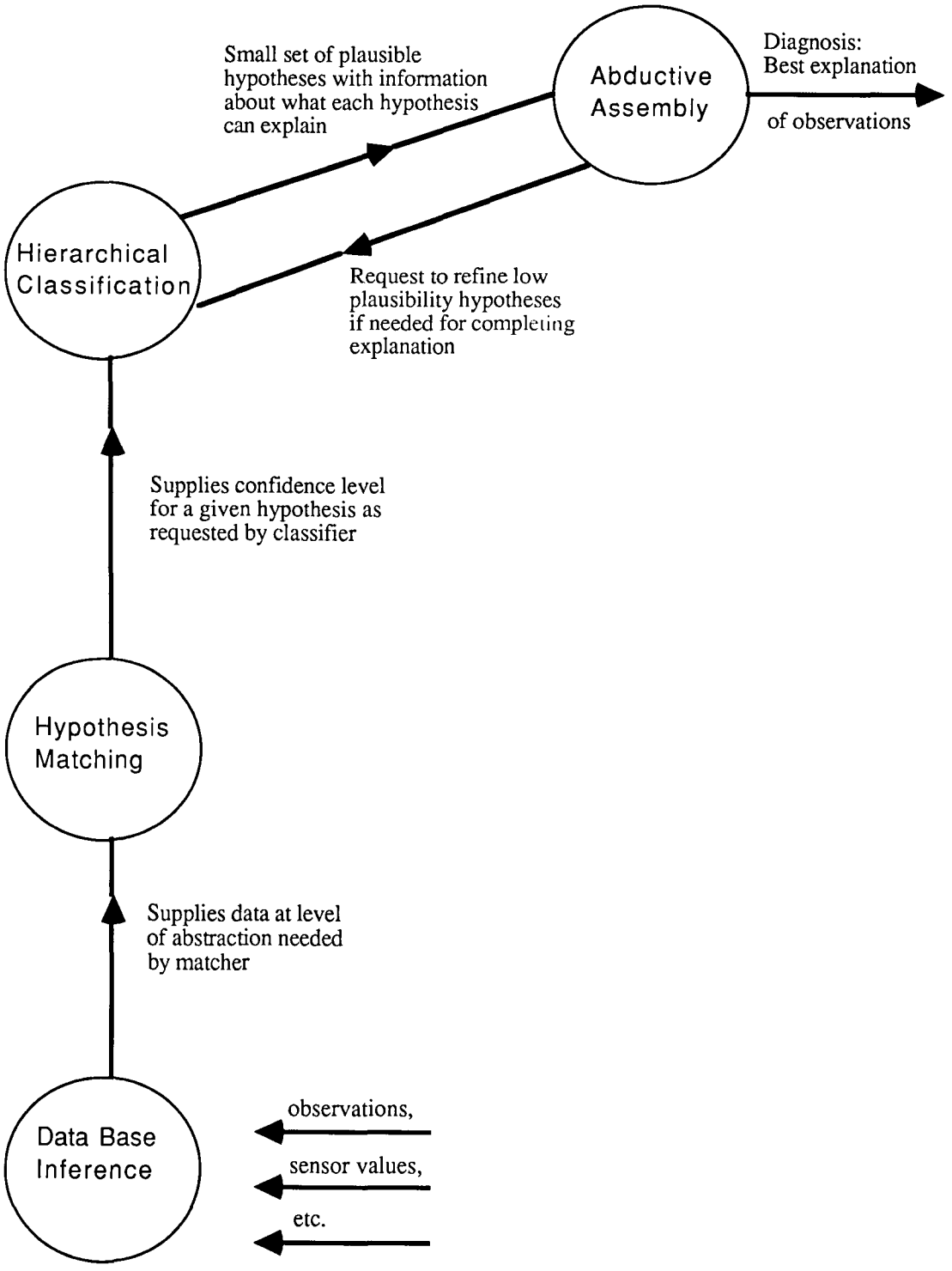

Figure 1 A generic task architecture for diagnosis with compiled knowledge

and Mittal (1982) described the architecture of MDX, which included all the components except abductive assembly. Josephson et al. (1987) added abductive assembly as part of a general architecture for abduction.

Describing how all the components of the diagnostic architecture can be built and integrated requires much more space than we have available. We describe in some detail the use of CSRL (the version to be described has HYPER embedded in it) for classification and hypothesis matching, and describe the assembly process in less detail. Josephson, Sticklen and Smith (1988) and Punch, Tanner and Josephson (1986) give further information on the tools IDABLE and PEIRCE.

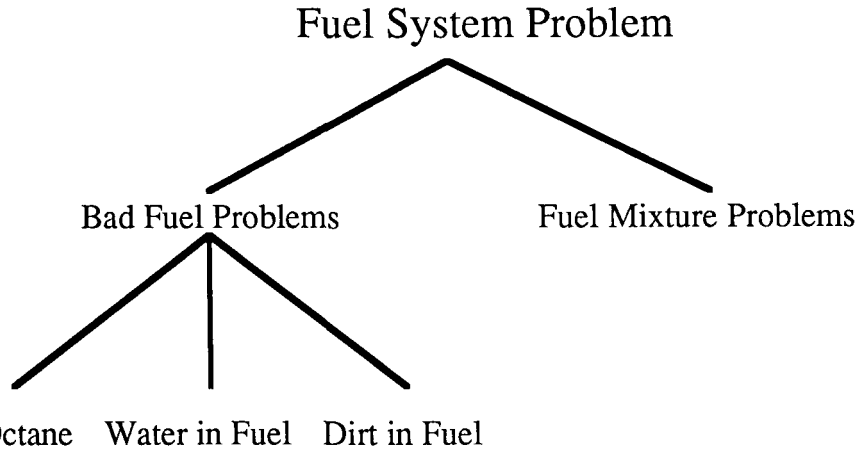

Figure 2 Fragment of fuel system classification tree. In this case, the hierarchy is largely of classes and subclasses of "causes". In other cases, the subclasses may be subfunctions, or physical parts

*3.c Classificatory problem solving and CSRL**

3.c.1 Classificatory hierarchies

Hierarchical classification (HC) is a particular method of performing the classification task. HC requires the availability of a classification hierarchy that organizes the classificatory hypotheses. Medical diagnosis, for example, uses disease hierarchies, and in many engineering domains, malfunction hierarchies are quite common.

In section 2, we gave a characterization of hierarchical classification. Fig. 2 illustrates a fragment of a tree from a hierarchical classification system for the diagnosis of fuel system malfunctions in a car engine.

Note that as the hierarchy is traversed from the top down, the categories (or in this particular case, hypotheses about the failure of the fuel system) become more specific. Thus the children of the hypothesis "bad fuel problems" can be broken into more specific hypotheses of "low octane", "water in fuel" and "dirt in fuel".

Each node in the hierarchy is responsible for calculating the "degree of fit" or *confidence value* of the hypotheses that the node represents. For example, the bad fuel problems node is responsible for determining if there is a bad fuel problem and the degree of confidence it has in that decision. Each node can be thought of as a "specialist" in determining if the hypothesis it represents is present. To create each specialist, knowledge must be provided to make this confidence value decision. The general idea is that each specialist specifies a list of *features* that are important in determining whether the hypothesis it represents is present and a list of *patterns* that map combinations of features to confidence values. In the fuel system problems specialist, such features might include gas mileage problems, poor performance, difficulty in starting the engine etc. One pattern might be that if all the features are present, then the fuel system problems hypothesis is likely.

3.c.2 The control strategy of hierarchical classification

Given that the knowledge of the system is organized as a set of specialists in a hierarchy, how can the hierarchy be *efficiently* traversed? This process is primarily accomplished through a type of hypothesis refinement called *establish-refine*. Simply put, a specialist that *establishes* its hypothesis (has a high confidence value), *refines* itself by activating its more detailed subspecialists. A hypothesis that is ruled out or rejected (has a low confidence value) is not refined, thus effectively pruning the sub-

* Much of the material in this section is from B. Chandrasekaran and William F. Punch III, "Hierarchical classification: its usefulness for diagnosis and sensor validation," invited talk at the Second AAAI/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program, March 9–11, 1987.

```
(SPECIALIST BadFuel
  (DECLARE (SUPERSPECIALIST FuelSystem)
      (SUBSPECIALIST LowOctane WaterInFuel
      DirtInFuel))
  (KGS ...)
  (MESSAGES ...))
```

Figure 3 Skeleton specialist for BadFuel. The code specifies the location of BadFuel in the hierarchy, points to the knowledge groups that contain information about how to establish or reject the concept and contains messages that specify control behaviour

tree below it. The reason for this becomes obvious when one thinks again of how the specialists are organized. The subhypotheses of fuel system problems, for example, are simply more detailed hypotheses. If there is no evidence for fuel system problems (it is ruled out), then there is no point in examining more detailed hypotheses about failures of the fuel system.

The process of establish-refine continues until no more refinements can take place. This can occur either by reaching the leaf level hypotheses of the hierarchy or by ruling out mid-hierarchy hypotheses.

3.c.3 CSRL, a language tool for hierarchical classification systems

CSRL (Conceptual Structure Representation Language) (Bylander and Mittal, 1986) is a language for writing hierarchical classification expert systems. The current version of CSRL is really a mixture of the shells of both hierarchical classification and hypothesis matching. A new version of the hypothesis matcher shell is available as HYPER. In this section, we will describe the older form of CSRL.

CSRL allows a knowledge engineer to do three things:

1 Create a hierarchy of malfunction hypotheses in a particular domain.
2 Encode the pattern matching knowledge for each hypothesis into a specialist.
3 Control the process of establish-refine problem solving.

Encoding the hierarchy of malfunctions

In CSRL, a hierarchical classification system is implemented by individually defining a specialist for each malfunction hypothesis. The super and subspecialist of a specialist are declared within the definition. Figure 3 is a skeleton of a specialist definition for the bad fuel node from Fig. 2. The declare section specifies its relationships to other specialists. The other sections of the specialist will be examined later.

Designing a classification hierarchy is an important part of building a CSRL expert system, but the exact structure of the final system is a pragmatic decision rather than a search for the perfect hierarchy. The main criterion for evaluating a classification hierarchy is whether enough evidence is normally available to make confident decisions. To decompose a specialist into its subspecialists, the simplest method is to ask the domain expert what subhypotheses should be considered next. The subhypotheses should be subtypes of the specialist's hypothesis, and will usually differ from one another based on a single attribute (e.g. location, cause).

For the diagnosis problem the criteria for forming classification hierarchies are discussed, with examples from the medical domain, by Bylander, Smith and Svirbely (1986), and in the engineering domain by Myers, Davis and Herman (1988). The hierarchy may mix function-subfunction and part-subpart views depending upon the way diagnostic reasoning actually works in the domain. Multiple hierarchies of the same domain, each from a different perspective, are also useful for some domains: the MDX-2 system of Sticklen (1987) uses such multiple hierarchies.

```
(RELEVANT TABLE
  (MATCH
    (ASKYNU? "Is the car slow to respond")
    (ASKYNU? "Does the car start hard")
    (AND (ASKYNU? "Do you hear knocking or
          pinging sounds")
      (ASKYNU? "Does the problem occur while
          accelerating"))
    WITH (IF T ? ?
       THEN-3
     ELSEIF ? T ?
       THEN-3
     ELSEIF ? ? T
       THEN 3
     ELSE 1)))
```

Figure 4 "Relevant" knowledge group of BadFuel. The ASK arguments are questions to the user, but they can also be queries to the database. The argument of WITH specify truth tables in the knowledge group

Encoding pattern-match knowledge

The knowledge groups in the KGS section (See Fig. 3) contain knowledge that matches the features of a specialist against the case data. Each knowledge group is used to determine a confidence value for some subset of features used by the specialist. As such, a knowledge group becomes an abstraction of evidence, representing an *evidential abstraction* of a particular set of features important to establishing the specialist. A knowledge group is implemented as a cluster of production rules that maps the values of a list of expressions (boolean and arithmetic operations on data, values of other knowledge groups) to some conclusion on a discrete, symbolic scale.

As an example, Fig. 4 is the relevant knowledge group of the BadFuel specialist mentioned above. It determines whether the symptoms of the automobile are consistent with bad fuel problems. The expression in the MATCH part queries the user (who acts as the database for this case) concerning whether the car is slow to respond, starts hard, has knocking or pinging sounds, or has the problem when accelerating. ASKYNU? is a LISP function which asks the user for a Y, N, or U (unknown) answer from the user, and translates the answer into T, F, or U, the values of CSRL's three-valued logic (note that any LISP function may be used here). The results of the MATCH expressions are then compared to a condition list in the WITH part of the knowledge group. For example, the first pattern "T ? ?" in the figure tests whether the first match expression (ASKYNU? "Is the car slow to respond") is true (the ? means doesn't matter). If so, then -3 becomes the value of the knowledge group*. Otherwise, subsequent patterns "? T ?" or "? ? T" are evaluated. The value of the knowledge group will be 1 if no rule matches. This knowledge group encodes the following matching knowledge:

If the car is slow to respond or if the car starts hard, then BadFuel is not relevant in this case. Otherwise, if there are knocking or pinging sounds and if the problem occurs while accelerating, then BadFuel is highly relevant. In all other cases, BadFuel is only mildly relevant.

Figure 5 is the summary knowledge group of BadFuel. Its MATCH expressions are the values of the relevant gas knowledge group (the latter queries the user about the temporal relationship between the onset of the problem and when gas was last bought). In this case, if the value of the relevant knowledge group is 3 and the value of the gas knowledge group is greater than or equal to 0, then the value of the summary knowledge group (and consequently the confidence value of BadFuel) is 3, indicating that a bad fuel problem is very likely.

This method of evidence combination allows the calculation of the confidence value to be hierarchically organized. That is, the results of any number of knowledge groups can be further abstracted by a knowledge group than can combine their values into a single confidence value.

* In this case, the values assigned are on a discrete scale from -3 to 3, -3 representing ruled-out and 3 representing confirmed.

```
(SUMMARY TABLE
  (MATCH RELEVANT gas
  WITH (IF 3 (GE 0)
       THEN 3
       ELSEIF 1 (GE 0)
       THEN 2
       ELSEIF ? (LT 0)
       THEN-3)))
```

Figure 5 "Summary" knowledge group of BadFuel

As mentioned earlier the above pattern matching knowledge and problem solving structure is a generic task that we have identified as hypothesis matching, and a separate shell called HYPER is available to capture just this functionality.

The mapping from data to confidence in a concept is a form of probabilistic mapping. The symbolic degrees of confidence are qualitative measures of subjective likelihood. However, the way data combine to produce a confidence for a higher level feature is *not* modelled by any normative calculus, be it Bayesian or one based on fuzzy sets, but *directly* obtained from the domain expertise localized to that particular context. These are important issues of how this view of handling uncertainty differs from the more traditional formal methods for which we refer the reader to (Chandrasekaran and Tanner, 1986; Bylander and Mittal, 1986).

Encoding of establish-refine strategy

The MESSAGES section of a specialist contains a list of message procedures which specify how the specialist will respond to different messages from its superspecialist. ESTABLISH and REFINE are the predefined messages in CSRL though others may be created by the user. The establish message procedure of a specialist determines the *confidence value* (i.e. the degree of fit) of the specialist's hypothesis. Figure 6 illustrates the establish message procedure of the BadFuel specialist. Relevant and summary are names of knowledge groups of BadFuel (see previous section). SELF is a keyword which refers to the name of the specialist. This procedure first tests the value of the relevant knowledge group. (If this knowledge group has not already been evaluated, it is automatically evaluated at this point.) If it is greater than or equal to 0, then BadFuel's confidence value is set to the value of the summary knowledge group, else it is set to the value of the relevant knowledge group. A value of $+2$ or $+3$ indicates that the specialist is established. In this case, the procedure corresponds to the following strategy.

First perform a preliminary check to make sure that BadFuel is a relevant hypothesis to hold. If it is not (the relevant knowledge group is less than 0), then set BadFuel's confidence value to the degree of relevance. Otherwise, perform more complicated reasoning (the summary knowledge group combines the values of other knowledge groups) to determine BadFuel's confidence value.

The refine message procedure determines what subspecialists should be invoked and the messages they are sent. Figure 7 shows a refine procedure which is a simplified version of the one that BadFuel uses. SUBSPECIALISTS is a keyword which refers to the subspecialists of the current specialist. The procedure calls each subspecialist with an ESTABLISH message. If the subspecialist establishes itself ($+?$ tests if the confidence value is $+2$ or $+3$), then it is sent a REFINE message.

```
(ESTABLISH (IF (GE relevant 0)
    THEN (SETCONFIDENCE self summary)
      else (SETCONFIDENCE self relevant)))
```

Figure 6 Establish procedure of BadFuel. If it has been evaluated not to be relevant, that sets the confidence value to be negative. If it is relevant then more complex matching is invoked to set the confidence value

```
(REFINE (FOR specialist IN subspecialists
        DO (CALL specialist WITH ESTABLISH)
          (IF (+? specialist)
            THEN (CALL specialist
              WITH REFINE))))
```

Figure 7 Example refine procedure. This specifies the control behaviour for exploring the successors of a classificatory hypothesis that has been established

3.c.4 The computational advantages of hierarchical classification

The major advantage of a hierarchical classification system is the organization of both the hierarchy of malfunctions and the knowledge groups within a specialist. This organization allows an efficient examination of the knowledge of the system based on need.

Consider again the hierarchy of Fig. 2. The problem solving begins by evaluation of the specialist Fuel System Problems. If that specialist establishes, then the two subspecialists, Bad Fuel Problems and Fuel Mixture Problems are invoked. If however, the Bad Fuel Specialist does not establish, then none of its subspecialists will be invoked. Thus, if a specialist rules out (i.e. does not establish), then none of the knowledge of the subspecialists need be run.

The same is true of the knowledge groups in the specialist. Only that knowledge necessary to confirm or deny the knowledge group is run. If a row of the knowledge group matches, then none of the subsequent rows are evaluated. Again, this results in running only the knowledge necessary for the problem at hand.

Compare this with other hierarchical approaches to diagnosis. The *fault tree* is a sequence of causally related events that leads to an observable symptom in the system. Given an initial malfunction, all possible causal results of the event are traced out, terminating with the symptoms that would be observed by a human diagnostician. When applied to an entire system, the result is a network of events that represent all the causal relationships of the system's constituent parts. While useful in design tasks, application of fault trees to diagnosis has a number of problems.

1 The combinatorial fan-out from an initial event can be very large. This makes the job of creating and traversing the network difficult. Compare this with the abstraction of hypotheses in hierarchical classification systems. Each node in the hierarchy represents a malfunction hypothesis that is listed in more detail through its subspecialists. If many subspecialists occur in the hierarchical decomposition of the domain, more levels of abstraction can be introduced to limit the fan-out. Such abstraction does not exist in fault tree representations.
2 Fault trees make no attempt to limit the number of nodes of the network that must be evaluated. Given a significant event, all possibilities are examined. However, hierarchical classifiers make use of the abstraction of malfunction hypotheses to limit the number of nodes that must be examined based on the data of the case.

The issues of control and communication in hierarchical classification can be more complex than our description in this paper. Gomez and Chandrasekaran (1981) describe the use of blackboards in exchanging information between different portions of the hierarchy. Sticklen, Chandrasekaran and Josephson (1985) describe the more complex control issues that need to be faced in some situations, and Sticklen (1987) describes the use of multiple hierarchies in classification.

3.d Hypothesis assembly in diagnosis

In typical diagnosis problems, the available data cannot always be explained by one malfunction hypothesis, but may require several hypotheses which must be combined in order to account for the observations. As the space of hypotheses grows in size, the problem of finding the best combination of hypotheses which apply to a particular situation becomes exponentially more difficult. Hierarchical classification can trim down the space of applicable hypotheses tremendously, yet it may still be

necessary to find a subset of the hierarchy's plausible candidates which gives the simplest and best diagnosis.

For example, suppose that the Auto-Mech hierarchy were extended to classify malfunctions for other car subsystems, including the braking mechanism. When classifying a case which has data such as "the car won't start when cold," "the engine runs roughly," "the brakes are hard to push," and "the car doesn't stop quickly," several diagnostic hypotheses may be found to be applicable, such as "water in fuel" and "loss of brake fluid", among others. In this case, the classification hierarchy has trimmed the set of all fuel and brake system diagnostic hypotheses down to a handful of highly relevant ones. However, the classification mechanism as such is incapable of determining whether this subset of relevant hypotheses accounts for all of the data. Furthermore, some of these hypotheses may be inconsistent or superfluous with respect to each other. Therefore, it is necessary to invoke a process of hypothesis assembly to complete the diagnosis by assembling a parsimonious subset of these hypotheses which gives the best explanation. In this case, the final subset of hypotheses could be "water in fuel" and "loss of brake fluid", since both are highly plausible, between the two of them they account for all the data, and neither is inconsistent or superfluous with regard to the other.

To carry this example further, it is possible that the hypothesis assembler may uncover relevant relationships which are only implicitly represented in the classification hierarchy. For example, in some models of cars the vacuum generated by airflow through the engine is diverted to assist the braking mechanism. Thus a punctured vacuum hose reduces the airflow through the carburettor, causing the engine to run roughly, and disabling the assistance to the brakes. When running the Auto-Mech hierarchy on the previous case, for an appropriate model of car, the following hypotheses may be found to be relevant: "engine vacuum hose punctured" and "brake vacuum assist inoperational". Both of these hypotheses need to be present in the classification hierarchy, because the first refines the "engine problem" hypothesis, whereas the second refines the "brake system problem" hypothesis. In this case, however, it is clear that the second problem is causally related to the first. Therefore, when the hypothesis assembler is putting together the best explanation for the data, it will uncover that "engine vacuum hose punctured" accounts for all the data that "brake vacuum assist inoperational" accounts for, and propose "engine vacuum hose punctured" as a one-hypothesis set which best explains the data.

Combining the process of hierarchical classification and hypothesis assembly in this way results in an efficient process for diagnosis. Classification without assembly cannot evaluate the final diagnosis in the case of multiple failures. Assembly without classification is an intractable problem in the general case. The issue of complexity in hypothesis assembly is particularly noticeable in the medical domain, where physiological and anatomical subsystems interact in highly complex ways. When both generic tasks are combined, however, their interaction reduces the complexity of the diagnostic problem while maintaining the ability to find the best explanation when there is one.

Josephson et al. (1987) is a good source of how the abductive assembly process works. Punch, Tanner and Josephson (1986) describe the tool PEIRCE, which is intended to build abductive assembly systems.

3.e Applications of the abductive architecture

A number of diagnostic systems have been built using the hierarchical classification approach provided by the CSRL tool. This section enumerates some of these applications and their domains.

It should be noted that of the following systems, Auto-Mech is strictly a pedagogical system, the nuclear power and chemical engineering systems are complex laboratory systems and Red, WELDEX and ROMAD are being developed to be used in real world situations.

3.e.1 Auto-Mech (Tanner and Bylander, 1985)

Auto-Mech is an expert system which diagnoses fuel problems in automobile engines. The purpose of the fuel system is to deliver a mixture of fuel and air to the air cylinders of the engine. It can be

divided into major subsystems (fuel delivery, air intake, carburettor, vacuum manifold) which correspond to initial hypotheses about fuel system faults.

Auto-Mech consists of 34 CSRL specialists in a hierarchy which varies from four to six levels deep. Before running, Auto-Mech collects some initial data from the user. This includes the major symptom that the user notices (such as stalling) and the situation when this occurs (e.g. accelerating and cold engine temperature). Any additional questions are asked while Auto-Mech's specialists are running. The diagnosis continues until the user is satisfied that the diagnosis is complete.

A major part of Auto-Mech's development was determining the assumptions that would be made about the design of the automobile engine and the data that the program would use. Different automobile engine designs have a significant effect on the hypotheses that are considered. A carburetted engine, for example, will have a different set of problems than a fuel injected engine (the former can have a broken carburettor). The data was assumed to come from commonly available resources. The variety of computer analysis information that is available to mechanics today was not considered in order to simplify building Auto-Mech.

3.e.2 Red (Smith et al., 1985)

Red is an expert system whose domain is red blood cell antibody identification. An everyday problem that a blood bank contends with is the selection of units of blood for transfusion during major surgery. The primary difficulty is that antibodies in the patient's blood may attack the transfused blood, rendering the new blood useless as well as presenting additional danger to the patient. Thus identifying the patient's antibodies and selecting blood which will not react with them is a critical task for nearly all red blood transfusions.

The Red expert system is composed of three major subsystems, one of which is implemented in CSRL. The non-CSRL subsystems are a data base which maintains and answers questions about reaction records (reactions of the patient's blood in selected blood samples under a variety of conditions), and an overview system, which assembles a composite hypothesis of the antibodies that would best explain the reaction record. (This assembly is itself a generic task called "abductive assembly" and PEIRCE can be used to build the assembly system). CSRL is used to implement specialists corresponding to the common blood antibodies and to each antibody subtype (different ways that the antibody can react).

The major function of the specialists is to rule out antibodies and their subtypes whenever possible, thus simplifying the job of the overview subsystem, and to assign confidence values, informing overview of which antibodies appear to be more plausible. The specialists query the data base for information about the lab test results and other patient information, and also tell the data base to perform certain operations on reaction records.

3.e.3 Complex mechanical systems

CSRL has been used in creating expert systems that do diagnosis of faults both in the domain of nuclear power plants and in the domain of chemical engineering.

The nuclear power industry must be very careful in the maintenance of running power plants since mistakes can prove costly not only in terms of power plant damage but also in terms of radiation leakage and broad environmental damage. Nuclear power plants are therefore heavily instrumented in many areas, so heavily in fact that it is difficult (if not impossible) for the operator to maintain an understanding of just what exactly is going on. The nuclear power plant expert system (Hashemi et al., 1986) is designed to take in large amounts of data and classify them into one of approximately 25 different failures. One advantage of the CSRL approach is that the operator can be informed of a high level view of the problem if no specific failure can be discovered.

The problems of the chemical engineering plant are similar, but it does have a number of differences. While safety is also of concern, there is also the problem of product quality in a chemical engineering plant. If a malfunction occurs that produces an unusable product, the operation must

be brought quickly back into line or large amounts of material will be wasted. The chemical engineering expert system (Shum et al., 1988) does diagnosis of a typical reactor producing a solid product as a result of the reaction of liquid product and oxygen. It consists of approximately 30 specialists that represent hypotheses about failures of the various physical parts of the plant. In addition to data that monitors the state of the reactor, these specialists also use data about product quality to make the confidence value decision.

3.e.3 Other real world uses of CSRL

CSRL is being used to develop two commercial systems by the knowledge-based systems group at the Battelle Columbus Institute. WELDEX and ROMAD are diagnostic systems for, respectively, detecting welding defects and evaluating machinery. A brief description of WELDEX follows.

WELDEX identifies possible defects in a weld from radiographic data of the weld. Industry standards and regulations require careful inspection of the entire weld and a very high level of quality control. Thus for industries which rely on welding technology, such as the gas pipeline industry, radiograph inspection is a tedious, time consuming, and expensive part of their operations.

This problem can be decomposed into two tasks: visual processing of the radiograph to extract relevant features of the weld, and mapping these visual features to the welding defects which give rise to them. WELDEX is intended to perform the second task. The current prototype consists of 25 CSRL specialists that are organized around different regions of the weld, taking advantage of the fact that each class of defects tends to occur in a particular region. The knowledge groups in these specialists concentrate on optical contrast, shape, size and location of the radiograph features. A customer version of WELDEX is currently being developed. Future work is anticipated on developing a visual processing system whose output would be processed by WELDEX, thus automating both parts of the radiograph inspection problem.

4 Routine design and DSPL*

4.a Subprocesses in design

Design is in general complex and, from the viewpoint of AI, a relatively poorly understood activity. For our purposes here, it is useful to think of design problem solving as having two sets of parts: those that "generate", i.e. propose designs or parts of designs and those that "test", i.e. analyse, critique and evaluate designs. Evaluating a design may involve problem solving behaviour such as qualitative simulation (e.g. to see if the projection from the wheel will rub against the body during rotation) or quantitative analyses (e.g. finite element methods to evaluate stress in a design component to see if the maximum stress is below the specifications), but these processes are not specific to design as a problem solving activity. On the other hand, the processes that participate in the "generate" portion are specific to design. In (Chandrasekaran, 1988) we have identified some of these generic processes:

- design problem decomposition,
- design plan instantiation and expansion,
- retrieval and modification of similar designs, and
- global satisfaction of constraint equations.

Among the above four processes, the first two are especially important for the discussion in this paper. In design by decomposition pre-stored domain knowledge is available which proposes possible decompositions of the design problem into specific subproblems, each hopefully of lesser complexity. In design by plan selection, instantiation and refinement, similarly a prestored *design plan* is available setting out a set of steps, some of them involving making some design commitments and

* Parts of this section are taken from a number of joint papers by the author with D. C. Brown.

others possibly involving calling other design plans, for solving the design problem at hand. The combination of decomposition and design plan instantiation and refinement can lead to quite complex problem solving. In design from past cases, both the decomposition and design plan are implicitly available for a previous design case, but they typically call for further criticism and modification.

4.b Classes of design

The framework suggests that design by decomposition (i.e. breaking problems into subproblems), by plan selection, and by plan synthesis (as a last resort) are the core processes in knowledge-based design. This suggests an informal classification of design problems based on the difficulty of these subtasks or processes.

4.b.1 Class 1 design

This is open-ended "creative" design. Goals are ill-specified, and there is no storehouse of effective decompositions, not to speak of design plans for subproblems. Even when decomposition knowledge is available, most of the effort is in searching for potentially useful problem decompositions. For each potential subproblem, further work has to be done in evaluating if a design plan can be constructed. This design problem is not routine. The average designer in industry will rarely, if ever, do Class 1 design: such design leads to an invention of new products.

4.b.2 Class 2 design

Class 2 design is characterized by powerful problem decompositions already available, but design plans for some of the component problems in need of *de novo* construction or substantial modification. Design of a new automobile, for example, does not involve new discoveries about decomposition: the structure of the automobile has been fixed for quite a long time. On the other hand, several of the components in it constantly undergo major technological changes, and routine methods of design for some of them may no longer be applicable.

Complexity of failure analysis will also take a problem away from routine design. Even if design plans are available, if the problem solver has to engage in very complex problem solving procedures in order to decide how to backtrack, the advantage of routine design is reduced. In short, whenever substantial modifications of design for components are called for, or when synthesis in the design plan space is especially complicated, we have a class 2 problem.

4.b.3 Class 3 design

This is relatively routine design: effective problem decompositions are known, compiled design plans for the component problems are known, and actions to take on failure of design solutions are also explicitly known. There is very little complex auxiliary problem solving needed. In spite of all this simplicity, the design task itself is not trivial: complex backtracking can still take place. The design task is still too complex for simple algorithmic solutions or table look up.

Class 3 problems are routine design problems, but still requiring knowledge-based problem solving. The ensuing sections of this paper deal with an approach to building knowledge-based systems for routine design problems of this type. The processes described here can work in conjunction with auxiliary problem solvers of various types, but we do not discuss such additional problem solvers here. The examples used all assume that the information to be provided by the auxiliary design processes, e.g. design criticism, verification, and subproblem constraint generation, are all available in a compiled manner.

4.b.4 A class 3 product

In a large number of industries, products are tailored to the installation site while retaining the same structure and general properties. For example, an air-cylinder intended for accurate and reliable

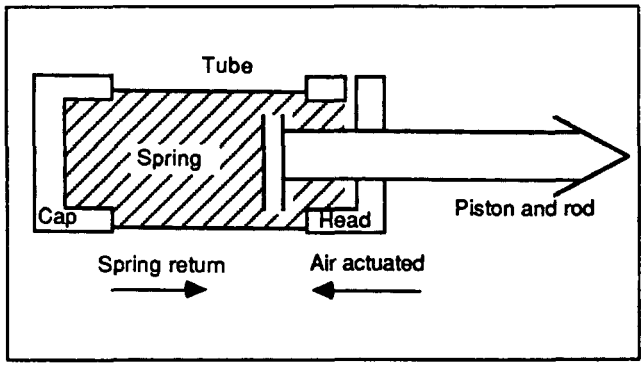

Figure 8 An air cylinder

backward and forward movement of some component will need to be redesigned for every new customer in order to take into account the particular space into which it must fit or the intended operating temperatures and pressures. This is a design task, but a relatively unrewarding one, as the designer knows at each stage of the design what the options are and in which order to select them. Note that this doesn't mean that the designer knows the complete sequence of steps in time (i.e. the trace) in advance, as the designer has to be in the problem solving situation before each decision can be made. There are just too many combinations of requirements and design situations to allow an algorithm to be written to do the job. This class of problems, while simple, is not by an means trivial: in fact, a typical class 3 problem involves more complex problem solving behaviour at the design level than say is implicit in R1 (McDermott, 1982).

DSPL is a language designed by D C Brown (1984) which captures the problem decomposition knowledge in the form of a design hierarchy of *design specialists* and the planning knowledge of each specialist is in the form of *design plans*. The specialists also have a certain amount of compiled *failure handling knowledge*, i.e. knowledge to help them recover when any of the chosen plans fail to accomplish their mission.

The approach taken is to consider design knowledge to be in the form of active cooperating design specialists. These specialists are organized in a hierarchy that reflects the human designer's conceptual organization of the design activity. Specialists use their own local design knowledge, but can also use the specialists directly below them in the hierarchy. This use is controlled by plans embedded in every specialist. Each specialist is responsible for some portion of the design, while its plans represent alternative methods for designing that portion. Communication between specialists is in the form of messages that flow up and down the hierarchy between the specialists and between their local agents (i.e. local design knowledge).* Messages flowing up may indicate failure or success.

The domain chosen was that of designing an air cylinder that was used by a local company in many pieces of equipment but which needed to be designed again each time due to changing requirements, such as the air pressure and the length of the stroke. A system called AIR-CYL has been written that does the design given a set of requirements.

The rest of this section will describe DSPL using the example of AIR-CYL (Brown and Chandrasekaran, 1986).

4.c DSPL: The design specialists and plans language

The air cylinder (AC) has about 15 parts, almost all of which are manufactured by the company according to their own designs, as their requirements are such that the components cannot be pur-

* We use the term "agent" to denote any chunk of knowledge that performs some action with well-defined functionality to it. The term "specialist" refers to those agents which correspond to the nodes in the part-subpart hierarchy of the object under design. Each specialist consists of further agents which take care of the subtasks of the specialists. In DSPL, these agents come in a number of different types.

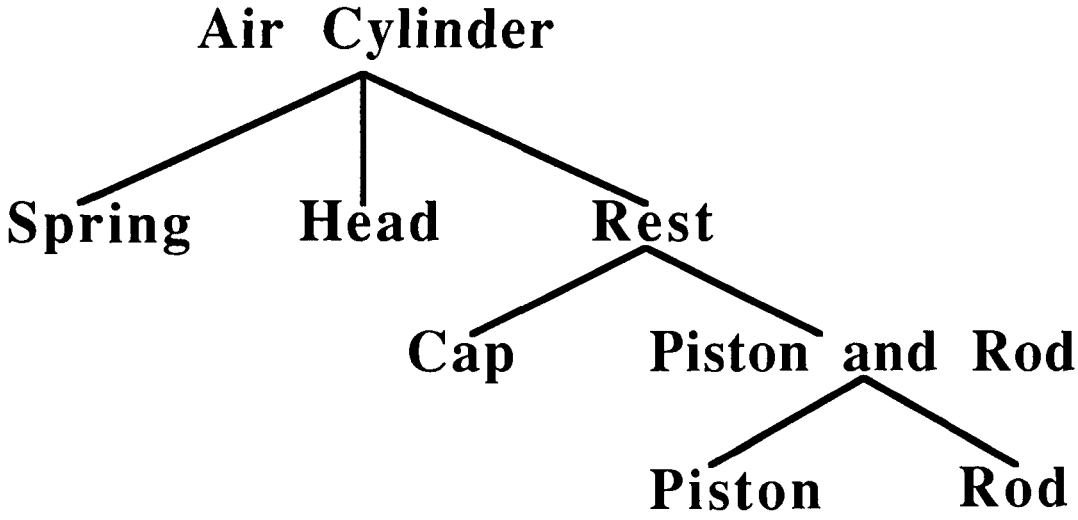

Figure 9 Specialist hierarchy for designing air cylinder. Design hierarchies may follow a function-subfunction or a part-subpart or a combination thereof as principles for organizing the hierarchy

chased. The AC is redesigned and changed slightly for applications with markedly different requirements. This characteristic makes it routine design in type. In operation, compressed air forces a piston back into a tube against a spring. Movement is limited by a bumper. The spring returns the piston, and the attached "load", to its original position when the air pressure drops.

The corresponding design specialist hierarchy is given in Fig. 9. The expert system for design of air cylinders is organized as a hierarchy of such specialists.

DSPL provides a way of writing declarations of Specialists, Plans, Tasks, Steps, Constraints, Failure Handlers, Redesigners, Sponsors and Selectors, allowing the user to specify the knowledge contained in them. In the following section we will address each of these declarations in turn.

To build a Design Expert System (DES), the user declares in DSPL all the agents (i.e. active design knowledge) required, and then allows the underlying system to link them together after some checking. Once formed, the DES can be invoked by requesting a design from the top-most specialist. The design then proceeds according to the specialist's plans. After a successful termination the design data base contains the completed design. If failure occurs reasons are given. The DSPL system provides the underlying problem solving control.

4.c.5 Design agents

Specialists

A specialist is a design agent that will attempt to design a section of the component. The specialists chosen, their responsibilities, and their hierarchical organization will reflect the mechanical designer's underlying conceptual structure of the problem domain. Exactly what each specialist's responsibilities are depends on where in the hierarchy it is placed. Higher specialists have more general responsibilities. The top-most specialist is responsible for the whole design. A specialist lower down in the hierarchy will be making detailed decisions. Each specialist has the ability to make design decisions about the part, parts or function in which it specializes. Those decisions are made in the context of previous design decisions made by other specialists. A specialist can do its piece of design by itself, or can utilize the services of other specialists below it in the hierarchy. We refer to this cooperative design activity of the specialists as design refinement.

Every specialist also has some local design knowledge expressed in the form of constraints. These will be used to decide on the suitability of incoming requirements and data, and on the ultimate success of the specialist itself (i.e. the constraints capture those major things that must be true of the specialist's design before it can be considered to be successfully completed). Other constraints,

```
(SPECIALIST
    (NAME      Head)
    (USED-BY AirCylinder)
    (USES      None)
    (DESIGN-PLANS      HeadDP1)
    (DESIGN-PLAN-SELECTOR   Headdpselector)
    (ROUGH-DESIGN-PLANS   HeadRDP1)
    (INITIAL-CONSTRAINTS   None)
    (FINAL-CONSTRAINTS   None)
```

Figure 10 Specialist "Head". The code specifies the location of the design specialist in the hierarchy, names the design plans that are used by it and the constraints that its parameters may need to satisfy

embedded in the specialists plans, are used to check the correctness of intermediate design decisions. Still more constraints are present in the design data base as general consistency checks. A typical specialist is shown in Fig. 10.

The selectors will be used to select from amongst the specialists plans. If no selector is specified a default selector will be used which selects plans in declaration order. Note that in this declaration, as with others, the order of the individual parts of the declaration may vary according to the user's wishes.

Plans

Each specialist has a collection of plans that may be selected depending on the situation, and it will follow the plan in order to achieve that part of the design for which it is responsible. A plan consists of a sequence of calls to specialists or tasks (see below), possibly with interspersed constraints. It represents one method for designing the section of the component represented by the specialist. The specialists below will refine the design independently, tasks produce further values themselves, constraints will check on the integrity of the decisions made, while the whole plan gives the specific sequence in which the agents may be invoked. Typically as one goes down in the hierarchy, the plans tend to become fewer in number and more straightforward. An example of this is shown Fig. 11.

The type of PLAN can be "Design" or "RoughDesign" depending on which phase of the design the knowledge applies to. The SPONSOR's job is to give an opinion to a selector about how suitable this plan is given the current state of the design. The BODY contains the details of the plan, and consists of an ordered list of plan items. In this example the plan consists entirely of tasks, with the exception of a constraint test and the last item which is a function provided by DSPL to print out the attributes and values of some part of the design.

```
(PLAN
    (NAME   HEADDP1)
    (TYPE   Design)
    (USED-BY Head)
    (SPONSOR HeadDP1Sponsor)
    (BODY HeadTubeSeat)
        MountingHoles
        Bearings
        SealAndWiper
        AirCavity
        AirInlet
          (CHECK-CONSTRAINT Air)
        TieRodHoles
(REPORT-ON Head)
```

Figure 11 Plan "HeadDP1". The design plan specifies the specialist that uses it, what criteria should be met for it to be chosen, and the plan steps

```
(STEP
    (NAME      AirCavityID)
    (USED-BY      AirCavity)
    (ATTRIBUTE-NAME HeadAirCavityID)
    (REDESIGNER   AirCavityIDRedesigner)
    (FAILURE-SUGGESTIONS
        (SUGGEST (DECREASE RodDiameter))
        (SUGGEST (DECREASE HeadBearingThickness))
        (SUGGEST (CHANGE HeadMaterial
                TO DECREASE MinThickness))
)
(COMMENT "FIND air cavity internal diam")

(BODY
    (KNOWN
    BearingThickness
        (KB-FETCH   "Head "HeadBearingThickness)
    RodDiameter
        (KB-FETCH   "Rod   "RodDiameter)
    HeadMaterial
        (KB-FETCH   "Head "HeadMaterial)
    MiniThickness
        (KB-FETCH   HeadMaterial "MinThickness)
)
    (DECISIONS
    MaxRodRadius (VALUE + (HALF RodDiameter))
    MaxBearingThickness
            (VALUE + BearingThickness)
    AirCavityRadius
            (+ MinThickness
                (+ MaxRodRadius
                  MaxBearingThickness))
    AirCavityID   (DOUBLE AirCavityRadius)
    REPLY (TEST-CONSTRAINT ACID)
    REPLY (KB-STORE
        "Head "HeadAirCavityID AirCavityID)
     )
    ))
```

Figure 12 Step "AirCavityID"

Steps, tasks, and constraints

We consider a step to be a design agent that can make one design decision given the current state of the design and taking into account any constraints. For example, one step would decide on the material for some subcomponent, while another would decide on its thickness.

A typical STEP in AIR-CYL is given in Fig. 12. A brief explanation of the role of the knowledge in the STEP is as follows. The sample STEP is USED-BY the AirCavity task. The ATTRIBUTE-NAME is the attribute for which this step is to design a value; that is, the internal diameter of the air cavity in the head of the air cylinder. If a failure occurs the REDESIGNER will attempt to recover from it by altering the value just selected for this step's attribute. The declaration REDESIGN NOT-POSSIBLE is also allowed. If the step itself fails the FAILURE-SUGGESTIONS get passed up to the controlling task in a failure message. The suggestions refer to attributes that might be the cause of the failure. Each item in the suggestion list is evaluated at failure time. This allows conditional suggestions such as (IF (. x y) THEN SUGGEST ...)). However, if the suggestion includes an expression, as in (DECREASE xyz BY (+ pqr 0.56)), then the SUGGEST function will arrange for the value to be computed. The actions DECREASE, INCREASE or CHANGE refer to attributes, such as RodDiameter. In the current system all attribute names must be unique.

The BODY of the step is divided in KNOWN and DECISION sections. The keywords

KNOWNS and DECISIONS will work just as well, and, in general, singular or plural keywords may be used as required. The KNOWN section obtains the values from the design data base by doing KB-FETCH. The KB-FETCH uses the component and attribute names. The single quote (i.e. ') is used to indicate that the name given is to be used directly without evaluation, as opposed to the use of a variable (e.g. HeadMaterial) that should be evaluated prior to use (e.g. giving the value "Aluminium").

The DECISIONS sections contains the design knowledge. It consists of variable-action pairs, where the action is evaluated and its value assigned to the variable. That variable may then be used in subsequent actions in the step. The variables set in the KNOWNS section may also be used. Arithmetic expressions use prefix operators. The function VALUE+ returns the value plus the positive tolerance of the value, and consequently provides the largest magnitude for that value. There are many other functions available.

There are two distinguished variable names. One is REPLY, the other COMMENT. A comment will act as a dummy assignment and will expect a string as the action. This is just a way of inserting a comment into the body of the step. A REPLY variable is used when there is no value produced by the action but a message showing success or failure is produced instead. The TEST-CONSTRAINT and the KB-STORE are two examples. The value calculated by the step is put into the design data base with a KB-STORE. It can produce a failure message if a constraint in the design data base fails. Any failure will stop the execution of the body and will cause the DECISION section to fail.

A task is a design agent which is expressed as a sequence of steps, possibly with interspersed constraints. It is responsible for handling the design of one logically, structurally, or functionally coherent section of the component; for example a seat for a seal, or a hole for a bolt.

A constraint is an agent that will test for a particular relationship between two or more attributes at some particular stage of the design. Constraints can occur at almost any place in the hierarchy. For example, a constraint might check that a hole for a bolt is not too small to be machinable given the material being used.

4.c.6 Other features

The main purpose of this exposition is not to give a complete description of DSPL, but to give a feel for the task-specific nature of the tool. A few other essential features of the language will now be briefly described.

Failure handling and redesign capability is an important requirement for anything other than relatively simple design problems. DSPL does not have failure analysis capabilities, but it can accept explicit knowledge about how to handle different kinds of failures during design. All design agents detect their own failure, are able to determine what went wrong (at least superficially), attempt to see if they can fix it locally, do so if they can, and report failure only if all attempts fail. Agents which have some control over other agents can use those agents in their attempt to correct the detected problem.

Each kind of agent can have different kinds of reasons for failing. For example, a step finds that a decision violates some constraint, a task discovers that a step's failure can't be mended locally, a plan can fail if it is discovered that it's not applicable to the situation to which it is being applied, while a specialist can fail if all of its plans fail.

For every kind of failure a message giving details is generated and passed back to the calling agent. The message includes, wherever possible, suggestions about what might be done to alleviate the problem. As there are usually many kinds of problems that can occur, an agent will first look at the message to decide what went on below. This is done by the failure handler associated with the agent. Much of the failure analysis is provided by the system, but for some cases, for example for constraint failures, the user (that is the person using the plan language to write a design system) has to supply some details. For some conditions immediate failure can be specified, for others an attempt to redesign might be made.

Knowledge about how to recover from failure can be coded as a *redesigner*. There appears to be a

difference between the "most reasonable choice" knowledge encoded in the step and the "most reasonable adjustment" knowledge encoded in the redesigner. The language provides a number of constructs for representing failure handling knowledge of the above types.

Sponsors and selectors: A *sponsor* is associated with a plan and it is responsible for estimating the suitability of a plan for a particular design situation. A selector takes the output of the sponsor and will decide whether or not to use the plan if it has been recommended as suitable. The sponsor is expected to provide a suitability value for the plan, and if it cannot, a "use of plan language" failure will occur.

The *selector* takes as input the names of the plans being considered and their suitabilities for use in the design situation as decided by their sponsors. It will pick a plan for the specialist to execute.

4.c.7 Use and extensions of DSPL

This section has discussed the idea of languages in which to express problem solving knowledge, and has presented the language DSPL for a class of design problem solving. DSPL embodies an underlying theory of routine design problem solving. Examples of Specialist, Plan, Task, Step, Constraint, Redesigner, Failure Handler, Sponsor, and Selector knowledge were given. Most of these examples were taken from AIR-CYL, an expert system to design an air cylinder.

DSPL has been used for the construction of MPA, a system for routine logistics planning (Chandrasekaran and Tanner, 1986), and in the construction of a design system in the domain of chemical engineering (Myers, Davis and Herman, 1988).

More work needs to be done to test the applicability of this language to other design problems and other domains. Extensions to the theory must be made in order to handle design activity which is not of the type where both problem solving and knowledge are known in advance. We feel that the identification of some of the types of design knowledge and their use is a substantial contribution towards understanding routine design activity.

5 The generic task toolset

5.a Important properties of the toolset

So far, we have outlined the generic task theory and also described two such generic task tools: CSRL and DSPL. Tools corresponding to other generic tasks are also in existence in varying degrees of completeness. These tools are currently available for the Interlisp/Loops environment in the Xerox 1100 series of Lisp machines. CSRL and DSPL are also available in versions that are compatible with the KEE development system of Intellicorp. CSRL is also available in Commonlisp.* Currently, a project is under way at our laboratory for the entire toolset to be made available in Commonlisp.

The integrated generic task toolset is extensible in the sense that more generic tools can be added as they are invented and additional problem solvers can be invoked as needed. The tools are intended to ensure the following advantages of the generic tasks, as described in (Bylander and Johnson, 1987).

- *Multiformity.* The more traditional architectures for the construction of knowledge-based systems emphasize the advantages of uniformity of representation and inference. However, in spite of the advantage of simplicity, we argued earlier that uniformity results in a level of abstraction problem. A uniform representation cannot capture important distinctions between different kinds of problems. A uniform inference engine does not provide different control structures for different kinds of problems.

  The generic task approach provides multiformity. Each generic task provides a different way

*CSRL is available as a supported product from Battelle Memorial Laboratories, Artificial Intelligence Group, in Columbus, Ohio, including versions in Commonlisp.

to organize and use knowledge. The knowledge engineer can choose which generic task is the best for performing a particular function, or can use different generic tasks for performing the same function. Different problems can use different generic tasks and different combinations of generic tasks.

- *Modularity.* A knowledge-based system can be designed by making a functional decomposition of its intended problem solving into several cooperating generic tasks, as illustrated in our discussion on diagnosis. Each generic task provides a way to decompose a particular function into its conceptual parts, e.g. the categories for hierarchical classification, and allows domain knowledge of other forms to be inserted into a generic task, e.g. evidence combination knowledge in hierarchical classification (Sticklen, 1987). Each generic task localizes the knowledge that is used to satisfy local goals.
- *Knowledge acquisition.* Each generic task is associated with its own knowledge acquisition strategy for building an efficient problem solver (Bylander, Smith and Svirbely, 1986). For example in hierarchical classification, the knowledge engineer needs to find out what specific categories should be contained in the classification hierarchy and what general categories provide the most leverage for the establish-refine strategy.
- *Explanation.* This approach directly helps in providing explanations of problem solving in expert systems in two important ways: how the data match local goals and how the control strategy operates (Chandrasekaran, Tanner and Josephson, 1988). Also, the control strategy of each generic task is specific enough for generating explanations of why the problem solver chose to evaluate or not to evaluate a piece of knowledge. This is because of the higher level of abstraction in which control is specified for generic tasks.
- *Exploiting interaction between knowledge and inference.* Rather than trying to separate knowledge from its use, each generic task specifically integrates a particular way of representing knowledge with a particular way of using knowledge. This allows the attention of the knowledge engineer to be focused on representing and organizing knowledge for performing problem solving.
- *Tractability.* Under reasonable assumptions, each generic task generally provides tractable problem solving (Allemang et al., 1987; Goel, Soundararajan and Chandrasekaran, 1987). (One major exception is abductive assembly, which can become intractable under certain conditions, making it hard then for humans and machines to perform the task.) The main reasons why they are tractable are that a problem can be decomposed into small, efficient units, and knowledge can be organized to take care of combinatorial interactions in advance.

It should be noted that these advantages are attained at the cost of generality. Each generic task is purposely constrained to perform a limited type of problem solving and requires the availability of appropriate domain knowledge.

5.b Integrating and combining GTs in an application

It is hard at this overview level to give enough details of how the tools are to be combined to put together complex applications. There are both significant theoretical issues of integration as well as practical issues of technology and implementation in this regard. The diagnosis example is a good example to discuss the practical issue of how the tools in the toolset can be combined for an application.

We need to make the following distinctions that will be helpful here. Each tool such as CSRL can be regarded a "shell" of a particular problem solving type. When the tool is invoked and knowledge and inference encoded using the knowledge primitives and message types, we have a *problem solver*, and different problem solvers built with the same shell can (and will typically) exist in an application. Each of the problem solvers is a specialist in two different senses: it specializes in a particular body of knowledge *and* in a type of problem solving, e.g. the *automech* (domain/concept) hierarchical classifier (type of problem solving) built out of CSRL, or the *BadFuel* (domain/concept) hypothesis matcher (type of problem solving) built out of HYPER, or the *AIR-CYL* (domain/concept)

hierarchical designer (type of problem solving). A given problem solver will typically need to access other problem solvers for information to continue its problem solving, e.g. the *BadFuel* matcher will need to know if various specific data items are present and for this it will need to access the data inference problem solver built from the tool IDABLE.

When the KBS is being built using the tools in the toolset, the knowledge engineer controls and directs the interaction among problem solvers by shaping and directing the messages appropriately. Without getting into details, the idea is simple: Each problem solver is characterized by (1) the kind of questions it can accept: e.g. the *Badfuel matcher* can accept messages that concern confidence values about the *Badfuel* concept and (2) the kind of questions that it can ask, e.g. the *Badfuel* concept can ask the data base problem solver for values of specific data attributes. In the current version of the toolset these decisions have to be explicitly made by the knowledge engineer, i.e. which problem solver has to send what types of queries to what other problem solvers has to be specified at the time the system is being built. The toolset itself is built on top of a substratum that is object- and message-oriented so that building additional generic tools within the framework is a straightforward thing to do. See (Josephson et al., 1988) for details on how the toolset is built up in this way.

For a complex application which involves portions which match the problem solving behaviours of the tools in the toolset, but which also has portions which require other methods of reasoning and represention not included within the current toolset, escaping to the object level or even the Lisp level (as in current implementations) to program AI or numerical techniques may be necessary and the toolset implementation supports this.

6 Concluding remarks

In the late 1970s, when we embarked on this line of research—characterized by an attempt to identify generic tasks and the forms knowledge and control required to perform them—the dominant paradigms in knowledge-based systems were rule and frame type architectures. While our work on use-specific architectures was evolving, dissatisfaction at the limited vocabulary of tasks that these architectures were offering was growing at other research centers. Clancey (1981) in particular noted the need for specifying the information processing involved by using a vocabulary of higher level tasks. Task-level architectures have been gathering momentum lately: McDermott and his coworkers (Marcus and McDermott, 1987) have built SALT, a shell for a class of design problems, where critiquing proposed designs by checking for constraint-violations is applicable. Clancey (1985) has proposed a shell called Heracles which incorporates the heuristic classification strategy for diagnosis. Bennett (1986) presents COAST, a shell for the design of configuration problem solving systems. All these approaches share the basic thesis of our own work, viz the need for task-specific analysis and architecture support for the task. However, there are some differences in assumptions and methodology in some cases that needs further discussion.

The following conceptual distinctions are useful:

- "Building blocks" out of which more complex problem solvers can be composed, such as the tasks in the theory presented in the paper.
- Explicit high level strategies which we want a system to follow, where the strategies are expressed in terms of some set of tasks. Clancey's Heuristic Classification is an example of this. McDermott's and Marcus' Salt system uses a strategy called "propose and refine" which is also of this type.
- Compound tasks, such as the form of diagnosis described earlier in the paper. An architecture for this compound task will bring with it its constituent generic tasks and also help in integrating the problem solving from the viewpoint of the overall task. In our laboratory, we are at work on building such diagnostic-level problem solving architectures for diagnosis of process engineering systems.
- Tasks which do not necessarily correspond to those that human experts do well, but nevertheless can be captured as appropriate combinations of knowledge and inference and a clear function can

be associated with them, e.g. constraint satisfaction schemes. Bennett's Coast system (Bennett, 1986) is an example of this.

Once we identify task-level architectures as the issue for highest leverage, then a number of immediate questions arise: what is the criterion by which a task is deemed to be not only generic but is appropriate for modularization as an architecture? How about an architecture for the generic task of "investment decision"? Diagnosis? Diagnosis of process control systems? Is uncertainty management a task for which it will be useful to have an architecture? Are we going to proliferate a chaos of architectures without any real hope of reuse? What are the possible relationships between these architectures? Which of these architectures can be built out of other architectures? I do not propose to answer all these questions here, but they seem to be the appropriate kinds of questions to ask when one moves away from the comfort of universal architectures and begins to work with different architectures for different problems.

At this stage in the development of these ideas, empirical investigation of different proposals from the viewpoint of usefulness, tractability and composability is the best strategy. From a practical viewpoint, any architecture that has a useful function and for which one can identify knowledge primitives and an inference method ought to be considered a valid candidate for experimentation. As the tools evolve, one may find that some of the architectures are further decomposable into equally useful, but more primitive, architectures; or that some of them do not represent particularly useful functionalities, and so on.

The generic tasks that are represented in our toolbox were specially chosen to be useful as technology for building diagnosis, planning and design systems with compiled expertise. For capturing intelligent problem solving in general, we will undoubtedly require many more such elementary strategies and ways of integrating them. For example, the problem solving activities in qualitative reasoning and device understanding, e.g. qualitative simulation, consolidation, and functional representation, all have well-defined information processing functions, specific knowledge representation primitives and inference methods. The candidates for generic information processing modules in our sense are indeed many.

What does all this mean for an architecture of intelligence?

I am led to a view of intelligence as an interacting collection of *functional units*, each of which solves an information processing problem by using knowledge in a certain form and corresponding inference methods that are appropriate. Each of these units defines an information processing faculty. I discuss elsewhere (Chandrasekaran, 1987) the view that these functional units share a computational property: they provide the agent with the means of transforming essentially intractable problems into versions which can be solved efficiently by using knowledge and inference in certain forms. For example, Goel, Soundararajan and Chandrasekaran (1987) show how classification problem solving solves applicable cases of diagnosis with low complexity, while diagnosis in general is of high complexity. Knowledge is indeed power, but how it acquires its power is a far subtler story than the first generation knowledge-based systems made it appear.

This view generates its own research agenda: As a theory, the generic tasks idea has quite a bit of work ahead of it in terms of a coherent story about how the tasks come together, are integrated and how more complex tasks such as planning come about from more elementary ones. How complex inference methods develop from simpler ones and how learning shapes these functional modules are issues to be investigated.

The relationship of task-specific architectures such as the GT ideas in this paper to, on one hand, more general architectures, such as SOAR (Laird, Newell and Rosenbloom, 1987), and on the other to "weak methods" is an intriguing one. My view is that from the perspective of modelling cognitive behaviour, a GT-level analysis provides two closely related ideas which give additional content to phenomena at the SOAR architecture level. On the one hand, the GT theory provides a vocabulary of goals that a SOAR-like system may have. On the other hand, this vocabulary of goals also provides a means of indexing and organizing knowledge in long-term memory such that when SOAR is pursuing a problem solving goal, appropriate chunks of knowledge and control behaviour are

placed in short-term memory for SOAR to behave like a GT problem solver. In this sense a SOAR-like architecture, based as it is on goal achievement and universal subgoaling, provides an attractive substratum on which to implement future GT systems. In turn, the SOAR-level architecture can give graceful behaviour under conditions that do not match the highly compiled nature of GT-type problem solving.

Acknowledgements

I am indebted to my colleagues with whom I have written a number of papers over the years on the generic task approach. In the preparation of this paper I have used excerpts from such papers. In particular the papers by the author and W Punch, the author and D C Brown, the author, T C Bylander and John Josephson have been used. I acknowledge the assistance of Richard Fox and Vibhu Mittal in the preparation of this version of the paper. Matt Dejongh helped with some sections of this paper. The comments of John Fox, the editor of *Knowledge Engineering Review*, have helped improve the paper substantially. I also gratefully acknowledge the support of the Defense Advanced Research Projects Agency, RADC Contract F30602-85-C-0010, and the Air Force Office of Scientific Research, grant 87-0090.

References

Allemang, Dean, Tanner, Michael C, Bylander, Tom and Josephson, John R, 1987. "On the computational complexity of hypothesis assembly" *Proceedings of IJCAI-87*, Milan, Italy, August, 1987.

Bennett, James, 1986. "COAST: A task-specific tool for reasoning about configurations" In: *Proc AAAI Workshop on High-Level tools*, AAAI, Shawnee Park, Ohio, 1986.

Brown, D C, 1984. *Expert Systems for Design Problem-Solving using Design Refinement with Plan Selection and Redesign*, PhD thesis, The Ohio State University, August.

Brown, David C and Chandrasekaran, B, 1986. "Knowledge and control for a mechanical design expert system" *IEEE Computer* **19**, pp. 92–101, July.

Buchanan, B, Sutherland, G and Feigenbaum, E A, 1969. "Heuristic DENDRAL: A program for generating explanatory hypotheses" *Organic Chemistry*.

Bylander, T and Mittal S, 1986. "CSRL: A language for classificatory problem solving and uncertainty handling" *AI Magazine* 7(3), pp. 66–77.

Bylander, T, Smith, J and Svirbely, J, 1986. "Qualitative representation of behavior in the medical domain" In: *Proceedings of The Fifth Conference on Medical Informatics*, pp. 7–11. Conference on Medical Informatics, Washington, DC, October 26–30, 1986.

Bylander, Tom and Johnson, Todd R, 1987. *Structured Matching* OSU CIS LAIR Technical Report.

Chandrasekaran, B, 1983. "Towards a taxonomy of problem solving types" *AI Magazine* pp. 9–17, Winter/Spring.

Chandrasekaran, B, 1986. "Generic tasks in knowledge-based reasoning: high level building blocks for expert system design" *IEEE Expert* 1(3), pp. 23–30.

Chandrasekaran, B and Tanner, M, 1986. "Uncertainty handling in expert systems: Uniform vs. task-specific formalisms" *Uncertainty in Artificial Intelligence*, Kanal, L N and Lemmer, J (Eds). North Holland Publishing Company, pp. 35–46.

Chandrasekaran, 1987. "Towards a functional architecture for intelligence based on generic information processing tasks" In: *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 1183–1192, Milan, Italy, August, 1987.

Chandrasekaran, B, Smith, J and Sticklen, J, 1987. *Deep Models and Their Relation to Diagnosis* Technical report. Invited paper, Toyoba Foundation Symposium on Artificial Intelligence in Medicine, Tokyo, Japan, August 1986, available as technical report from Laboratory of AI Research, Ohio State University.

Chandrasekaran, B, 1987. "What kind of information processing is intelligence? A perspective on AI paradigms and a proposal". This paper will appear in *Source Book on the Foundations of AI* Partridge and Wilks (Eds.), Cambridge University Press.

Chandrasekaran, B, Tanner M C and Josephson, J R, 1988. "Explanation: the role of control strategies and deep models" In: *Expert Systems: The User Interface*, James Hendler (Ed.), Ablex Publishing Corporation, Norwood, New Jersey 07648, pp. 219–247.

Chandrasekaran, B, 1988. *Design: An Information Processing Level Analysis* technical report. The Ohio State University, Laboratory for AI Research, Columbus, Ohio 43210.

Clancey, W J, 1981. "NEOMYCIN: Reconfiguring a rule-based expert system for application to teaching" In: *Proc. Seventh International Joint Conference on Artificial Intelligence*, pp. 829–836. IJCAI, Vancouver.

Clancey, W J, 1985. "Heuristic classification" *Artificial Intelligence* **27**(3), pp. 289–350.

Duda, R O, Gaschnig, J G and Hart, P E, 1980. "Model design in the prospector consultant system for mineral exploration" In: *Expert System in the Microelectronic Age*, Michie, D (Ed.), pp. 153–167, Edinburgh University Press.

Friedland, P, 1979. *Knowledge-based Experiment Design in Molecular Genetics.* PhD thesis, Stanford University, Computer Science Department.

Goel, A, Soundararajan, N and Chandrasekaran, B, 1987. "Complexity in Classificatory Reasoning" In: *Proc. National Conference on Artificial Intelligence*, pp. 421–425. Seattle, Washington, July 13–18, 1987.

Gomez, F and Chandrasekaran, B, 1981. "Knowledge organization and distribution for medical diagnosis" *IEEE Transactions on Systems, Man, and Cybernetics*, **SMC-11**(1), pp. 34–42, January.

Hashemi, S, Hajek, B K, Miller, D W, Chandrasekaran, B and Josephson, J R, 1986. "Expert systems application to plant diagnosis and sensor data validation" In: *Proceedings of the Sixth Power Plant Dynamics Control and Testing Symposium*, Knoxville, Tennessee, April, 1986.

Johnson, T, 1986. "HYPER: The hypothesis matcher tool" In: *Proceedings of Expert Systems Workshop*, pp. 122–126. Defense Advanced Research Projects Agency, Pacific Grove, CA, April 16–18.

Johnson, K, Sticklen, J and Smith, J W, 1988. "IDABLE—application of an intelligent data base to medical systems" In: *Proceedings of the AAAI Spring Artificial Intelligence in Medicine Symposium*, pp. 43–44. American Association for Artificial Intelligence, Stanford University, March 22–24.

Josephson, J R, Smetters, D, Welch, A K, Fox, R, Flores, G and Lyndes, D, 1988. *Generic Task Toolset DRACO Release—Beta Test Including RA* Technical report. The Ohio State University, Computer & Information Science Department, Laboratory for Artificial Intelligence Research, February.

Josephson, J R, Chandrasekaran, B, Smith, J W and Tanner, M C, 1987. "A mechanism for forming composite explanatory hypotheses" *IEEE Trans. on Systems, Man and Cybernetics* pp. 445–454.

Marcus, Sandra and McDermott, John, 1987. *SALT: A Knowledge Acquisition Tool for Propose-and-Revise Systems* Technical report. Department of Computer Science, Carnegie-Melton University, Pittsburgh, PA.

McDermott, J, 1982. "R1: A rule-based configurer of computer systems" *Artificial Intelligence* **19**(1), pp. 39–88.

Miller, R A, Pople, H E and Myers, J D, 1984. "Internist-1, an experimental computer-based diagnostic consultant for general internal medicine" *Readings in Medical Artificial Intelligence*, Addison-Wesley Publishing, pp. 190–209.

Mittal, S, 1980. *Design of A Distributed Medical Diagnosis and Data Base System* PhD thesis. The Ohio State University.

Myers, D R, Davis, J F and Herman, D, 1988. "A task oriented approach to knowledge-based systems for process engineering design" *Computers and Chemical Engineering, Special Issue on AI in Chemical Engineering Research and Development*, August.

Laird, J E, Newell, A and Rosenbloom, P S, 1987. "SOAR: An architecture for general intelligence" *Artificial Intelligence* **33**, pp. 1–64.

Punch, W F, Tanner, M C and Josephson, J J, 1986. "Design Considerations for PEIRCE, a high-level language for hypothesis assembly" *Expert Systems in Government Symposium* pp. 279–281, October.

Shortliffe, E H, 1976. *Computer-based Medical Consultations: MYCIN* Elsevier/North-Holland Inc.

Shum, S K, Davis, J F, Punch III, W F and Chandrasekaran, B, 1988. "An expert system approach for malfunction diagnosis in chemical plants" *Computers and Chemical Engineering* **12**(1), pp. 27–36.

Smith, J W, Svirbely, J R, Evans, C A, Straum, P, Josephson, J R and Tanner, M C, 1985. "RED: A red-cell antibody identification expert module" *Journal of Medical Systems* **9**(3), pp. 121–138.

Sticklen, J, Chandrasekaran, B and Josephson, J, 1985. "Control issues in classificatory diagnosis" In: *Proceedings of The 9th International Joint Conference on Artificial Intelligence.* International Joint Conference on Artificial Intelligence, University of California, Los Angeles, CA, August, pp. 18–24.

Sticklen, Jon, 1987. "MDX2: An integrated medical diagnostic system" PhD. Dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio 43210.

Tanner, M and Bylander, T, 1985. "Application of the CSRL language to the design of expert diagnosis systems: The auto-mech experience" *Artificial Intelligence in Maintenance.* Noyes Publications, Park Ridge, N.J.