Combining Multiple Knowledge Sources in an Integrated Intelligent System

David M. Steier, Richard L. Lewis, and Jill Fain Lehman, Carnegie Mellon University
Anna L. Zacherl, Allegheny College

USING A STRATIFIED APPROACH TO SYSTEM DESIGN IN SOAR, WE CAN INTEGRATE MULTIPLE KNOWLEDGE SOURCES, TRADITIONALLY A BARRIER TO THE RAPID CONSTRUCTION OF KNOWLEDGE-BASED SYSTEMS. WE HAVE IMPLEMENTED SUCH SYSTEMS IN NATURAL-LANGUAGE COMPREHENSION, PRODUCTION SCHEDULING, AND ALGORITHM DESIGN.

DESIGNING A KNOWLEDGE-based system with a single knowledge source is certainly easier than integrating multiple knowledge sources. However, as knowledge-based systems become increasingly autonomous, they will be required to exploit many knowledge sources to solve increasingly difficult problems. The knowledge sources may use representations that differ from each other and from the representation used by the knowledge-based system itself. Some knowledge-engineering approaches rely on programmers to convert manually all the necessary knowledge into a single representation at design time, but the resulting bottleneck could become overwhelming. Therefore, researchers need solutions to knowledge integration that do not rely on human intervention. Using a stratified approach to system design in Soar, we can integrate multiple knowledge sources and overcome this traditional barrier to the rapid construction of knowledge-based systems.

Our characterization of the problem relies on the concept of the *knowledge level*. Following Allen Newell's terminology,[1] the knowledge level is a way of specifying intelligent systems as agents with goals, knowledge, perceptions, and actions. Agents follow the *principle of rationality*; that is, an ideal knowledge-level agent uses its knowledge to select actions that it believes will result in the attainment of its goals. In a physically realizable rational agent, knowledge is embodied in *sources*, which are the symbol structures representing the knowledge together with their associated access methods. The knowledge-level description is transformed into a set of *problem spaces* that provide access to the necessary knowledge. These problem spaces are compiled, and that output is a set of productions that is input to the Soar interpreter at the symbol (or program) level.

In terms of the Soar system levels we are using, the fact that knowledge from various sources cannot be used defines a failure at the knowledge level; that is, the agent acts as if it did not possess that knowledge and selects the wrong action. Further analysis and resolution of such failures must occur at lower system levels, where representations and access methods are specified. The type of integration required at these levels depends on the degree of search needed to perform the integration.

The search, in turn, depends on the type of interaction between knowledge sources. At one end of the continuum the knowledge sources are independent, and little or no effort is required to integrate them. A possible integration method in this case is assembly, in which the system simply aggregates component results from two or more sources to yield the directly usable

0885/9000/93/0600-0035 $3.00 © 1993 IEEE

knowledge. For example, a move generator for a game might operate by generating "from" positions using one knowledge source and "to" positions using another knowledge source. In most games, this would be impractical, in that the choice of a "from" position constrains the choice of a "to" position, and vice versa. The component knowledge sources are coupled, so that the integration process must become more complex.

This leads to the other end of the continuum, where knowledge sources are interdependent and integration requires a great deal of search. One method for integration in this case is generate-and-test: One knowledge source generates candidate solutions, and the other filters the candidates to yield directly usable solutions. Chess machines take advantage of special-purpose move generators, sometimes embedded in hardware, to operate in this way. However, for truly intelligent agents that operate in a range of task environments, any mechanism for knowledge integration should be recursive: Problem solving in one component knowledge source may also encounter a lack of immediately available knowledge, and thus require further knowledge integration to retrieve the knowledge it needs, and so on.

One way to gradually overcome the problem of repeatedly performing the same task is to store knowledge once it has been integrated, so that the new knowledge will be directly available in future situations. Soar's problem-space computational model and its symbol-level production system ease both knowledge source integration and learning (although they do not yet solve the problem on their own).

Basic Soar principles

Soar's support for knowledge integration derives from the architecture's basic principles: a *recognition memory* to store and retrieve immediately available knowledge, the use of *multiple problem spaces* for formulating tasks, *universal subgoaling* to transfer control between problem spaces, and *chunking* to preserve knowledge acquired in subgoals for future use.

When all knowledge is directly available, Soar solves a problem by searching a single problem space (a set of states and operators for solving a problem). Soar selects the problem space and initial state to formulate the problem, and then cycles between operator proposal, selection, and application to produce new states until it reaches a desired state representing its goal. Directly available knowledge is defined as the knowledge that can be retrieved from recognition memory using a simple pattern match. The information retrieved takes the form of preferences for changes to the problem-solving context, including what action to perform next.

If there is insufficient directly available knowledge for Soar to select the next action (for example, if no operators have been proposed, or no judgment can be made about the relative desirability of multiple proposed operators), Soar is at an *impasse*. Universal subgoaling then comes into play, and a subgoal is generated to resolve the impasse. Soar works on this subgoal as it did the higher level goal that was the source of the impasse, that is, by selecting problem spaces, states, and operators. In a subgoal, however, the impasse that generated it dictates when problem solving should be terminated: when sufficient knowledge has been acquired to resolve the impasse. Put another way, problem solving in the subgoal stops when the problem-solving context has been elaborated with sufficient structure that formerly unavailable knowledge has become directly available. The newly available knowledge allows problem solving in pursuit of the higher level goal to proceed.

FOR TRULY INTELLIGENT AGENTS OPERATING IN A RANGE OF TASK ENVIRONMENTS, ANY MECHANISM FOR KNOWLEDGE INTEGRATION SHOULD BE RECURSIVE.

The chunking mechanism compiles the retrieval process for more efficient application in future similar situations. Processing in a subgoal can be viewed as a chain of memory accesses, starting from immediately recognizable patterns in the context above the subgoal, and resulting in retrieved information used in the service of that subgoal. Chunking traces back through the chain of memory accesses and builds new pattern-information mappings that collapse the chain. Thus chunking makes directly available those problem-solving results formerly accessible only by generating a subgoal. (For more details, see Jack Smith and Todd Johnson's article on pages 15-25 of this issue, and elsewhere.[2])

Thus in Soar, for any impasse, a variety of problem spaces accessing different knowledge sources can be consulted to resolve an impasse. Each of these spaces can have a different representation and set of operations, but each must deliver results that can resolve the impasse for which it was evoked. The design of such problem spaces sometimes requires significant effort. However, we can take advantage of a number of architectural features to integrate results, particularly for the assembly method. The temporary working memory permits the accumulation of preferences for actions and changes to the problem-solving context. This accumulation can occur over several memory accesses or over several operator applications. Since a subgoal and corresponding impasse indicate that new knowledge is needed, the context of a subgoal can provide guidance in structuring the knowledge. Most significant, however, is that the chunking mechanism reduces the integration problem over time. This is due to the increased amount of directly available knowledge as chunks are added. Because chunking caches the results of all of Soar's problem solving, it makes operation within each knowledge source and within the integration process more efficient, as well as reducing the need to convert between different knowledge representations. The chunks are actually compiled versions of the knowledge used in the original problem solving; as with all compiled code, chunks need not be easily read or explained by people as long as the computer system can apply them at the appropriate time.

Our use of Soar to integrate multiple knowledge sources differs substantially from other work in knowledge engineering. The goal of the work on knowledge sharing,[3] for example, is to develop an infrastructure for composing knowledge bases from reusable components. The infrastructure takes the form of interfaces

and knowledge representation languages mostly developed independently of the problem-solving architecture and the methods that will use the knowledge. Method-based knowledge acquisition frameworks, such as generic tasks,[4] address integration in some respects, but do not provide architectural support for the necessary processes. In both cases, the concept of an autonomous agent performing integration and the use of learning to capture the results of integration are absent.

The following discussion of Soar-based implementations depends on the following formalism: An agent performs some problem-solving function F (for example, selecting an operator). If there is no integration problem, the necessary knowledge K is delivered by knowledge source KS immediately and in a form directly usable for performing F. An integration problem arises when K must be constructed by consulting other knowledge sources $KS_1 ... KS_n$, one or more of which differ from KS. To complicate matters, the knowledge of how to integrate knowledge from various knowledge sources might be available only by consulting yet another knowledge source. The search for knowledge bottoms out only when the process yields directly usable knowledge.

The three implemented systems perform different tasks: natural-language comprehension, production scheduling, and algorithm design. They demonstrate that architectural mechanisms can play a key role in constructing systems to perform difficult knowledge-intensive tasks.

Language comprehension

Language comprehension is the process by which knowledge is used to map an utterance to a meaning. As comprehension proceeds word by word through the utterance, multiple knowledge sources play a role in finding the correct mapping. Yet, *when* a knowledge source is used during comprehension is every bit as important as *whether* it is used. Language is replete with local ambiguities, that is, choice points where multiple interpretations are possible. The number of interpretations depends on what knowledge is available to eliminate globally incorrect choices. Does the word "saw," for example, refer to a cutting tool or visual activity? When the word is encountered in the context "John saw," lexical knowledge produces multiple meanings, but syntax can help disambiguate if syntactic information is immediately available. Lexical and syntactic information alone, however, cannot choose the correct interpretation for "John saw the man with the gun." Did John use the gun to see the man (as in "John saw the man with the telescope") or was the man holding the gun? Only semantics can decide. Even the current context may be crucial to arrive at a unique meaning. If "John saw the man on the hill with the telescope," does John have the telescope, or does the man (or does the hill)? Previously established relationships in the situation (John is holding a telescope) may dictate how the prepositional phrases should be attached.

TO GUARANTEE A CORRECT MAPPING WE MIGHT NEED ALL THE KNOWLEDGE SOURCES, BUT TO GUARANTEE A MINIMAL SEARCH SPACE WE MUST BRING EACH TYPE OF KNOWLEDGE TO BEAR AS SOON AS IT IS RELEVANT.

In each of these examples, if the disambiguating knowledge is not immediately available, multiple interpretations must be maintained until that knowledge becomes available. The more local ambiguities there are in an utterance, the more interpretations must be carried along. Since local ambiguity is pervasive, language understanding is easily recognized as a search problem. Reducing search is necessary; without multiple knowledge sources, even common constructions such as prepositional phrases can lead to an exponential number of interpretations. The key to reducing search is integration of knowledge sources. To guarantee the correct mapping we might need all the knowledge sources, but to guarantee a minimal search space we must be able to bring each type of knowledge to bear as soon as it is relevant.

The need to integrate knowledge efficiently has been a central research issue in building working comprehension systems. Most integration attempts have proposed uniform representations designed to capture certain kinds of precompiled syntactic and semantic knowledge. Examples include semantic grammars, word experts, domain-specific syntactic grammars, and unification formalisms.[5] While such approaches have succeeded to varying degrees in achieving integration, it is not clear how easily they can be extended to handle additional knowledge sources. Systems that preserve the independence of multiple knowledge sources, such as blackboard architectures, may gain some advantage in extensibility, but face serious issues in controlling the efficient application of modules. For example, the problem of scheduling modules is the primary concern in principle-based parsing systems,[6] which use recent linguistic theories covering a wide range of syntactic constructions via a relatively small number of interacting independent knowledge sources (principles). (Here, independent knowledge sources are all syntactic, but the point remains the same.)

Knowledge integration. NL-Soar[7] is a comprehension system developed in the Soar architecture. Following the formalism introduced earlier, we take F as the function mapping a word in context to (partial) knowledge about the meaning of the utterance. The meaning is represented in the situation model, which contains the objects and relations that the utterance is about, and sets the context for future utterances. Functional syntactic relations among words (such as subject or indirect object) aid in the mapping process; they are represented in a separate structure, the utterance model. The mapping function F is carried out by Comprehend operators, shown in the top problem space in Figure 1 (the circled lowercase letters and numbers will be used in a forthcoming example). A Comprehend operator applies to each word in a sentence. If the knowledge to apply these operators is not directly available, they are implemented in the Construct space, which constructs the utterance and situation models. The Constraint space checks independent syntactic and semantic constraints on model-constructor operators.

The following list describes how NL-Soar represents and integrates the relevant knowledge sources and how the system uses each type of knowledge:

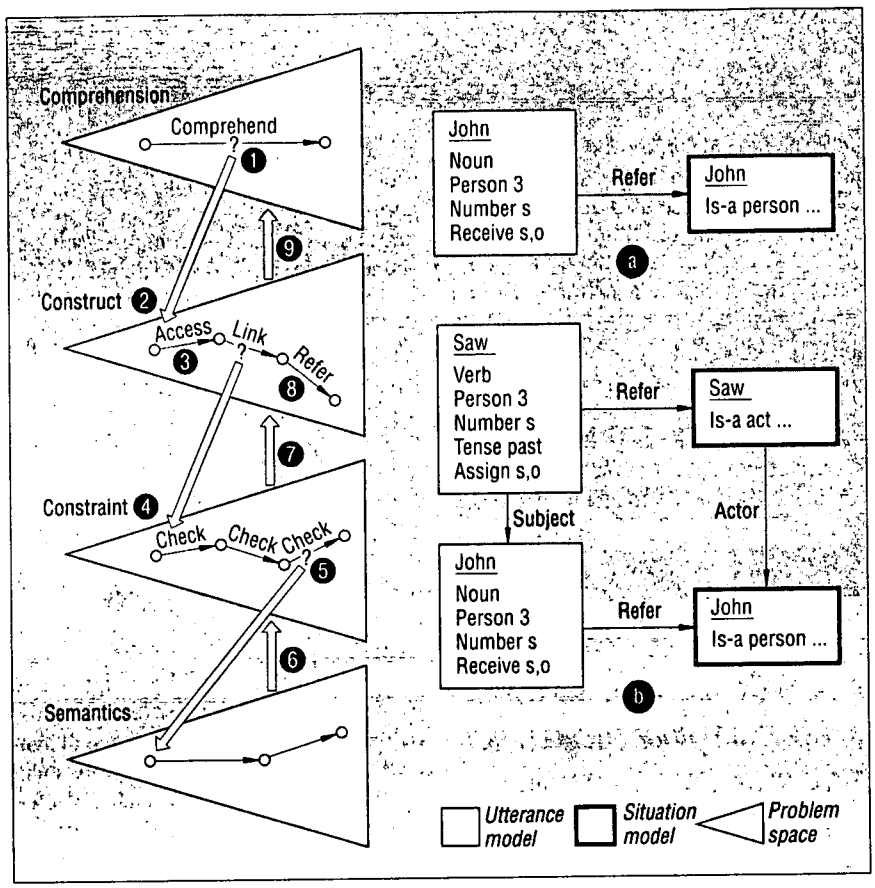

Figure 1. NL-Soar comprehending "John saw..."

- KS_1: *The lexicon* provides a context-independent mapping from words to a set of possible senses. Each word sense specifies the syntactic relations it may assign or receive, as well as semantic features to be used in determining the meaning of the utterance. The Access operator in the Construct space represents lexical knowledge (see Figure 1): It takes a single word and delivers a set of utterance model objects representing the different senses.
- KS_2: *Syntactic knowledge* is organized around syntactic relations, and specifies the well-formedness of putative links between words. This knowledge is represented by Check operators in the Constraint space. They take two words and a syntactic relation, and determine whether the link is well formed with respect to a particular constraint such as number agreement.
- KS_3: *Semantic knowledge* includes knowledge about how objects are categorized and how they relate to each other in the world. In NL-Soar, the lexicon contains information that constrains the kinds of objects that can enter into particular events or predications. This knowledge is used to create semantic constraints on links between words, represented by Check operators in the Constraint space. These Check operators are implemented in the Semantics space, which contains the object classification hierarchy.
- KS_4: *Pragmatics* includes many different kinds of knowledge, but here it is knowledge about how to identify objects being discussed in the current context (referent resolution), and knowledge about how to construct new pieces of the current context as novel topics are introduced. The Refer operator in the Construct space establishes the links between the utterance and the context. It incrementally matches accumulating descriptions in the partial utterance model against objects in the situation model, creating new objects when the match fails. Since this process is an integral part of word-by-word comprehension, the success or failure of referent resolution can be used to resolve local ambiguities.

The natural way to integrate knowledge sources KS_1, KS_2, and KS_3 is by generate-and-test, since KS_1 provides a generator, and KS_2 and KS_3 specify tests. This method is implemented by the Construct and Constraint spaces, which perform the Generate and Test operations, respectively. Combining KS_1 to KS_3 in this way produces a syntactically and semantically well-formed utterance model, which is combined with KS_4 to establish meaning in the current context. The latter happens via a modified assembly. The knowledge produced by KS_1 to KS_3 and KS_4 become two parts of the total knowledge delivered by F: knowledge about the syntax, and knowledge about the contextualized meaning of the utterance. (The method is modified assembly for two reasons. First, referent resolution (KS_4) takes as input the utterance model, and therefore depends on the content of KS_1 to KS_3. Second, when KS_1 to KS_3 fail to deliver a unique result, referential success can be used to select among the possible interpretations, thus making KS_1 to KS_3 dependent on KS_4.)

Unfortunately, the generate-and-test method of integration leads to comprehension based on search. NL-Soar naturally overcomes this limitation, however, via Soar's learning mechanism. Over time, chunking converts search-based integration into a recognitional capability that integrates the multiple knowledge sources into a single operator application. For example, consider the word "saw" in "John saw the man with the telescope." For explanatory purposes, we assume the system lacks the immediately available knowledge required to comprehend "saw" in this context. In the discussion that follows, numbers and letters in parentheses label the left- and right-hand sides of Figure 1, respectively.

Once the system has comprehended "John," the utterance and situation models each contain a single object (a). The former represents a noun with certain syntactic properties, while the latter represents the individual referred to by that noun. When the word "saw" is encountered, NL-Soar tries to apply the Comprehend operator (1). Because the system lacks immediately available knowledge of what "saw" means in this context, an impasse arises.

Conditions and actions	Knowledge source
If	
comprehending "saw",	Lexical
and there's a preceding word,	Syntactic
and the word can receive the subject relation,	Syntactic
and the word refers to a person,	Semantic
and the word is third person singular,	Syntactic
Then	
use the verb sense of "saw",	Lexical
and assign the word as the subject of "saw",	Syntactic
and establish a referent for the act of seeing,	Pragmatic
with the subject's referent as the actor	Pragmatic

Figure 2. The chunk integrating all the knowledge sources used to comprehend "saw."

NL-Soar chooses the Construct problem space to deliberately build the utterance and situation models (2). It creates an utterance model object for each word sense of "saw" retrieved by the Access operator (3). Then NL-Soar proposes a Link operator for each syntactic relation that "John" can receive and "saw" can assign. For the sense of "saw" as a noun, no links are possible. For the verb sense, however, there are two such relations: subject and object. NL-Soar chooses which of the two links to try arbitrarily; since the object link leads to a failure of the word-order constraint, we will follow the subject link. (No special mechanism is required to recover from a failed Link operator. Since Soar maintains operator proposals as long as their conditions are satisfied, as soon as one operator terminates—with success or failure—the remaining operators may apply.)

The proposal of the subject link leads to another impasse, because the Construct space does not know whether such a link leads to well-formed utterance and situation models. That knowledge is available from the Constraint space (4). In this example, three Check operators are applied corresponding to the three constraints that must be met: word order (the subject must precede the verb), agreement (the number of the subject and verb must agree), and semantics (the referent of the subject must be a meaningful actor for the act referred to by the verb). The knowledge required to verify that the proposed link passes the two syntactic checks is immediately available in Constraint. The semantic check, however, creates a final impasse into the Semantics space (5). In this space, to verify that John is a legitimate actor for "saw," the system performs simple inferences based on a small hierarchy of types.

Once the inference has been made, the impasse that led into the Semantics space is resolved because the system now has the result of the semantic Check operator. As the impasse resolves, a chunk is created that makes the inference immediately available in the Constraint space (6). Similarly, the impasse that led into the Constraint space is also resolved because each Check operator was applied successfully, guaranteeing well-formed models after the creation of the subject link. A chunk is created in Construct that makes the combined syntactic and semantic knowledge available in the future (7).

With the link made, the system creates a referent for the action in the situation model, with the actor of the event corresponding to the referent of the syntactic subject (8). Since the mapping is complete, NL-Soar has resolved the impasse that led to the Construct space (9). Figure 2 shows the resulting chunk that integrates all the knowledge sources used to comprehend "saw" in this context.

This chunk is part of the Comprehend operator for "saw." (It is not the operator itself, since productions do not correspond to operators in Soar). It was created automatically when the impasse between Comprehension and Construct was resolved. The next time "saw" appears in a similar context, the chunk will fire, bringing all the knowledge sources to bear simultaneously to create a structure similar to (b).

Increasing the efficiency of knowledge integration in NL-Soar depends crucially on the transfer (future firings) of these Comprehend operator chunks. Figure 3 shows the efficiency increase due to chunk transfer. The plot compiles data from running NL-Soar over a corpus of 61 sentences (358 words) from four domains. The horizontal axis is the cumulative number of words comprehended, and the vertical axis is a running 24-word average of the percentage of words mapped via the Comprehend operator, without impasse. When the system starts, comprehension is search intensive, but in the latter part of the run, the system understands about 80 percent of the words recognitionally. (The sentences were chosen primarily to test the syntactic coverage of the system, not to demonstrate transfer. The local fluctuations in the curve—for example, the drop at the very end—simply reflect the introduction of new lexical items or new syntactic constructions in the corpus. Future work will investigate the average over many permutations.) Without learning, the curve would be a flat line at 0 percent; that is, initially the system has no productions that directly implement Comprehend operators. During the run, the system learns about 1,000 new productions, more than doubling its size. Although this is just one data point over a relatively small corpus, it indicates that chunking provides real benefits for systems that integrate multiple knowledge sources, and that chunking may play a significant role in the development of large, efficient systems.

Production scheduling

In contrast to the situation for natural-language comprehension, relatively few adults perform as experts in the domain of our second example: scheduling production in a manufacturing environment. For the particular task we studied, scheduling the production of replacement car windshields, there are only two experts in the company, with a combined total of 30 years of experience. Merle, our domain expert, produces schedules for most situations by immediate recognition, and achieves satisfactory performance in almost all situations. We built a system, called Merle-Soar, that can also generate schedules for windshield production.[8] While Merle-Soar cannot produce schedules as quickly as Merle, it demonstrates a rather simple framework for scheduling and can account for how it might acquire scheduling expertise of Merle's caliber after years of practice

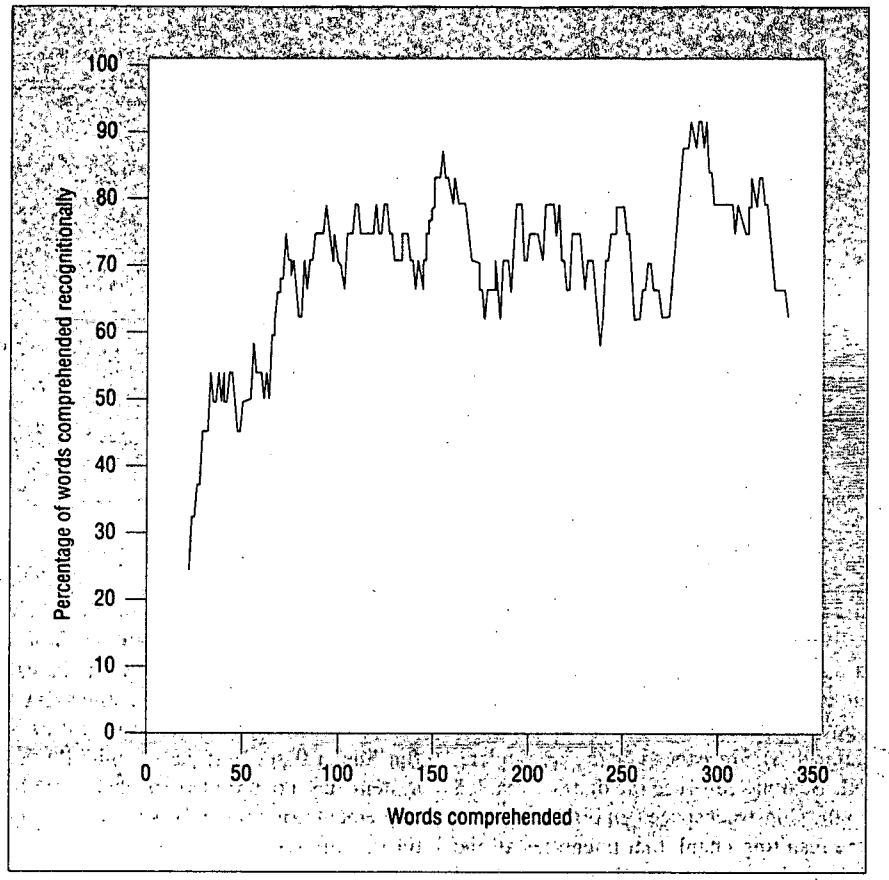

Figure 3. Efficiency increase due to chunk transfer in NL-Soar.

and building up chunks to perform the task through recognition.

Both Merle and Merle-Soar focus on the part of the production process in which windshields must be bent using a large oven called a lehr. The glass rests on molds, or irons, as it moves through the lehr, and gravity bends the glass to the correct shape. Generating the schedule involves deciding what type of windshield to produce (there are several hundred types), which lehr to use (a plant may have a few lehrs, each with different properties), how long the glass should be in the lehr (the belt carrying the glass can move through the lehr at different speeds), and how much glass should be put in the lehr (normally over a thousand windshields are produced daily). The schedule is a plan for five weeks of production, but is updated weekly to account for changes in the production environment, such as rush orders or a machine malfunction. The schedule is composed of sequences of reservation groups, where a reservation associates a windshield to be produced with a particular production environment (time, lehr, setting), and a reservation group is a group of reservations with the same lehr setup. Grouping allows several reservations to be shifted together without incurring additional setup cost. Every reservation group contains a primary (or driving) windshield, selected on the basis of the externally generated demand for its production, and one or more secondary (or companion) windshields, selected to make efficient use of the lehr capacity remaining after the driving windshield has been scheduled.

The need for knowledge integration in this task arises from the diversity of constraints that must be considered to produce a schedule. Merle has become an expert by virtue of his ability to incorporate a variety of knowledge sources into his problem solving. Following the knowledge integration formalism we have been using, the function F maps a set of orders for windshields into a schedule for producing those orders. There are three major knowledge sources for the constraints:

- KS_1: *Product.* Each windshield type has an associated list of attributes that can affect the schedule. These attributes specify length and width, presence or absence of sun shading, rearview mirror buttons, embedded antennae, ceramic bands, and so on. Each windshield also has associated with it an economic run length, giving the optimal batch size to be produced given previous experience with setup and production costs, inventory levels, and external demand for the windshield.
- KS_2: *Equipment.* Limits on production capacity follow from the capacity of the lehr and the number of irons available at one time to produce a particular windshield type. There are also bounds on the throughput of activities preceding or following the bending process; for example, the machines that attach antennae or rearview mirror buttons to windshields can only process a given number of units of glass per hour. Furthermore, more than one type of windshield can be produced concurrently, and this is often done to use up excess lehr capacity for small runs. However, compatibility constraints (such as size and heating patterns) must also be considered. Switching between windshield types may involve setup time, and it is desirable to minimize this.
- KS_3: *Labor.* Employees have a major impact on the production process. The production of certain types of windshields demands specialized (thus expensive) help that can only be brought in on a part-time basis, so these special jobs need to be scheduled together. Additionally, experience has shown that defects are reduced when complex parts are not scheduled for the first or last production sessions in the week, and when large and small jobs are alternated to avoid exhausting the workers.

In analyzing the constraints that arise from each knowledge source, we found it useful to distinguish between hard constraints, which must be followed for the schedule to be physically feasible, and soft constraints, which evaluate candidate schedules according to relative desirability. About 25 of each type of constraint have been identified. The primary method of integration is generate-and-test, combining the hard constraints in the generator and then using soft constraints to rank candidates.

Figure 4 illustrates this with the organization of eight of Merle-Soar's 18 problem spaces (the 10 not shown are implementation spaces that handle operator processing in these eight spaces). In the Solve-scheduling-problem space, the operators select a lehr on which to schedule production, select a driving windshield from a list of possibilities, create a reservation group, and update the schedule. These operators are repeatedly applied until the entire five-week schedule is completed, and then a final operator prints the schedule.

Each operator is implemented in a separate problem space, selected in response to the corresponding operator impasse. The Select-lehr space (not shown in Figure 4) selects the current lehr to be scheduled based on the earliest available one. The Generate-feasible-windshields operator is applied to the initial state to generate a list of the feasible windshield types that can be scheduled on the selected lehr. In applying this operator, Merle-Soar might encounter an impasse, resulting in its selection of the Apply-hard-constraints space. Here the system checks the five hard constraints (more can be added) by applying a corresponding operator, and separate problem spaces implement the operators. The Glass-availability-constraint space checks if the necessary glass is in stock. The Blocksize-width-constraint space checks windshield size restrictions for production on particular lehrs. The Antenna space checks that no more windshields with antenna buttons are produced per hour across all three lehrs than the antenna machine can process. The Rearview space checks that no more windshields with rearview mirror buttons are produced per hour across all three lehrs than the rearview mirror machine can handle, and the No-anten-last-shift space checks that no windshields with antennae are produced during the last shift of each week. There may be more than one feasible driving windshield available, so the Apply-soft-constraints space must decide which to select. Currently it chooses randomly.

Unfortunately, the Merle-Soar research project is no longer active, so soft constraints have not been implemented. While soft constraints are generally more difficult to measure than hard constraints, there is a straightforward mapping between the evaluation of a soft constraint in a particular situation and Soar's desirability preferences for operators. Thus it should be easier to incorporate soft constraints into a problem-space search framework than into alternative frameworks such as optimization of a mathematical model.

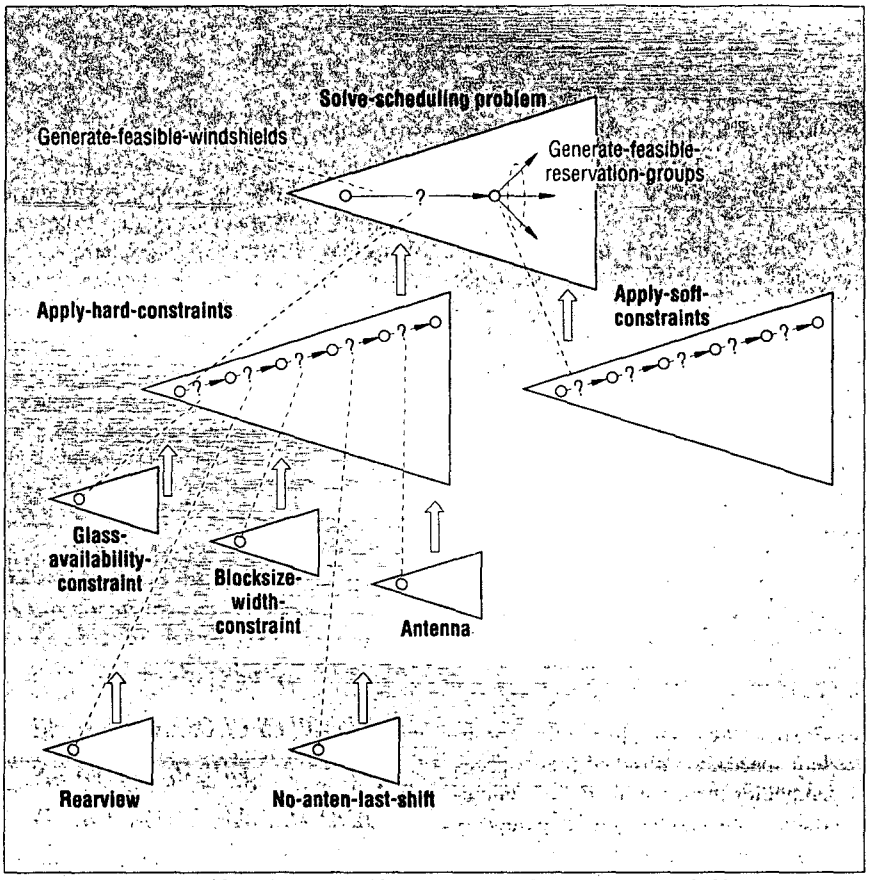

Figure 4. Organization of Merle-Soar problem spaces. Problem spaces are in bold type; operators are in regular roman type.

Similar processing (not shown in the figure) creates the reservation group for the chosen driving windshield. If Merle-Soar needs a companion windshield to fill the lehr, it generates a list of feasible choices. This generation differs from that for the driving windshield, in that it must apply four additional constraints to determine if the two windshield types are compatible. After selecting both, Merle-Soar uses several other problem spaces to carry out the mechanics of calculating times (including setup time if necessary), and updating several counters. These bookkeeping processes are all relatively algorithmic and do not involve searching.

Based on generate-and-test and assembly, Merle-Soar and NL-Soar's mechanisms for knowledge integration differ. In NL-Soar's generate-and-test, one source provides the knowledge for generating well-formed situation and utterance models, and others are applied successively to filter the candidates. In Merle-Soar's generate-and-test, several sources are combined to generate feasible schedules, and other sources are applied (or more precisely, would be applied if the system were fully implemented) in parallel to indicate preferences among candidates. In NL-Soar, modified assembly must be used to incorporate knowledge of pragmatics into the situation and utterance models, modified because of the interdependence of the knowledge sources. Merle-Soar assumes that product, equipment, and labor constraints are independent, so it has no analogue to NL-Soar's modified assembly. When the independence assumption fails, a solution might still be found at the cost of extra search—though this does not seem to be required for most of the hard and soft constraints we examined.

Algorithm design

Our first two examples illustrated knowledge integration in the context of

comprehension (a form of analysis) and scheduling (a form of planning). Design is a third form of problem solving. Specifically, algorithm design refers to what occurs from the time one understands, for example, what it means to sort a sequence in some order until the time one has an efficient procedure for sorting sequences on a computer. More abstractly, algorithm design converts an understanding of the problem in domain terms to an algorithm description in terms of the target computational model. Programmers recognize that algorithm design can require significant ingenuity, as there are no ready-made recipes. Using Soar, we developed a theory of the algorithm design process, virtually all of which was implemented in the Designer-Soar system.[9] The theory can be summarized as follows:

(1) Design takes place in multiple problem spaces. A subset of these spaces embodies the target model of computation, and another subset embodies the application domain model.
(2) The task of algorithm design is to use knowledge of the application domain, including a domain-level procedure for mapping inputs to outputs, to build a procedure for computing the desired output in the computational spaces.
(3) The computational spaces have functional operators corresponding to steps in an algorithm (Test, Apply, and so on), at whatever level of abstraction is needed for the design.
(4) Means-ends analysis on the results of execution drives the design, with the resulting series of execution passes in a design session exhibiting a pattern of progressive deepening, repeatedly traversing the same structure, exploring different aspects each time. Execution can be used for a variety of purposes, including explanation and efficiency analysis.
(5) Any part of the knowledge necessary for accomplishing the design task may be acquired by learning, ranging from an understanding of the problem in the domain spaces, to the algorithm itself, which is represented as a generalized path for navigating through the computational spaces.

Consider the design of a sorting algorithm. As part of the problem specification, we give Designer-Soar a set of problem spaces that allow it to access knowledge about sequences. This knowledge ranges from the fact that the sequence with zero elements is the smallest possible, to procedures that will sort a given sequence using domain-specific primitives. Designer-Soar will consult this set of problem spaces while deciding what to execute in the computational spaces. For example, it could choose to instantiate a scheme for Divide-and-conquer and obtain the test for the base case (the point at which the input need not be decomposed further and can be solved directly) by consulting the domain space to sort the empty sequence. The decomposition and composition steps are represented as operators in the computational spaces; they become targets for further refinement when the system encounters impasses in trying to apply those operators. At certain points, symbolic inputs (that is, "a sequence") are inadequate to drive algorithm refinement, in which case concrete sequences such as "{1, 2, 3}" are generated to drive the execution down one branch of a conditional or loop. The output sequences are compared to sequences sorted in the domain space to test for correctness. Later, Designer-Soar generates new inputs to force execution down alternative paths, and refinement continues on subsequent execution passes. When all execution paths have been explored (barring some abstraction allowed for loops), the algorithm is declared complete. The resulting algorithm is the set of chunks containing the search control and implementation knowledge for the generalized algorithm. Of course, because of its dependence on the examples chosen and the available domain knowledge, this process does not guarantee correct algorithms (no such general guarantees of success are possible for any generally applicable algorithm design method). But the method often works if the necessary knowledge is available to the designer, and Designer-Soar's behavior is remarkably similar to that observed in our protocol studies of human algorithm designers.[10]

OF COURSE, BECAUSE OF ITS DEPENDENCE ON THE EXAMPLES CHOSEN AND THE AVAILABLE DOMAIN KNOWLEDGE, THIS PROCESS DOES NOT GUARANTEE CORRECT ALGORITHMS.

Virtually every important aspect of the theory has been shaped by the necessity to integrate knowledge efficiently. Algorithms are represented in spaces of abstract functional operators because such spaces are easiest to search for the design task. Specifications are cast in procedures for searching application domain spaces, because this is the form in which knowledge about the problem is initially available in the environment. Execution in the context of means-ends analysis is a natural way to inspect models so that any potentially relevant knowledge can be brought to bear. The use of models is indicated because purely propositional representations might necessitate large amounts of computation to guarantee correct inferences (if such a guarantee is possible at all). With progressive deepening, the entire state of the design need not be kept continuously in working memory, since the relevant parts can be regenerated on each execution pass. Learning ties all this together by shifting knowledge between spaces, minimizing the expense of repeated execution, and lessening the load on limited working memory by transferring the knowledge to long-term memory.

Knowledge sources and integration. At any step of the algorithm, the function F applies the correct computational operators given any valid input. (This is the F for executing the algorithm; a different type of knowledge integration might be needed if F required the retrieval of the complete algorithm in some representation in one step.) Algorithm design is just whatever problem solving is required before the execution knowledge can be obtained recognitionally. We have incorporated the following knowledge sources into our theory of the algorithm design process:

- KS_1: *High-level algorithm schemes.* Part of the expertise of designers is the knowledge of design schemes (such as divide-and-conquer), which are used as kernel ideas to be refined later. These schemes are represented as collections of abstract

functional operators in the computational spaces of Designer-Soar.

- KS_2: *Transformations*. Knowledge for reformulating and refining algorithm schemes is represented as operators and control knowledge in computational spaces.
- KS_3: *Correctness*. By executing the algorithm in the computational spaces with either symbolic or concrete inputs, Designer-Soar can determine the input/output behavior of an algorithm after a candidate transformation. It then compares the results to those obtained by execution in the application domain spaces to determine correctness.
- KS_4: *Efficiency*. Knowledge can be encoded in Designer-Soar as operators and control knowledge. An example is the balancing principle, which states that the optimal divide step for Divide-and-conquer algorithms produces subproblems of equal size. Knowledge of the rough time complexity of an algorithm, such as would be gained by counting nested loops, can be accessed by an execution process specialized to perform efficiency analysis.
- KS_5: *Domain definitions, procedures, and relationships*. Concepts from the domain are used to decompose the problem specification and test for the correctness of proposed algorithms. For example, sorting can be defined as the problem of producing an ordered permutation of the input.

We might be tempted to classify the integration mechanism here as generate-and-test, because KS_1 and KS_2 combine to generate algorithms, and KS_3 and KS_4 combine to evaluate them. But in fact KS_1 and KS_2 generate only fragments of algorithms, and KS_5 plays roles in both Generate and Test. It would be more correct to say that integration in the service of design is performed in Designer-Soar via means-ends analysis, with KS_1 providing the initial state, KS_2 the operators that reduce differences, KS_3 and KS_4 the knowledge that detects differences, and all four knowledge sources incorporating KS_5 to some extent.

It is clear, however, that we are operating here at the frontiers of our understanding of knowledge source integration: Soar supports integration through means-ends analysis, but significant human intervention is still needed to use this method and organize the problem spaces for new applications. Furthermore, we believe the current organization is sufficient to produce the desired integration, but because the problem of designing good organizations is underconstrained at this point, we cannot advance the corresponding claim of necessity. Similarly, while we believe the Soar architecture is useful for providing the desired flexibility, we do not claim that Soar is the only architecture that can do so.

SINCE PEOPLE PERFORM THE tasks of language comprehension, production scheduling, and algorithm design, it is worth noting that our arguments for the usefulness of Soar mechanisms in integrating multiple knowledge sources do not depend on the merits of Soar as a theory of human problem solving. Rather, we claim that Soar has the potential to cope with diversity in knowledge sources in circumstances similar to those encountered by people.

Soar mechanisms reduce both design-time and runtime overhead associated with knowledge integration. Of these mechanisms, chunking is perhaps the most significant, because it converts knowledge from independent sources into directly available knowledge and thus, over time, eliminates the need for problem solving to integrate knowledge.

Remaining issues include errors in integration, and origins of knowledge about integration. That is, how does Soar recover if it has made an error in integrating multiple knowledge sources? And how can a system learn to integrate knowledge for new tasks without human intervention to organize the spaces? Our desire to increase Soar's autonomy will force us to address these issues more closely.

Acknowledgments

Allen Newell's guidance while performing this research was invaluable, and we acknowledge the extensive comments of Todd Johnson and other referees on earlier drafts. The research was sponsored in part by a Schlumberger Graduate Fellowship, by the Defense Advanced Research Projects Agency monitored by the Air Force Avionics Laboratory under contracts F33615-81-K-1539 and F33615-87-C-1499, and by the National Science Foundation under grant DCR-84-12139. Preparation of this article has been supported by the Engineering Design Research Center at Carnegie Mellon University.

References

1. A. Newell, "The Knowledge Level," *Artificial Intelligence*, Vol. 19, No. 2, 1982, pp. 87-127.
2. P.S. Rosenbloom et al., "A Preliminary Analysis of the Soar Architecture as a Basis for General Intelligence," *Artificial Intelligence*, Vol. 47, No. 1-3, 1991, pp. 289-325.
3. R. Neches et al., "Enabling Technology for Knowledge Sharing," *AI Magazine*, Vol. 12, No. 3, Fall 1991, pp. 36-55.
4. B. Chandrasekaran, "Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert-System Design," *IEEE Expert*, Vol. 1, No. 3, 1986, pp. 23-30.
5. S.M. Shieber, *An Introduction to Unification-Based Approaches to Grammar*, Center for the Study of Language and Information, Stanford, Calif., 1986.
6. *Principle-Based Parsing: Computation and Psycholinguistics*, R.C. Berwick, S.P. Abney, and C. Tenny, eds., Kluwer Academic, Dordrecht, The Netherlands, 1991.
7. J.F. Lehman, R.L. Lewis, and A. Newell, "Integrating Knowledge Sources in Language Comprehension," *The13th Annual Conf. of the Cognitive Science Soc.*, Lawrence Erlbaum Associates, Hillsdale, N.J., 1991, pp. 461-466.
8. W. Hsu, M. Prietula, and D.M. Steier, "Merl-Soar: Scheduling Within a General Architecture for Intelligence," *Proc. Third Int'l Conf. on Expert Systems and the Leading Edge in Production and Operations Management*, Management Science Dept., Univ. of South Carolina, Columbia, S.C., 1989, pp. 467-481.
9. D.M. Steier, "Automating Algorithm Design Within an Architecture for General Intelligence," in *Automating Software Design*, M.R. Lowry and R.D. McCartney, eds., AAAI Press, Cambridge, Mass., 1991, pp. 577-602.
10. E. Kant, "Understanding and Automating Algorithm Design," *IEEE Trans. Software Eng.*, Vol. SE-11, No. 11, Nov. 1985, pp. 1,375-1,386.

David M. Steier is a research scientist at the Engineering Design Research Center and the School of Computer Science of Carnegie Mellon University. His research centers on the application of AI techniques, especially integrated problem-solving and learning architectures, to problems in engineering design and human-computer interaction. He received his BSc from Purdue University and his MS and PhD from Carnegie Mellon University, all in computer science. The algorithm design research was performed while he was a graduate student there. He is a member of AAAI.

Richard L. Lewis is a doctoral candidate in computer science at Carnegie Mellon University. His research interests are in building computational theories of human cognitive processes, particularly language comprehension. He received his BSc in computer science from the University of Central Florida in 1987. He is a member of the IEEE Computer Society, the Cognitive Science Society, the American Psychological Society, and AAAI.

Jill Fain Lehman is a research computer scientist in the School of Computer Science at Carnegie Mellon University. Her research interests include natural-language comprehension, generation, and acquisition. She received her BS in computer science from Yale in 1981, and her MS and PhD in computer science from Carnegie Mellon in 1987 and 1989, respectively. She is a member of the Cognitive Science Society and AAAI.

Anna L. Zacherl is a graduate student at Georgia Institute of Technology, where she is currently working on a case-based design aid for architects performing conceptual design. The research in this article was performed while she was a consultant at Carnegie Mellon University and an undergraduate at Allegheny College, Meadville, Pa., where she earned her BS in computer science in 1991.

Steier can be reached at the Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA 15213.
Lewis and Lehman can be reached at the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.
Zacherl can be reached at the Georgia Institute of Technology, College of Computing, Atlanta, GA 30332.