Intelligent control of external software systems

by
Allen Newell, David Steier

EDRC 05-55-91

INTELLIGENT CONTROL

OF EXTERNAL SOFTWARE SYSTEMS

Allen Newell
School of Computer Science

David Steier
Engineering Design Research Center

Carnegie Mellon University, Pittsburgh, PA

4 April 1991

We wish to acknowledge the extensive help of the other members of the CESS (Interaction with External Software Systems) subgroup in Soar Bob Doorenbos, Jill Lehman, Xiaoping Li, Ajay Modi, Dhiraj Pathak, and Gary Pelton. This research was supported in part by the Defense Advanced Research Projects Agency (DOD), monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSQ, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597; and in pan by the Engineering Design Research Center, an NSF Engineering Research Center. The views and conclusions contained in this document are those of the authors and should noc be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the National Science Foundation.

Intelligent Control of External Software Systems

1. Introduction

A major objective of artificial intelligence (AI) is to develop *intelligent agents* that are capable of interacting with an external world using a broad spectrum of problem-solving and learning methods. The most familiar examples, already with a long history, are robotic systems, in which the input focus is on vision and touch and the output focus is on manipulation and movement — the goal being to interact freely and effectively in the physical environment in real time. However, in our technological world many highly specialized external environments exist that present configurations of technical demands quite different from the generic robotic problem. Some of these environments are also very important arenas from an applied viewpoint and therefore offer important opportunities for the application of intelligent agents.

We examine here environments comprised of *external software systems (ESSs)* — software systems that are created and exist external to the agent that uses them. With the pervasive growth of software packages and tools, along with networks that let them intercommunicate, *software-system environments* have been created in which software systems can contact and use other software systems. We see this already in the extensive role played by servers and mail systems on networks, which engage in sustained autonomous action. We see it also in the way humans use such tools to compose transient assemblages of widely-distributed large software systems, in order to accomplish some computational objective. As we move toward gigabit networks, the reality, complexity and importance of these software-system environments will continually increase.

Software-system environments permit many modes of interaction, depending on the nature of the systems. Systems vary in terms of the grain size of the interaction and the amount of data involved in their inputs and outputs. Some systems operate in an essentially batch mode, receiving the total input for their task all at once, doing an extended computation and then storing their results for later retrieval. Others, such as editors, split the total task into an extended discourse of small commands and responses between the user and the system. Humans, of course, are part of such software-system environments, interacting with the system via workstations and terminals. But increasingly many software systems interact with each other, as in protocols that set up communication paths or systems that receive and automatically process mail requests, over a network.

We wish to consider a software-system environment that has a collection of software systems and a single *artificial intelligent agent*, hereafter, simply *agent*. This agent is to accomplish its tasks jointly by its own internal powers and by making use of the other external software systems in the environment. Our main concern is to understand what capabilities are required for this agent in order to use these ESSs. Consequently, we will assume there are no humans in the environment. This would introduce issues of the interaction of two intelligent agents, which are beyond the scope of this paper. Ultimately, of course (and maybe not too far away), software-system environments will be inhabited by large collections of humans, agents and other software systems, all interacting with and using each other. But we wish to avoid the complexities of human-agent and agent-agent interactions until we understand how a single agent can effectively use ESSs. Moreover, this special case — the agent-ESS situation, with one agent and one or many ESSs — is extremely important in its own right. It will permit the construction of many

total software systems of high effectiveness.

Our discussioo is motivated by a particular AI agent, *Soar [7],* which has been under development for several years. This agent already exhibits integrated problem solving and learning, and has been used as a controller in several simple robotic situations. We will need only a few details of Soar (which we will introduce as appropriate), but Soar provides us with a concrete model of an agent and its general structure. It also provides examples of an agent working with ESSs.

We first describe, in Section 2, a concrete example of an ESS situation, to ground the discussion. Then, in Section 3, the main section, we provide an analysis of what capabilities are required of an agent for it to be able to use ESSs effectively. In Section 4 we describe several different ESS situations, which are currently being developed with Soar. These show that a variety of critical issues arise, depending on the type of ESS situation involved. In Section 5 we briefly place the agent-ESSs situation in the world of software technology. In Section 6, the final one, we summarize the situation in terms of a general research agenda. Most of the issues raised in this paper have never been seriously explored before. They must be addressed, both theoretically and experimentally, if we are to develop the capability for agents to control ESSs.

2. An Example: Design of high-rise office buildings

Many ESS situations involve only a single ESS — e.g., the use of a database or a process simulator by an agent. But ESS situations may more complex, as the following example illustrates.

The evolution of engineering design is toward increased use of software design tools, each of which represents a significant investment in design research, software development and knowledge engineering. The design of any substantial actual artifact, e.g., a building, involves using a number of such tools. Thus, engineering design finds itself drawn into an ESS situation, where there is a human designer (or team) with access to all the appropriate tools on a local-area network, and with the task of using these tools in an integrated manner to create a total design of a complex artifact.

A good example occurs in the work of the Engineering Design Research Center (EDRC) at CMU (an NSF engineering research center). Seven knowledge-based systems have been created at EDRC to support the preliminary design of high-rise office buildings. The seven systems were developed independently, originally as part of students[9] PhD thesis research, with some re-implementation and extension. Listed in the order they are usually invoked, the seven systems are:

1. ARCHPLAN: Develops architectural plan from site, budget, geometric constraints.
2. CORE: Lays out building service core (elevators, stairs, etc.).
3. STRYPES: Configures the structural system (e.g., suspension, rigid frame, etc.).
4. STANLAY: Performs preliminary structural design and approximate analysis of the structural system.
5. SPEX: Performs preliminary design of structural components.
6. FOOTER: Designs the foundation.

7. CONSTRUCTION PLANEX: Generates construction schedule and estimates cost

These tools, together with a computational infrastructure that allows the tools to work together, have been combined in the EBDE (Integrated Building Design Environment) project [4]. The development of **the** integration framework of IBDE has been a rather formidable undertaking, involving the bulk of several PhD theses and a substantial effort measured in person-years.

Figure 1 shows the structure of IBDE. It adopts a hub-and-spokes approach with a centralized global datastore to contain all the information about the building in a hierarchical object-oriented representation. However, each tool uses its own data representation and these differ radically. CORE uses orthogonal structures for generating alternative layouts of rectangular objects; ARCHPLAN, STANLAY, and STRYPES use the so-called Tartan Grid for storing information about the spatial, planar, linear, and point elements on a common orthogonal grid; PLANEX uses labeled points for combining spatial information about each building element with associated construction activities; and FOOTER and SPEX use ad hoc attribute-value representations. Thus, custom-built translation modules are required to allow the datastore to pass information to and from each tool The datastore manager is responsible for communication between the datastore and the individual processes, retrieving and storing data when necessary, performing format conversions and maintaining an audit trail of the producers and consumers of each data item in the datastore. Communications with the human are relatively complex, since each component system has its own interface (recall that each such system is a large software system-on its own account and was developed independently); but there is also an interface that deals with the collections of ESSs as a whole. This latter keeps audit trails and process information on a blackboard Thus, the entire IBDE system is quite complex, which arises in part simply from the total functionality involved (the design of an entire high-rise building) and in part from the fact that it was assembled from individually-designed large software systems.

The IBDE structure assumes a human user, who works via the control module shown in the figure. The function of the controller is to make it possible for the human to specify the execution of sequences of the seven systems. This controller is rather sophisticated, using a blackboard structure and a modified contract-net framework to allocate tasks. Nevertheless, its functions are fundamentally low-level. It is not capable of getting a complete task performed by the total system. That is the function of the (intelligent) human user.

Our interest here is to replace the human in Figure 1 with an agent (i.e., an AI system) that is to perform the same overall functions as the human designer. We can take for granted the functions performed by the controller, which permit communication and execution. The question is what other functions axe being performed by the human (now, the agent), what knowledge must be available in the agent to perform these functions, and how is that knowledge to be acquired. The discovery of these functions and the construction of agents that can accomplish them is the central core of research on agent capabilities for controlling external software systems.

An actual development path for IBDE would not begin with the replacement of the human by an agent Rather, an agent would be inserted between the human and the controller. This agent would attempt to do as much of the total design task as the human user-designer found useful. Undoubtedly, this would tend toward having the agent know the details and idiosyncrasies of the operation and results of the ESSs (the seven systems in the figure), so that the human user-designer could operate in a higher supervisory and guidance mode. One motivation for this path

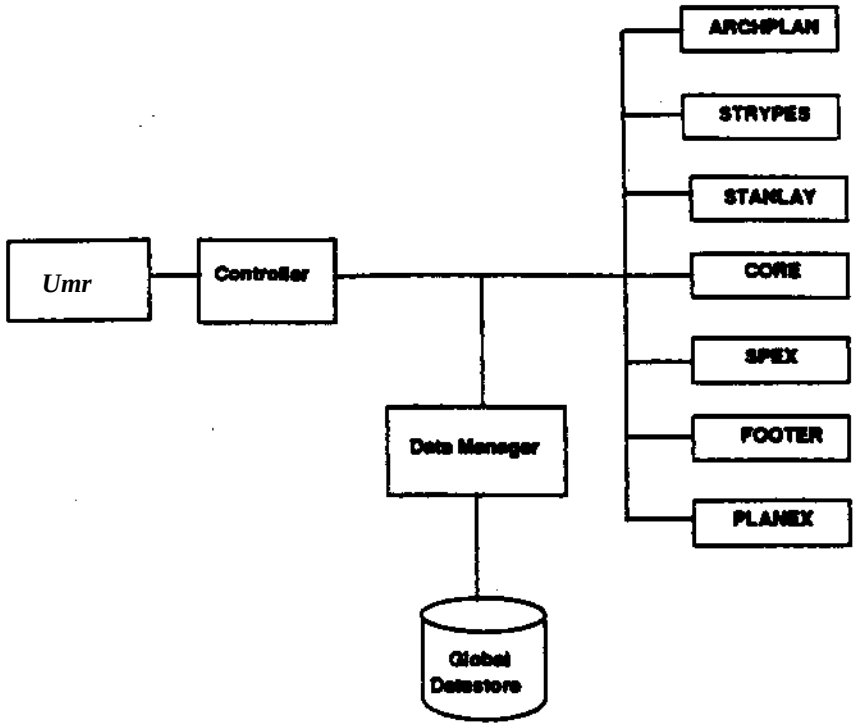

Figure 1: IBDE: An example of an external software system environment

might seem simply that society wants human-controlled and -designed engineering artifacts. But equally strong is that no one knows all the functions that the intelligent human is performing in Figure 1. All we know is that the human does *whatever is necessary* to get a total effective design. Only when we discover what these functions are, and the ease or difficulty of incorporating them in agents, will we be able to create total systems that have agents as appropriate designer-assistants for humans functioning as high-level supervisors. The purpose of the present paper is to explore what these functions might be and to consider the research required to see how to include them in agents. We do so, as we noted above, by focusing on a single agent with a collection of ESSs. But the ultimate application path envisioned involves cooperative arrangements with humans and agents.

3. The Capabilities Required to use ESSs

Let us now ask what functions the agent must have. Figure 2 gives the generic situation. It looks similar to Figure 1, except that we have expanded the agent, showing its inner structure. We take the agent as already having some task to perform, represented within it in some fashion independent of the ESSs that are available to it. Thus, at the most general level the agent must accomplish all those things that will lead it to use the ESSs in some manner of its own determination in order to accomplish some aspects of its task. However, it need not accomplish all of its task via the ESSs, since it has problem solving and computational capabilities of its own.

As in IBDE, some of the capabilities required for performing these steps are operating-system

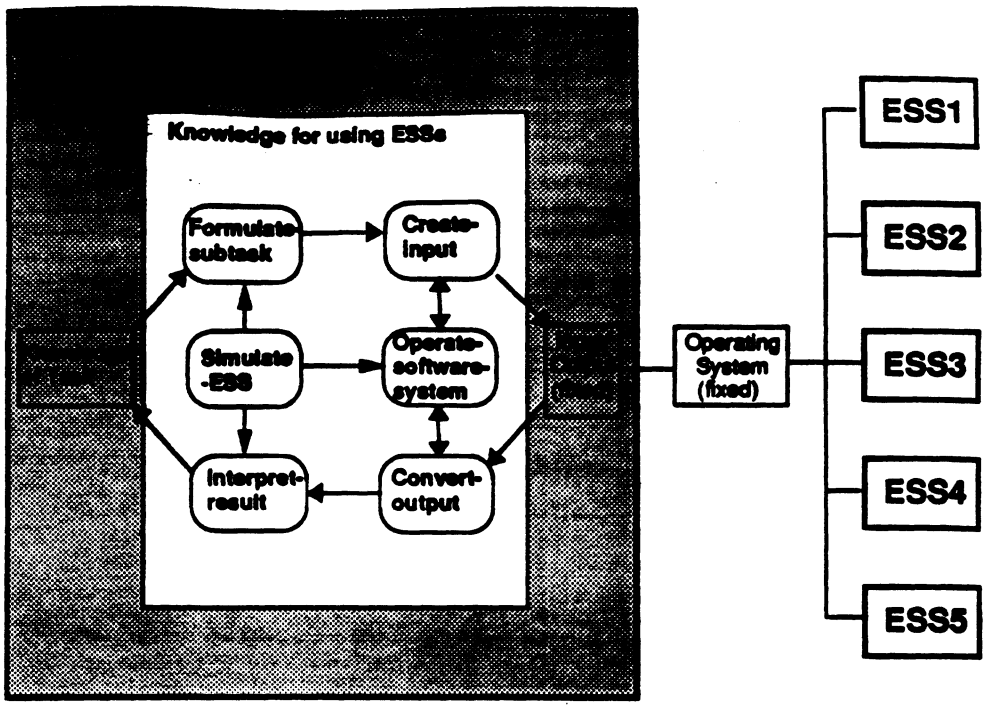

Figure 2: Performance capabilities for using external software systems (ESSs).

level functions. We assume, as in IBDE, that the agent has available to it a system that performs those functions. EDRC's Distributed Problem Solving Kernel [3], used in IBDE, is an existing (though imperfect) example. Then, as long as the agent and the ESSs are on the same network, the operating system can locate ESSs, cause inputs to be transmitted to them, evoke them, direct their results to be transmitted elsewhere, etc. The design of this operating system is important, but its functions are already reasonably well understood. Thus, we will not deal with it further.

Our analysis takes the following form. We view the agent as having a collection of *capabilities*, defined by the functions they can perform, rather than by how they are implemented (in particular they are not assumed to be separate modules in the agent). We are interested only in capabilities that are required for utilizing ESSs. Capabilities can, of course, be performed with different degrees of *skill*. The level of skill makes a great deal of difference to the performance of the agent as a whole, and also to its flexibility. Finally, capabilities and the skill with which they are performed must be *acquired*, and they must be acquired from *knowledge sources*. This also is important. Much of what is involved in using ESSs is obtaining the appropriate capabilities in the first place and keeping them current. The world of ESSs is much too dynamic to think of an agent existing at the start with all its capabilities, full fledged.

3.1. Performance Capabilities

The **fundamental** cycle required for an agent to use an ESS consists of four capabilities. These are the outer four capabilities in Figure 2 (the other two will be introduced presently). Within the context of a **local** task to perform, the agent *formulates* a subtask to be performed by the ESS. It then *creates* the appropriate input data structures for the ESS. Using the operating-system functions, it communicates this input to the ESS, evokes its operation, and obtains the resulting output It then *converts* the received data structures to an internal form it can process. Finally, it *interprets* the results in terms of the original task.

Let us consider these capabilities in more detail

1. **Formulate-subtask:** The capability to formulate a computational subtask with the potential of being performed by an ESS. The agent must decide whether a subtask can be solved internally, or if externally, which ESS should perform the task This capability requires understanding the functional demands of the task and the functional capabilities of available solution methods in order to determine whether and how an ESS can be used in the attempt to do the subtask.

2. Create-input: The capability to create appropriate input to an ESS in the ESS's input language, starting from task knowledge in the task's own representation. The translation capability is required because the task representation and the ESS representation are uncoordinated. (The operating system capability we are assuming deals only with lower-level operations of making connections and transmitting messages.)

3. **Convert-output:** The translation capability, analogous to create-input, to receive output from the ESS in the ESS's output language, and represent it in the task's own representation.

4. Interpret-result: The capability to use the results of a computation in the service of the original task (when expressed in the task's own representation). In many cases, formulating the computational task determines exactly how the results are to be interpreted, so that this capability can effectively be dispensed with. But more generally, interpretation of results need not presuppose pre-envisionment of exactly how the results will be used. ESSs can be deliberately used for exploratory purposes. Also, results from a computation almost always have the potential for surprise, requiring reconsideration of the larger task.

This basic four-capability cycle can be seen clearly in the simple situation of an agent that uses a relational database as part of some larger task it is doing. Input to the database is via some query language, such as SQL; output from the database is via some table structure. In accomplishing the larger task, there comes a point where the agent needs some data it does not have. This occurs in the context of dealing with the internal representation of the task, and the needed data in seen in terms of this representation. There follows *formulate-subtasfc* to determine exactly what dam should be requested as a function of other data in the task and the known types of data in the database. This determination is made in terms of the internal representation. Then comes *create-input,* to cast this request in terms of SQL and knowledge of the table organization of the database. This may be entirely routine, if the request is simple enough, but (as any user of SQL can testify) issues may arise how to express the request in SQL and which searches are the appropriate ones. In either case it is necessary to perform a process, which is the exercise of the create-input capability. There follows a series of activities, which

we have assigned to the *operating system* and taken for granted: getting the SQL request to the database, getting the search performed, and getting the results back. The agent then faces a set of tables in a fonnat determined by the database and some standard conventions of how to encode such tablet in text streams. This received representation is definitely not the representation of die internal task. Thus, *convert-output* is required to put this data into a form interpretable by the processes that ate performing the internal task. Finally, *interpret-result* can occur. This process is likely be minimal, since specific data has been requested and, when delivered (and converted) can simply take its place in the ongoing processing of the internal task. The elementary use of a database is a good illustrative example of the basic cycle, because it makes clear the role of conversion between representations. No one organizes their internal processing of a task around SQL and they are unlikely even to organize it around the sorts of tables that are retrieved

Additional capabilities are involved in performance. One is related to issues of how to *operate* software systems — more is required than just shipping inputs and receiving outputs, especially if the software system is at all complex.

5. Operate-software-system: The capability of dealing with the operation of the ESS as a software system, with its normal and abnormal operating conditions, and a corresponding need for diagnosis and operational response. The basic four-step cycle treats the ESS exclusively in terms of the semantics of what it does — it delivers as output certain task-relevant knowledge if given the appropriate inputs in the appropriate data representation. But ESSs are in fact software systems embedded in a larger software operating environment And the (perhaps sad) state of the an is that ESSs are not totally transparent in use. Well-designed and debugged ESSs are better than flaky systems or systems still under development Much can be smoothed over by a good operating system. But the software world, especially the world of large software systems, is far from perfect An agent that has no knowledge that its ESSs were indeed software systems, but only knew to shove inputs at them and wait for outputs from it, would soon find itself immobilized. Thus a capability for treating ESSs as a software system is required.

The last capability is related to being able to *simulate* the computations done by an ESS. We know humans use such knowledge in working with software systems. In fact, when they have no such knowledge we accuse them of operating mechanically or blindly, and of not understanding what they are doing.

6. Simulate-ESS: The capability to produce internally the same results as a specific ESS. The results may range from the exact output data structure delivered to a given input data structure, to abstract characterizations of the behavior. They may be accurate or only approximate. The essential feature is that the agent can do this itself, without actually evoking the ESS, and that it has some access to its own internal version of the computational process that produces the simulated results of the ESS. This capability is potentially of use in formulating a subtask, interpreting results and even operating the software. For example, it can provide sanity checks on the behavior of an ESS. Also, knowing something of the actual computations provided by an ESS can play a significant role in deciding what subtask an ESS can do and formulating it for (he ESS. If ESSs are expensive, as is often the case, being able to obtain their outputs cheaply in special cases can be useful. We know that humans that understand a set of ESSs gradually build up this sort of capability, but we have little idea of the full range of uses it can be put to.

3.2. Skills

Capabilities are the operational expression of some body of knowledge, realized ultimately in operations on **dffft** structures. But many different ways exist to organize a given body of knowledge. **Figure** 3 shows five such organizations, which form a sequence of levels of *skill. Recognition** at **die** top, provides the highest degree of skill and the fastest operation. As one **moves down to *deliberation* interpretation* simulation and finally** *derivation,* **more** and more processing is required to determine what to do, and the capability is exercised more slowly. The other side of this coin is that, as one moves down the sequence, the knowledge that constitutes the capability can be represented in more and more complete ways. Thus the possibility increases of exercising the capability flexibly in novel situations. This is just a variant of the familiar procedural-declarative dimension.

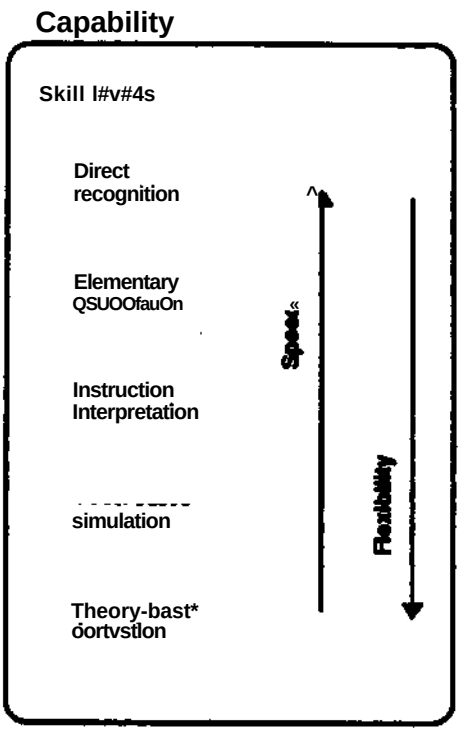

Figure 3: Skill levels for capabilities.

Let us consider each of the five skill levels in more detail.

1. **Direct recognition:** The agent immediately recognizes what operations are required to implement the capability. This can be viewed as a set of parallel-acting situation-action rules (or productions). If there are enough patterns and they cover all the situations that in fact arise, behavior is fast and direct. This corresponds to the skill we see in expert humans using a hand calculator or doing simple computations on a system such as an editor.

2. Elementary deliberation: The agent is skilled at performing the basic operations, knows about plausible ways of doing things and can evaluate proposed sequences of action. But it does not know immediately exactly the right thing to do, so it must consider options and alternatives, and evaluate them. This corresponds, in

humans, to there being a pause to "give thought" to what to do next. A user with moderate experience with relational databases exhibits this level of skill in formulating a query in SQL — it isn't quite automatic. In AI systems this corresponds to formulating tasks in problem spaces (i.e., heuristic search spaces) where some search is required before discovering a solution. It is the level at which general reasoning methods such as hill climbing, means-ends analysis, and planning by simple abstraction are routinely used. However, this activity should not be seen as the immense combinatorial searches associated with, say, current chess programs. Rather, this exercise of skill consists of a sequence of routinely resolved, small problems, each representing a small act of deliberation on the part of the agent. It is apparent why this level of skill is both slower than recognition (the extra steps) and also why it is more flexible (deliberation is precisely the ability to consider alternatives and bring new knowledge to bear).

3. **Instruction interpretation:** The agent may have a capability by virtue of having a set of instructions that dictate explicitly the operations to be performed under various conditions. At the extreme, of course, this is just what a computer program is — fetch-execute — which is the fastest way of operationalizing a capability. But the sort of instructional situation intended here is more open and flexible. It is having a manual of operation of an ESS, or a set of detailed guidelines for how to use an ESS, or a set of cliches to evoke specific ESS behavior. Thus the process of interpretation is more complex and requires more processing than the basic fetch-execute cycle for computer instructions. The instructions themselves are correspondingly more expressive and cover alternative and exceptional situations. Instruction following is like deliberation, except that the knowledge has not been internalized, but must be extracted from a declarative data structure (the instructions) at the time the task is performed, Hence, it is apparent why this level of skill is slower in general than deliberation. That it might, correspondingly, be more flexible arises from the amount of information that is kept in books, manuals, and other documents — rather than in people's heads. Thus, access to instructional documents opens up the range of tasks that can be solved. Unfortunately, describing this skill level as instruction taking de-emphasizes the correlative skill of finding the relevant instructions, which is required if the instructions are available in documents and not simply given to the agent at execution time.

4. **Model-based simulation:** The knowledge of a capability may be embodied in a model, which the agent can inspect or run to determine how to implement the capability. For instance, in the IBDE situation, the agent could have a highly abstract simulation model of the operation of the seven ESSs, which take in abstract inputs, characterized only by type of information, and deliver abstract outputs, similarly characterized. Then much about how to operate the ESSs could be obtained by running and examining the simulation model.

5. **Theory-based derivation:** The knowledge of a capability may be embodied in a set of principles, constraints or propositions, from which the agent can derive the actions to implement the capability. For instance, its knowledge of SQL syntax may be given by a BNF grammar. Or the ESSs may conform to a set of input/output conventions, and the agent has a set of expressions for these conventions in (say) predicate logic. Or the response time of the ESSs may be given by a set of equations on parameters of the input data, which the agent has access to. This mode of representing knowledge is, almost by definition, the most general of all (as witnessed by the fact that scientific theories invariably are

expressed by such formalized, linguistic descriptions), and thus offers the greatest flexibility in containing the knowledge to cover a wide range of novel contingencies and uses. But correspondingly, the effort required to derive specific results caa be essentially unbounded — it depends on the difficulty of the derivations and the gfriii of the agent at doing mathematics or proving theorems

Although there might be other forms in which the knowledge of capabilities can be encoded, the set of five in Figure 3 covers the major cases. They support the essential point that a capability can be implemented in radically different ways, with important consequences for the exercise of the capability, i.e., for its speed and flexibility.

We have described skill levels as pure types. An actual capability for a complex function, such as formulating a task to be done with the IBDE collection of ESSs, might well have elements of all these skills, for different aspects of its capability. Indeed, it might have several different skill levels for the same aspect If an agent started out with its knowledge in prepositional form — giving it generality, but slow response — it would be natural to expect it to acquire more efficient levels of skill. But there would be no reason to abandon the original forms of knowledge, especially since the higher (faster) levels of skill might well be narrower in scope.

3.3. Acquisition from knowledge sources

Capabilities, and their realization as particular skills, must be acquired In the current world of software systems (including programmed AI systems), the default is always that the capabilities are simply programmed by the system's creators or maintainers. Thus, some human programmer must design, code and debug the six capabilities of Figure 2 for each ESS, and, if these are to be available at multiple skill levels, each level must be so designed, coded and debugged. This is easily recognized as another instance of the *knowledge-acquisition bottleneck.*

The situation is actually more serious than indicated by the above, essentially static, view. The actual world of ESSs is highly dynamic and changing. New ESSs are introduced, as in additional modules for IBDE. Existing ESSs are continually corrected or functionally enhanced. Agents continually find themselves spending too much effort reasoning through some situation (say, the solution of a math model), hence need to introduce an ESS! (say, a math-model solver) to increase efficiency. In all such cases, if the static view above holds, the agent is helpless until some human reprograms it More generally, as any human that works with complex software systems can testify, large fractions of life are devoted to learning about particular software systems and developing skill in working with them.

Thus, the agent itself must be given the capability for acquiring the requisite ESS capabilities. This implies agents that learn and that do so in a variety of ways and from a variety of sources of knowledge. It is very likely that the degree to which agents can learn specific ESS capabilities will be a strong limiting factor on how effective agcnt-ESS system will be. Of course, even with only human-programmed systems, some useful arrangements will be possible. For example, agents can be provided once and for all with the capabilities to deal with generic relational databases. But agent-ESS system will never become an important part of software technology if they are so limited. In sum, learning capabilities is of the essence.

It requires a capability to produce a capability. Figure 4 shows the situation. On the right is

the capability to be acquired. This might be either to acquire the capabiUty from scratch or to extend the scope of the capability. Furthermore, to acquire a capability means to acquire it at some particular **skill** level. In fact, if the capability exists, it does so at some skill level. Thus, what is being acquired may be a different skill level — either a more efficient one or a more flexible one.

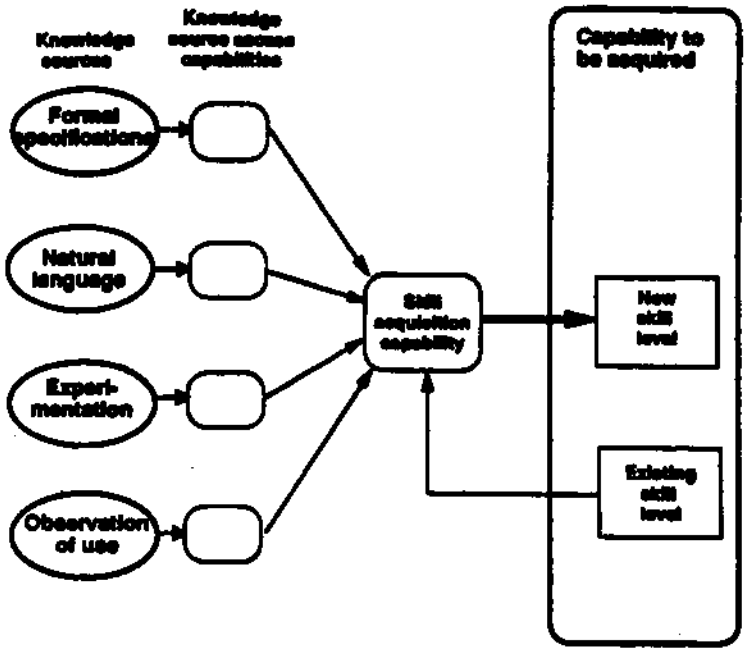

Figure 4: Capabilities for acquiring ESS capabilities.

There must be some existing *source of knowledge* about the new capability. It is not possible to get something from nothing. The learning capabilities exercised by the agent are really capabilities to exploit some knowledge source that has knowledge about the capability, and convert this knowledge into a skill leveL Thus, on the left, Figure 4 starts with various knowledge sources. These **are** characterized in terms of their form, i.e., of the way they hold their knowledge. Each of these types of sources can contain knowledge about any given capability, e.g., about create-input, or simulate-ESS. We have indicated some important types of sources, but the **list** is haidly complete.

1. **Specifications in some formal language:** The new capability may be described in some formal language whose function is to specify capabilities. E.g., it might be a formal specification language, or a high-level program description. Then the knowledge-source access capability would contain an interpreter or compiler for this language, as well as capabilities for analyzing the specification as a body of text.

2. Natural **language** text about **the capability:** The knowledge about a capability may reside in manuals, design documents, descriptions of how to use systems, etc. — all created originally for human consumption, hence all in natural language

(though often with additional diagrams and formal notations).

3. **Experimentation with the capability:** Much knowledge about how to do anything, **such** as run an ESS, comes from experimenting with doing so, and inducting **a** more general capability. Indeed, for humans we know that, after a point, the only way to learn about computer systems and how to use them is to try out simple cases and discover how matters go.

4. **Observation of the exercise of the capability:** Much can be learned about how to do something by watching an expert do it, i.e., being an apprentice. It helps, of course, if the expert is sympathetic and provides constructive commentary. In some ways this can be like guided experimentation.

These knowledge sources are each radically different in character and involve complex activities to extract the knowledge they contain. Comprehending natural language is a long-standing and difficult area of research in AL AI systems that learn from active experimentation in an external world are just being developed. Research in AI on learning apprentices was initiated some years ago, but they have actually received very little attention. Only the use of formal specifications designed for acquiring a capability has the possible character of requiring only routine operations to obtain their knowledge. Thus, each of these sources requires a separate, complex capability to extract its knowledge. In the figure we have associated such a capability with each knowledge source. However, the capability to exploit a source of knowledge is relatively independent of the particular task for which the extracted knowledge is to be put For instance, a single natural-language comprehension facility subserves the use of language to convey any body of knowledge. So we show all these knowledge-source capabilities as feeding into one skill-acquisition capability, which pertains to how to use the extracted knowledge to create a given skill.

Except in special cases, the total knowledge for a capability will not all be available in a single type of source. If human experience is a guide, to learn how to use a complex software system involves all four types of sources — some formalized description, surrounded by natural-language explanation, then some introductory sessions with an expert guide (whose helpful commentary is also in natural language), and finally hours of low-level experimentation with the system to build up familiarity and understand what all the previous instruction really meant.

All four of the knowledge sources at the left of Figure 4 are external to the agent But the existing skill level is also a potent source of knowledge. One of the best ways to create a higher skill level is to learn from the ability to operate at the current skill level. Thus, the figure shows a fifth knowledge source, which is internal to the agent There would be one such potential source for each existing level of skill, since each provides knowledge in quite different forms for moving to a new skill level.

Each of the five skill levels in Figure 3 is a software organization and technology of its own. Consequently, the skill acquisition capability in the center of Figure 4 stands in for five separate capabilities, depending on what skill level is being constructed. It is simply convenient to represent it by a single capability in the figure.

Figure 4 describes a quite general scheme for how to acquire capabilities and their skills. It applies to the acquisition of each of the six performance capabilities of Figure 2. But it also

applies to each of the capabilities in Figure 4 itself — that is, to acquiring the capability of acquiring a capability. It might seem that only the first-order acquisitions would be required, and that all the capabilities in Figure 4 could be fixed capabilities, implemented in a fixed way for all time. A moment's consideration shows this not to be the case. The problems of acquisition are much too central to working with ESSs. For example, consider a case analogous to IBDE case, but where new ESS modules are continually being created and added, e.g., to evaluate the intermediate results of the main design modules. Then, these added ESSs will all no doubt bear a family resemblance to each other. An agent should to be able to exploit the commonalities of this family of modules, and not to have to deal with each new ESS as an instance of a completely generic ESS. That is, the capability for acquiring the capabilities for members of this family of ESSs should improve with experience (a second-order acquisition).

The scheme laid out in Figures 2, 3 and 4 presents a bewildering array of capabilities, skill levels and knowledge sources. Everything seems to occur in multiple alternatives. Why can't one settle for one simple version — an operating system good enough to avoid having an operate-software-system capability, all capabilities described in some formal specification language, and a single skill level. The fundamental reason this simpler picture cannot prevail is the nature of real software-system environments. IBDE exists prior to trying to get an agent to use it. Not all databases use SQL; many have their own query language. If one wants to have an agent exploit *Mathematica*, because it has some powerful properties, then *Mathematica* must be taken as it is. Thus, the great appeal in getting agents to control ESSs is to be able to handle the actual variability and complexity of the real software world. Certainly, demonstration agent-ESS systems could be built by specializing all the dimensions of variability. But the challenge (and the real usefulness) of agent-ESS system lies in confronting and conquering the variability (even if only bit by bit).

4. The Space of ESSs and their Differing Demands

There are many ESS situations and they place quite different demands on the agents that use them. Understanding the range of these demands is important to developing effective agents for ESSs. Certain aspects can be critical universally for all applications, whereas others occur only in special situations. Certain aspects cannot be developed before others. It would be pleasant if we could describe a space of such demands, which would make the issues clear. However, at this stage of understanding, we do not know enough to do that.

We can examine briefly some actual examples of efforts to produce an agent capable of using ESSs. These examples exhibit, with varying degrees of clarity, the roles of different capabilities in different situations, and they give hints of important research issues. All the examples involve Soar [7], the AI agent that is our experimental vehicle for ESS research. All these Soar systems are in early stages of development and some of them are just getting started. Consequently, the amount of experience they provide about ESSs is still extremely limited.

Before taking up the examples, we make some general observations regarding the use of ESSs by Soar.

First, all these Soar systems have been initiated and developed for purposes other than that of exploring ESSs per se. For instance, the Soar/IBDE effort, for which ESS's are central, is focused on how to provide integration for engineering design of very large complex artifacts.

The ESS situation arises because that is the current state of engineering design, which is beginning to produce large numbers of independent computer tools, whose integration seems always to be left as an exercise for later. Only as we have become aware of the extent to which ESSs are entering into many Soar systems, have we begun to focus on the common intellectual core problems of obtaining intelligent control and use of ESSs.

Second, Soar organizes its tasks in terms of multiple *problem spaces*, which comprise sets of *operators*, which are applied to attain a *desired state* within the space. Operators executed in one space are performed in an implementation subspace (with its own operators). Thus, capabilities are evoked by the selection and application of operators, and capabilities are exercised by operator applications in some collection of spaces. This task organization is implemented by a scheme of immediate recognition of patterns of elements in a working memory, realized as an OPS5-like production system. Thus, an operator that does not require implementation in a subspace is one that is applied simply by immediate recognition, i.e., by the evocation of a set of actions directly in response to detected patterns in working memory. Soar has an automatic learning mechanism (chunking) that continually creates new productions (new recognition patterns with their immediate actions) that capture the processing performed in subspaces, thus avoiding its recurrence.

Third, Soar, as it normally operates, moves through the three higher levels of skill: instruction interpretation, elementary deliberation and direct recognition. The movement from deliberation to recognition is the automatic effect of the basic chunking mechanism described above. All deliberation produces additional recognition skill. The movement from instruction to deliberation (often directly to recognition) occurs because of general strategies that Soar employs in interpreting instructions, which cause chunking to learn the right knowledge to increase the skill level. These mechanisms comport exactly with the view in Figure 3. If knowledge is directly available, response is immediate at the recognition level of skill, but there is essentially no flexibility of response. If a solution requires search of problem spaces with directly available operators, response is slower at the deliberation skill level, but there is much more freedom to bring relevant knowledge to bear and be flexible. Interpretation of instructions is even slower, since knowledge must be accessed by working in a subspace that corresponds to the interpretation process, imposing in effect another inner-loop of activity.

These mechanisms and methods in Soar affect only the three highest levels of skill. Nothing in the current Soar deals with whether to use instructions, model-based simulations or theory-based derivations as the basis for a capability, or how to structure the processing for each of these levels of skills. That Soar provides for some changes in skill level has two effects. On the positive side, it makes clear that varying levels of skill is a real consideration in agents, not just something imported because we know humans exhibit varying levels of skill. It justifies the complexity of Figure 3 as relevant for the development of agents for ESSs. On the negative side, other mechanisms and methods exist for moving between skill levels, which would occur in other agents. None of that variety or its effects shows up in our considerations here.

Fourth, Soar communicates with ESSs through its regular input/output facility. The external environment is perceived by the arrival of elements placed by sensors in Soar's working memory, and the environment is affected by central cognition placing elements into working memory to trigger the actions of effector processes. I/O in Soar is asynchronous, so Soar does not have to enter a busy/wait state while waiting for input to arrive or output to be transmitted

and can deal with multiple external devices simultaneously. Because Soar's inner operational loop is the innmcdiafr recognition of whatever is in working memory, Soar can be strongly interrupt driven, reacting immediately to new input and shifting its attention to deal with the novel situation.

The following subsections describe Soar/ESS interactions, with the primary investigator for each project given in parentheses following the name of the effort in the title of each subsection.

4.1. Soar/IBDE (David Steier)

We have already described the EBDE ESS situation (Figure 1) and it is evident in a general way how all the basic capabilities are required IBDE actually provides a good example of needing to model the ESSs software, since a large specialized system organization is needed to integrate very complex individual ESSs.

The IBDE arrangement brings to the fore several aspects. First, because there are many ESSs, formulate-subtask must be concerned with the selection of which ESS to use at a given time. Actually, this demand goes further than just selection, since it is useful to plan out an entire computational sequence of the ESSs. Second, the data that flows between the ESS modules in EBDE consists of very large data structures that describe various physical structures for an entire building. These are fundamentally incomprehensible to the agent (i.e., Soar) and, indeed, to the human designer. They are simply too big to be read in and assimilated Thus, the strategy (for agents and humans alike) is to analyze the results of the ESSs at one remove. Additional critic ESSs are formed that process the output of a given IBDE module and produce compressed evaluations that can be understood by the agent (or human). For example, one critic ESS could make an initial rough cost estimate of the building from an early output Another could check for the satisfaction of some overall constraint that cuts across modules and therefore is not computed within any module in isolation. Thus, the operation of the agent is entirely in terms of high-level abstractions about the main IBDE ESSs, not in terms of their direct input/output data. This situation leads to the third interesting aspect of EBDE, namely, that the agent (or human) operates in terms of a high-level model of the computation. This abstract model does not exist currently in explicit form, although the engineers involved in IBDE clearly think in terms of such an abstract model (with individual variations, no doubt). So there are two interesting aspects: what should such a high-level model be like; and how is the agent to acquire it

There is an initial version of Soar/IBDE, i.e., a Soar system that operates as an agent for IBDE.[1] The current implementation acquires tool selection knowledge from model-driven simulation in a fashion alluded to earlier, the system models the functionality of the seven EBDE systems in terms of the types of inputs and outputs. This initial Soar/EBDE already illustrates that learning will be an integral part of every agent-ESS system. We have shown a significant reduction (75%) in processing effon due to learning, the skill-level shifting from the elementary decision level to direct recognition.

[1] **Its original name was Soar/FORS, because it initially was to operate through an experimental interface** system **called FORS [10].**

4.2. Soar/database (Bob Doorenbos)

Soar has recently acquired a capability for querying a relational database using SQL (Structured Query Language, a standard query language) to obtain information for use in other tasks. One can see clearly in the organization of Soar the four capabilities of the basic cycle, just as described in Section 3.1. In some task space Soar executes an operator that requires some data for its result The space that implements this operator exercises the formulatc-subtask capability by creating an internal representation that indicates the unknown data it wants in the context of the situation it believes that data depends upon. In this formulatc-subtask space it applies an operator to make a database query. This operator is implemented in a separate problem space (the SQL space). It contains operators that know the table structure and the semantics of the database. These operators extract from the representation provided by formulate-subtask the data to be used in the searches. Other operators know how to form SQL queries, including forming complex joins. Thus, this space is exercising create-input. Because of the complexity of SQL (i.e., the alternative ways it provides for searching the database), this capability is more than just conversion; it approaches creating simple programs. Finally an operator in this space sends the query to the database via Soar I/O and receives the data in return (as a data stream). The results of interest arc picked out of the returned data and inserted in the task representation. This occurs by another operator in the SQL space, so that this one space exercises both create-input and convert-output capabilities. As in most database uses, converting the result is routine and the processes that set up the query know exactly what should be done with the returning data.

The Soar spaces and operators for doing create-input and convert-output can be viewed as a general capability for Soar to deal with relational databases, at least of the standard variety. It is coupled with a general set of representational conventions for how formulate-subtask must indicate what data it wants. These conventions, however, relate to the way Soar systems in general represent their problem-solving states. It contains no knowledge about SQL or the specific structure of the database. It does of course contain some general knowledge on the pan of formulate-subtask of the content of the database or it wouldn't even know to attempt to use it.

Despite the view of a general capability, provided once and for all as part of the capabilities that Soar should offer all users, this capability cannot be programmed once and for all by a Soar system-designer. For instance, as Soar uses databases by means of these capabilities, it learns to do so faster. There is a continual shift from the elementary-deliberation skill level (selecting and applying operators in spaces) to the direct-recognition skill level. Some of this learning is quite general, and speeds up Soar cm all uses of SQL. But other parts of it are specific to the particular database and provide speedup on subsequent uses of it (preliminary experiments show about a factor of two for the combined effects — they haven't been separated out yet). These speed-ups are important for Soar to use databases effectively.

Databases pose in clear form the issue of the simulate-ESS capability. Suppose, in some circumstances, an agent can know the results from a database query without actually having to query the data base. Can it make any use of such knowledge (except to avoid the time delay of the query)? On the one hand is the programming intuition that if you can get the answer from one system (the ESS) of what use could another way of obtaining the same answer be (except speedup)? On the other hand is the intuition about humans that familiarity with a database is a requirement for its intelligent use. Perhaps one shouldn't think of simply obtaining directly the exact data in the database; rather, simulate-ESS would contain a more abstract or generalized

model of the contents of the database. The Soar database research is too recent to provide answers, but the obvious hypothesis is that such knowledge permits more efficient or discriminating queries.

4.3. CPD-Soar and Interval-Soar (Ajay Modi)

CPD-Soar is a system for designing chemical separation systems, collections of distillation columns that split an input volatile fluid containing several chemical species into several fluids that (ideally) each contain only a single chemical species. To evaluate candidate designs, it will use ASPEN PLUS [1], a chemical process simulator, to simulate the results of a single distillation column. ASPEN PLUS is a complex program, the result of many years of research and development, so its calculation cannot be done independently by an agent (Soar). This is one typical form of ESSs: a unique, practically nonreplicable repository of expertise, which must be accessed as an ESS by both agents and humans.

The Soar-ESS system here would appear to be quite simple: Whenever an evaluation is needed (which is relatively straightforward to determine), provide the inputs to ASPEN PLUS, execute it, and get the evaluation number back. There might be some number conversion in create-input and convert-output, if Soar and ASPEN PLUS use different number representations (which they do), but that should be it.

Additional considerations enter because ASPEN PLUS is very expensive. The basic structure of CPD-Soar is a combinatorial search through the space of candidate separation systems, using evaluation functions to prune the search. To use ASPEN PLUS for every evaluation is to completely determine the design process by minimizing the number of evaluations performed. Thus, there is a premium on finding some way to obtain cheap evaluation functions, even if they were much more approximate (ASPEN PLUS produces high quality answers). One path towards this is for CPD-Soar to learn from its uses of ASPEN PLUS enough to be able to produce internally some useful approximation to what ASPEN PLUS would produce. Here we see a specific reason why an agent should have a simulate-ESS capability. The most obvious way to do this is simply to remember that to the input (U, V, W) ASPEN PLUS produced the result Z; and indeed Soar chunking automatically provides this level of learning. The trouble is that this exact input almost never repeats, so this learning does essentially no good at all.

Interval-Soar is an attempt to explore an hypothesis about learning to simulate ESSs. Namely, what an agent learns from the operation of an ESS depends on what model of the ESS the agent brings to the experience. If the model is only that the ESS produces function values (numbers in, numbers out), then, per above, that is all that can be learned. If the model of the ESS is that it provides information about the shape of an evaluation function, then the agent can learn about this shape — and that might be enough to make some internal computations that would permit Soar to bypass using ASPEN PLUS. Interval-Soar employs an extremely simple and highly abstract model of the evaluation, namely, that it is a unimodal function. Then a sequence of uses of the ESS evaluation, permits locating the mode of the function with increasing precision. Since evaluation is being used by Soar to compare candidate designs, only the relative order of two evaluations is needed. Often this can be determined by the partial knowledge that has been built up about the value and location of the maximum of the evaluation function. More detailed a priori models of the evaluation could, of course, be used, changing the knowledge that could be learned from the execution of an ESS. We have an additional goal in this research, which is to

show **that** the **actual learning,** given the model of the ESS, occurs by chunking, Soar's basic learning, **mechanism.**

4.4. Soar/Mathematica (Dhiraj Pathak)

We are attempting to get Soar to be able to use *Mathematica* [12]. This differs from the systems described above in requiring **a** relatively long sequence of interactions between the agent and Mathematica, each fairly small, to obtain a useful result Thus it provides some experience along a major dimension of variation of ESS systems. Formulate-subtask must produce a computational plan, and the other capabilities of create-input, convert-output and interpret-result can be conceptually lumped together into the implementation process for the plan. Except when the agent is quite expert, the initial plans developed by formulate-subtask will be quite incomplete, because the agent will not know enough about how to do in Mathematica what it wants to do mathematically to lay it all out in advance — this is what it means to do experimental programming. So interpret-results will end up being a fairly intelligent diagnostic process (unlike the database case).

One issue that has shown prominently in the work with Mathematica can be called the *transduction fallacy* — the belief that the only capabilities involved in working with an ESS are transactions between the representations of the agent and the ESS, namely, create-input and convert-output (and indeed their most elementary functions). The arises in clear form with Mathematica because in attempting to pose an example or trial task for Soar/Mathematica, one necessarily couches it in some mathematical notation. Then it appears that all that is to be done is to convert from this mathematical expression (which may be very different from the notation and style of Mathematica) into the notation of Mathematica. After a few tries one cannot think that any other capabilities will be involved except transductions. The fallacy, of course, is that we, the ones who are inventing demonstration tasks for Soar/Mathematica, are doing the formulate-subtask ourselves, and leaving just the transduction to Soar. Only when we give Soar a larger analysis task to do, within which the use of Mathematica arises as a potential means, does it become clear that other capabilities are involved.[2]

We have seen enough examples where more than transduction is involved to not be taken in by the fallacy. Yet it seems to haunt the exploration of Soar/Mathematica in interesting ways. No matter how far back we go, in terms of Soar's task-oriented problem spaces, the initial form of the task is cast in some mathematical representation. It almost seems as if the primary task must be to convert from whatever is this internal Soar notation into Mathematica notation. We have been exploring some protocols of humans using Mathematica to see what tasks get formed early on. Many activities go on other than simply writing down mathematical expressions (in whatever notation). For instance, there is exploration of Mathematica to find out how to do something with it (so there is acquisition of capabilities through experimentation). Also, there is the fonnulation of the mathematical plan in essentially method-like or qualitative terms, before

[2] **The transduction fallacy also tends to arise in working with Soar/database, where the only way one can imagine to use Soar/database is to have a user make some, perhaps high-level, request of Soar to find some information — at which point there only remains the transduction into SQL from whatever language the user used to make the request. In our experimental work with Soar/database, we use an overall task that makes Soar determine what information it needs and when it needs it**

any expressions are created

Still, transductkm between mathematical notions is an activity that always seems to loom large. This sugfMts asking what would happen if Soar cast *all* its mathematical expressions in the notation of Mathematics After all, Mathematica (and other modern symbolic-mathematical systems) are intended to be close enough to natural human mathematical notation that humans are expected to come to think in its terms. How much simpler will a Soar system be if it thinks entirely in terms of Mathematica? Will it really make a lot of difference (one is inclined to think so, analogizing after the human phrases about "thinking it is terms"). But could the continual learning of the notation conversion processes by means of chunking, simply wash out such differences?

4.5. MFS: Model-formuIation-Soar (Xiaoping Li)

MFS is a Soar system that supports mathematical-model formulation in the area of operations research, where one goes from a qualitative description of a problem to a set of mathematical expressions that can be given to a standard mathematical programming package for optimization, such as LINDO[11] or GAMS [2]. MFS currently formulates mixed-integer linear-programming models of production-planning problems, and is at an early stage of development (e.g., it does not yet propose and retract assumptions to perform iterative model refinement).

It appears that MFS will use four functionally distinct ESSs: (1) A database that holds the massive given information about the model, e.g., background constants, generic process equations; (2) a statistical package for estimating parameters to functional forms from data; (3) an external memory to hold all the variables and mathematical expressions in the developing candidate model and (4) the mathematical-programming package (e.g., UNDO) that is to solve the model. The use of the database and the statistical packages do not appear special, in that the ESS issues are those already raised in discussing Soar/database and Soar/Mathematica. But the last two ESSs each have their special interest

The models for industrial application are often very large in terms of the numbers of variables and constraint expressions. Maintaining the full candidate model in working memory imposes a large processing burden on Soar — for the patterns in Soar's recognition memory continually survey the entire working memory to find matching data (the power of recognition memory lies in such complete and automatic surveillance). Mostly, the attention of MFS is focused on some part of the candidate model, and the remainder can be safely ignored. One way to realize this is to construct an external working memory, i.e., an ESS whose function is to hold the variables and expressions of die candidate model (or several of them, if alternative candidates are simultaneously under consideration). This is analogous to a database, except that storing changes in it and accessing information from it should be highly tailored to the structure of the model components, so that they can be highly efficient So it probably requires a special design. An interesting aspect of this ESS is that its need is not usually foreseen in the early stages of building a Soar system (here MFS), but emerges as the system is scaled up. The issue then arises of rapid acquisition of the ESS, i.e., acquiring the basic performance capabilities to deal with the new ESS. An essential pan of this is reorganizing the existing Soar task spaces to use the ESS, which requires introducing a fonnulatc-subtask capability that fits in with the pre-existing performance organization. If such reorganization and coupling with new ESS is a major enterprise, it is unlikely ever to be undertaken. Thus, speed and ease of acquisition become

essentiaL

The final ESS it the optimization programming package itself. MFS will submit completed models to the package for solving. But long before this happens MFS will have candidate models that it believes should work, but which in fact contain conceptual errors of various kinds. The models are large and complex, and they cannot be created correctly on the initial try, no matter how much advance intelligent analysis occurs. Thus, the main MSF interaction with the computational package will be in a conceptual debugging loop, in which strange things happen when the model is submitted to the package. Initially, these may be like syntax errors, many of which the package itself will catch and return appropriate error and warning messages. Later in the development, the solutions will appear to be comet, except they will be unrealistic, which must be diagnosed both in the output from the solver and then in the model in terms of what caused the strange result Thus, interpret-result will be the capability that will require the most development for this ESS.

4.6. Draw-Soar (Gary Pelton)

Draw-Soar takes in natural-language descriptions of a set of spatially structured and related objects, such as, "There is a tangential line at the top of the smaller circle/ and produces a set of commands to a drawing system (the ESS), such as McDraw, to draw the corresponding picture on a graphic display device.[3] The interaction with the ESS is two-way and fine-grained, and uses the command-language of the ESS. The picture is composed incrementally, as enough information accumulates to define another part of the picture. The system, as currently envisioned, does not itself have visual capabilities for seeing the developing picture whole. Thus, it is also in interaction with a user, which must be its eyes (this mixes the role of observer as client and as critic, which may not be the best way). Hence, feedback comes through additional natural-language dialog. The current system is working open-loop, just beginning to process simple requests.

Draw-Soar provides another point along the dimension of the time-grain of interaction, even more fine-grained than Soar/Mathematica. It can be expected that this will put a great deal of emphasis on developing the recognition skill level, since so little gets done with each command-interaction.

Draw-Soar provides an interesting opportunity to understand the role of simulate-ESS. The ESS, being command-language oriented, admits of a model with a very simple structure — to each command there is an effect — although the effect can be highly context dependent (a change in the geometric display, whose visual effects can be profound). Thus, the agent can develop a range of models of the ESS at different degrees of abstraction and adequacy. The quality of the final result — the final drawing in relation to the original description — depends strongly on the simulate-ESS capability, for any but the simplest pictures. So Draw-Soar should be quite revealing in this respect

Strictly speaking, the three-way interaction between Draw-Soar, its drawing-package ESS, and a human user, who sees the output of the drawing package and has two-way communication with

[3]Draw-Soar makes use of a developing capability for natural-language comprehension in Soar, NL-Soar [8].

Draw-Soar, is outside the paradigm we set ourselves for this paper, namely, the single-agent multiple-ESS situation. But it provides the opportunity to explore a number of issues in the acquisition of capabilities. The communication channel between the user and Draw-Soar is in natural language, a type of knowledge source that is important to learn to exploit But the arrangement can be used in many ways. The user could initially describe figures in terms rather close to the command language of die drawing package, and. then gradually move toward freer descriptions, as Draw-Soar's capabilities increase. The user could also instruct Draw-Soar directly about the drawing package and how to use it Arrangements, using the visual feedback from the user to Draw-Soar, could be set up for Draw-Soar to learn by experimentation or by observing the user doing things directly. These variations need not stress the natural language capability, employing only a small range of sentences; rather, they exploit the flexibility of natural language to express different sorts of situations.

4.7. EFH-Soar: Learning from Electric Field Hockey (Jill Lehman)

The final Soar-ESS system arises from an analysis of humans using interactive educational systems. The construction of the system has not yet commenced; work is still focused on psychological aspects. However, it raises an additional ESS issue that is worth mentioning.

The scientific problem is to discover what human students learn about some subject matter by the use of an interactive computer microworld that behaves according to the subject-matter field and is structured so that students can explore the subject matter via die microworld. Many such microworlds are being developed in the application of computers to education. The microworld under investigation is Electric Field Hockey, a computer game built to educate about the physics of electric fields. The student places positive and negative charges in a rectangular arena, creating an electric field; then a charged particle (the puck) is released at one end, the aim being to have it travel into the goal box at the other end. In more complex variations of the task, barriers are placed in the arena, so the puck has to make its way around or between the barriers. If the student understands how electric fields are determined by the distribution of charges, then he or she can place charges so as to win the game.

The Soar system is to be a simulation of a human student Thus, it is a single agent (EFH-Soar) with a single ESS (the Electric Field Hockey computer game), but the agent here is to model the student, so we can discover how humans learn in such situations. The interaction is again at the fine-grained end of the scale — students often operate in a highly interactive way, incrementally pushing around the charges already placed in the arena.

The interesting aspect from the agent-ESS viewpoint is the nature of the learning required to be successful* The student (EFH-Soar) should be acquiring general knowledge about electnc fields, no longer intertwined with the game of Electric Held Hockey ESS and its details. It should be the acquisition of the subject-matter knowledge that permits the student to be successful at the game. All the learning issues we have raised — the acquisition capabilities in Figure 4 — are focused on acquiring a skill for dealing with the ESS, i.e., for dealing with Electric Field Hockey. For Electric Field Hockey acquisition must go via an outer loop through knowledge that is independent of the ESS, and then back to developing skill in using the ESS.

The claim above is not ours. Rather it is implicit in the use of interactive microworlds for educational purposes. Our objective is to explore this claim and discover the extent to which it

holds. Along the way some interesting light may be shed on what is and can be acquired in the agent-ESS situation more generally.

4.8. Summary of the issues

The issues raised about the control of ESSs by these seven Soar efforts are in many ways a congeries. The systems themselves have been generated for independent reasons, and their involvement with ESSs has been dictated by their inner demands, not by any concern for how to conceptualize agent-ESS research issues. Nevertheless, we can attempt some summary of the issues that have arisen in this section.

1. **The space of ESSs.** Some of the dimensions along which ESS systems vary are already clear. The main one seems to be the *grain size* of interaction, indicated by the number of interactions between agent and ESS to get a formulated subtask accomplished. This dimension is coupled both with speed of interaction and with how much planning or programming has to be done (because meaningful tasks require sequences of interactions). Hence, it is not yet clear what comprises the essential characteristic of this dimension. A second dimension is the *number* of ESSs involved, and the extent to which their functionally overlaps, so what ESS to use requires decision. A third dimension is how *dynamic* the ESS situation is, in terms of new ESSs coming on line or changes in existing ESSs. The more dynamic, the greater the pressure on acquisition capabilities and the speed with which they must be accomplished. A final dimension is the degree of *indirection* with which the agent deals with the inputs and outputs of the ESSs. Does it understand the full semantics of the input/outputs (as in Draw-Soar) or does it have only some abstract model of the ESSs (as in IBDE/Soar)?

2. **Different performance capabilities are key to different ESSs.** In at least minimal form, all four basic performance capabilities (Figure 2) are involved in each ESS transaction. But beyond this we saw very different emphases. Figure 5 presents a table that shows for each of the Soar-ESS systems, the capabilities that play a prominent role in the given system. As can be seen, formulate-subtask shows prominently in Soar/IBDE, Soar/Mathematica and Draw-Soar. Create-input shows prominently in Soar/database and Soar/Mathematica (though the latter depends on some of the programming in Mathematica being part of create-input). Convert-result does not seem problematical for any of our situations. Interpret-result shows prominently in Soar/IBDE and MFS. Simulate-ESS shows prominently in CPD-Soar, Soar/database and Draw-Soar. Operate-software-system was entirely absent, because our agent-ESS systems have not matured to where this capability becomes a necessity.

3. **The pervasiveness of learning.** Learning capabilities show up essentially everywhere. Figure 5 also reveals this, both by the multiple skill levels that are primary to each system, which implies movement between levels, and in the different knowledge sources that play a role, implying acquisition. In the case of immediate performance, automatic skill acquisition for create-input and convert-output already plays an important role in Soar/IBDE and Soar/database, and will in all other systems as well. Learning simulate-ESS is the heart of CPD-Soar with Interval-Soar. The MFS situation shows the need to acquire the four performance capabilities for a new ESS (an external working memory). We could have used Soar/Mathematica to make the same point, namely, the need to extract quickly from an agent a computational function being performed internally, thenceforth to

be provided by an ESS. The acquisition of performance capabilities by drawing knowledge from the full range of available sources (Figure 4) is lying just below the surface in Draw-Soar and EFH-Soar.

4. A **few emtrging** issues. Not many sharp research issues have emerged yet, but perhaps two are worthy of note. The first is the transduction fallacy, namely, that the central problem of dealing with ESSs lies in create-input, convert-output and the operating system that mediates between agent and ESSs. So long as this view guides the interpretation of the essential nature of agent-ESS systems, the necessary research to construct effective agents for ESSs will not occur on the more intelligence-demanding capabilities of formulate-task, interpret-result and all the varieties of acquisition. The second issue is the possibility of the agent coming to think in terms of its ESSs, that is, to migrate the representations used by the ESSs back into the formulation of its own tasks. Then much of what has to happen in create-input and convert-output disappears. More exciting, the agent and the ESSs become, so to speak, impedance matched, so that the formulation of subtasks becomes more likely of success. How this actually works out is all future research, but it is clearly an issue of general import for agent-ESSs systems.

5. Agent-ESS Systems as a Component of Software Technology

The effort to develop agent-ESS systems is important for AI as **a** relatively unexplored area of intelligent action that is saturated with learning considerations. But its major contribution may ultimately be to software engineering, in making the power of computer software more easily accessible in the service of computational tasks. The type of system under discussion — a single agent with multiple ESSs — has an immense scope for application. This is indicated already by the collection of examples described above. Each example epitomizes an entire class of software systems of applied interest From the large scale collection of tools (represented by Soar/IBDE), to the routine intelligent use of databases (represented by Soar/database), to the use of highly interactive systems (represented by Draw-Soar) — each Soar-ESS system points to a population of application systems. That agent-ESS arrangements have important application potential justifies developing the requisite scientific knowledge base, even if some of the capabilities seem distant from immediate application.

Viewed generally, agent-ESS systems belong to the class of software systems that make the software system smarter to improve system effectiveness and software productivity. With the exception of a relatively small community at the interface between software engineering and AI [9], this tactic has not been widely pursued to date. Some areas however should be at least noted

One relevant **area** is that of intelligent interfaces in Human Computer Interaction [6]. Here, the attempt is to develop human-agent-ESS systems, rather than just agent-ESS systems — where a human uses a collection of ESSs via an interface that is an agent, i.e., an intelligent interface. This places most of the intelligence of the total system in the human and casts the agent in the role of aide, guide or facilitator. As noted earlier, this is the form a mature Soar/IBDE would undoubtedly take. Research in intelligent interfaces complements the agent-ESS paradigm. In the intelligent-interface work, the focus is on the interaction of an agent with the human user — how to make that communication intelligent. In the agent-ESS work, the focus is on the interaction of an agent with the software systems to be used Both of these types of interactions

*. Primary resevch concern
o *m* Secondary research concern

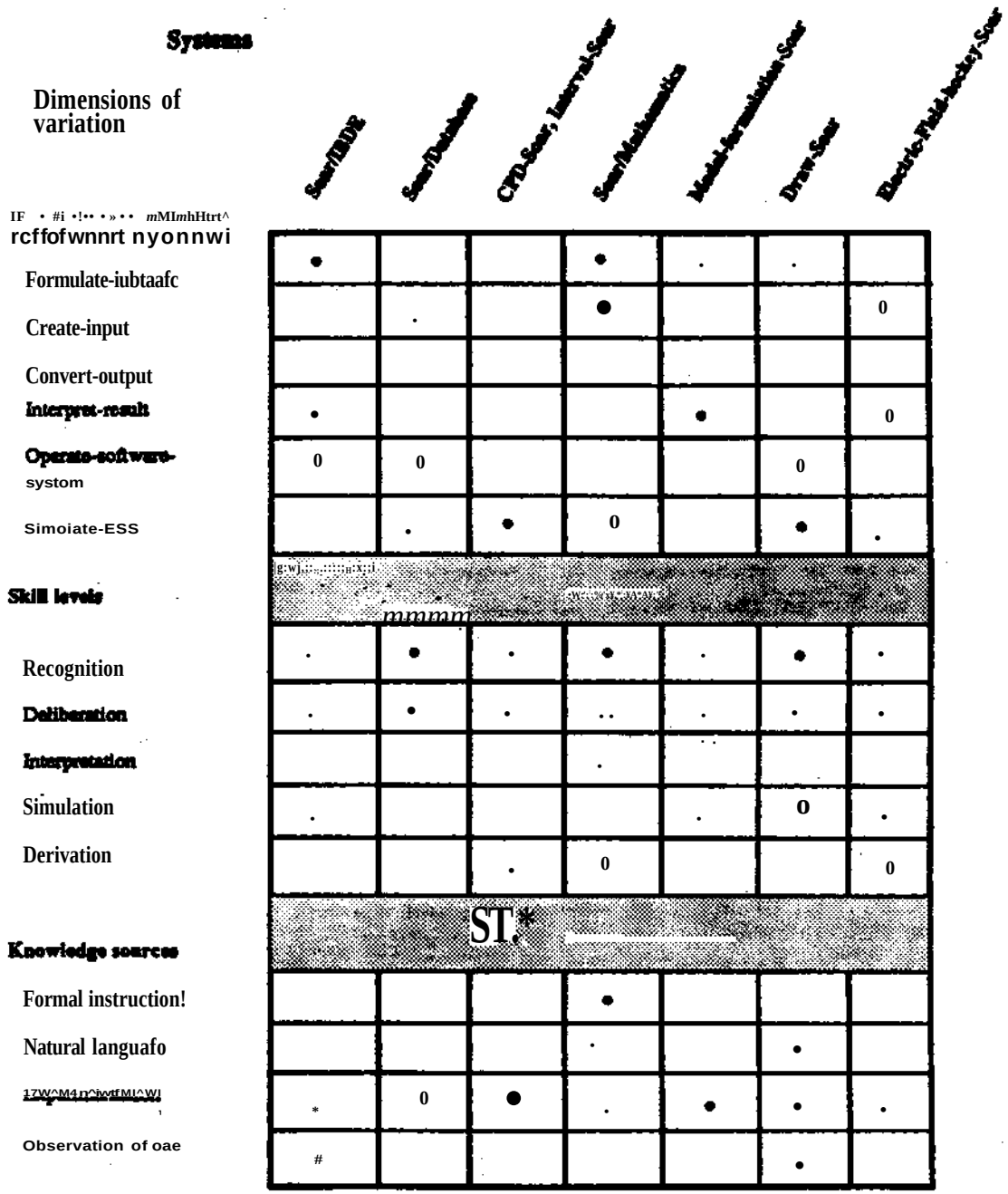

Figure 5: Summary of capabilities that are prominent in the Soar agents.

need to be understood and developed into effective technologies, so they become available as options, in the software engineering of large systems.

A second area of potential relevance is that of distributed artificial intelligence [5], which is concerned with collections of cooperative agents. Again, we see that the basic situation differs from that of agent-ESSs. At the center of attention in distributed AI is agent-agent communication and collaboration. Issues of negotiation, contracting, division of labor, inconsistency of knowledge, diversity of goals, etc., become central. Some of these may have a pale reflection in the agcnt-ESS situation, but basically the passive nature of the ESS removes these issues from center stage. Instead, the focus becomes coping with the input/output representation of the ESS, formulating the agent's task in terms that permit the ESS to provide help, acquiring models of the ESS, etc. — as described throughout this paper. Distributed AI is attacking a more complex set of issues. A mature agent-ESS technology would probably contribute to the substrate on which eventually to build useful distributed AI systems. But there does not seem to be much immediate connection in the other direction, from research in distributed AI to agent-ESS systems.

The agent-ESS situation should be viewed as one strand in an expanding conception of software technology — a tool in the total kit of techniques for engineering software systems. Actually, it adds at least two tools, and contributes to a third The first relates to the aim of software-technology research to reduce the amount of effort required to specify some computation to be performed. The evolution has been from requiring the user to be familiar with all the details of the hardware and software implementation, i.e., machine code, to building in increasing amounts of abstraction and having compiler-like software bridge the gap back to the fully specified software system that executes the application. When taken to the limit, the scenario for completely automatic programming has users specifying computations exclusively in terms natural to their application domain. In the long run, an intelligent agent equipped with the capabilities we have described provides an additional radical strategy for moving towards this goal — to wit, having the user deal with agent-ESS systems in abstract terms, because the agent deals with the concrete details of the ESSs that are the applications. Functionally, this casts the agent in the role of the "compiler-like" software, except that the agent engages in quite different operations, namely, the array of performance and acquisition capabilities we have outlined in this paper, which results in being able to adapt existing software to new uses. This, of course, does not substitute for compilation-like techniques (nor for interpretive ones either). Rather, it adds a third tool to the software engineering toolkit, which will be the preferred technique in many cases.

The second role relates to reusability. The power of existing software systems, which otherwise might have to be reimplemented or at least integrated by a programmer, can be made accessible through an agent-ESS arrangement. The reach of this technique — how inhospitable an existing collection of ESSs might still be made useful — would seem to depend strongly on the acquisition capabilities of the agent The whole point of this tool is to shift to the agent the burden currently borne by the human system programmer, who must normally refurbish the software system through extensive efforts to understand it and reprogram it.

The third (contributory) role relates to the potential of intelligent interfaces, mentioned above, to become an integral part of large software systems. Those efforts, as noted, focus on human-agent communication. But they will only become effective for real applications, if the agent-

ESS side of the systems is equally well developed. The area of intelligent interfaces is not itself likely to provide that development, because the research issues that are central (and properly so) to human-agent communication are quite different than those of agent-ESS operation. So there is a role for an autonomous agent-ESS development, with the expectation that it will provide the other half of what is needed for intelligent interfaces to be useful.

It is worth noting, finally, that ESSs and collections of them can comprise large software systems. The issues for agent-ESS systems do differ for simple systems versus complex systems. That can already be seen by comparing Soar/database with Soar/IBDE, which might be taken as representing simple and mid-range points along the dimension of system complexity and size. But both these agent-ESS systems are feasible, and both fit within the conceptual framework we have outlined in this paper. Thus, as tools for software engineering, agent-ESSs should be viewed as potential contributors to the large-system end of the software-system spectrum, which is where software engineering is most in need of development.

6. The Research Agenda

The considerations we have put forth can be summarized in a few points:

1. Artificial agents capable of using external software systems effectively will require a substantial array of sophisticated capabilities, including learning capabilities and the ability to deal with varieties of external knowledge sources.
2. These capabilities stretch from those that are central to the programming-system parts of computer science (the operating system capabilities, even though we did not stress them here) to those that are central to artificial intelligence.
3. We know very little yet about the details of the required capabilities — what is required to make them work together, what it takes to handle ESS situations of realistic size and complexity and what it takes to learn them.
4. The potentiality for application of agent-ESS arrangements is apparent. If developed, these arrangements would provide additional modes for effectively organizing large software systems in many situations.

The exploration and development of capabilities for agent control of ESSs has become an item on the research agenda of the Soar project, because many individual Soar systems have begun to require such capabilities in one way or another (as their enumeration in Section 4 indicates). This paper is our attempt to identify this arena as a fruitful one for focused research.

An extensive array of highly sophisticated capabilities appears to be required for effective agent-ESS systems. This may seem to add up to an impossible agenda — as if no progress could be made on providing effective agent-ESS systems until all the problems of AI have been fully solved. That does not seem to us the proper reaction. All research — including research in computer science and AI — moves forward incrementally, always tackling the next thing that seems possible. The existing Soar-ESS systems already show many lines of fruitful advance. It is relevant, however, that Soar is a highly sophisticated problem-solving and learning system, so we are not starting from scratch in building up the agent capabilities. Thus, moving up the skill level from deliberation to recognition has already been demonstrated in the existing Soar-ESS systems. This occurs because Soar already has substantial general learning capabilities. Likewise, NL-Soar (the natural-language capability in Soar) integrates completely with Soar

application systems (in contradistinction to the well-known difficulties of interfacing separate language and application modules). Thus some acquisition capabilities can already be approached in at least elementary form (Draw-Soar provides the clearest example). It is also relevant to the feasibility of ESS research that ESSs form a highly specialized class of systems. Although some capabilities benefit little from the specialization (natural language comprehension would seem to be an example), most of the performance and acquisition capabilities will be much simpler for ESSs than for general physical, chemical and biological systems. Software systems, being constructed from sequential programming languages and discrete data structures have relatively simple structure with good abstractions. The beneficial effects of these simplifications are readily seen in the cuirent Soar-ESSs system, e.g., Soar/database and Draw-Soar.

In sum, our own reaction to the requirement for a vast array of highly sophisticated capabilities is that it makes clear that agent-ESS systems are more than just getting the agents and ESSs hooked up at the software level with appropriate data-conversion software (the transduction fallacy). We see the array as providing a map for where we have to go.

Our research approach will be composed of the following activities:

1. Empirical exploration: The simultaneous development of diverse systems (such as the collection described above) that reveal the different capabilities needed and provide specific enough contexts to construct instances of such capabilities. These systems, which are independently motivated applications of Soar, also provide the test situations to determine whether we have developed an effective agent for controlling ESSs.

2. Focused search: The selection for development of new systems that focus on specific capabilities that need exploration and that fill out our total picture of the full array of capabilities.

3. Human capabilities: The analysis of how humans perform the same tasks using ESSs as do our Soar systems, in order to obtain clues to the additional capabilities and knowledge that humans have, of which we are currently unsuspecting. Such studies also provide benchmarks against which to calibrate agent performance.

4. Generalized capabilities: The recasting of the specific instances of the capabilities as general abilities that can exist in Soar at all times and be available in any Soar system that needs to make use of ESSs. Such generic capabilities become the starting points for the development of specialized capabilities for particular ESSs or classes of ESSs. Progress on this research activity awaits seeing diverse, multiple instances of the different capabilities, so common structure and operations can be discerned.

With this agenda we expect no difficulty in determining whether or not the research is making progress. It is relatively unambiguous whether a given agent-ESS system is *adequate* — either the agent can use the ESSs to help it do its tasks or it can't. Performance metrics are sometimes helpful in evaluating the individual agent-ESS. How skilled is the agent in using this ESS? Or, to rephrase it, how large a fraction of the agent's effort goes into operating the ESS? Suppose a Soar system uses a database as part of its operation (by having the capabilities of Soar/database). If 80% of its time is occupied by SQL programming (create-input), the agent-ESS performance is pretty poor. In some cases, the norms for performance come from analyzing how fast the

agcnt-ESS component must be so the total system can perform the total task satisfactorily. In other cases, human performance provides revealing comparisons.

Basically, however, the evaluation of an individual agem-ESS situation is simply one of adequacy. The important metric for evaluating research in this area is the *expansion of scope*. What agent-ESS situations can be handled adequately now and how is this growing? Draw-Soar can, say, do simple drawings — can it now produce drawings of the complexity of figures in publications? Soar/database can, say, use relational data bases employing SQL - can it now acquire a database that uses some other query language, not necessarily more complex than SQL but just different? Thus, progress is to be gauged by a sequence of challenge tasks, each posed to force a substantial, but attainable, increment of development of a given agent-ESS system (or class of them). As this sequence evolves these challenge tasks include real aoolicatinrK applications, providing additional calibration and measures of success.

References

[I] **Aspen Technology,** inc.
Aspen **Pins User** Guide to Release 8.2.
1988.

[2] Brooke, A., Kendrick, D. and Meeraus, A.
GAMS: A user's guide.
Scientific Press, Redwood City, CA, 1988.

[3] Caidozo, E.
***DPSK: A kernel** for distributed problem solving.*
PhD thesis, Carnegie Mellon University, January, 1987.

[4] Fenves, S. J., Hendrickson, C, Maher, M. L., Hemming, U, & Schmitt, G.
An integrated software environment for building design and construction.
Computer-Aided Design 22(l):27-36,1990.

[5] Gasser, L., & Huhns, M. N. (editors).
Distributed Artificial Intelligence.
Pitman/Morgan Kaufman, London, 1989.

[6] Hancock, P. A., & Chignell, M. H. (editors).
Intelligent Interfaces: Theory, research and design.
North Holland, Amsterdam, 1989.

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

[8] Lehman, J. F., Lewis, R. L., & Newell, A.
Natural Language Comprehension in Soar: Spring 1991.
Technical Report, School of Computer Science, Carnegie Mellon University, March, 1991.

[9] Mostow, D. J.
What is AI? And what does it have to do with software engineering?
IEEE Transactions on Software Engineering SE-11(11):1253-1256, 1985.

[10] Papanikolopolous, N.
FORS: Flexible Organizations.
Master's thesis, Carnegie Mellon, 1989.

[II] Schrage, L. & Cunningham, K.
Demo UNDO/PC: Language for Interactive General Optimization.
LINDO Systems Inc., Chicago, IL, 1988.
version 1.04a.

[12] Wolfram, S.
Mathematica - a System for Doing Mathematics by Computer.
Addison Wesley Publishing Co., 1988.