Toward a Methodology for AI Architecture Evaluation: Comparing Soar and CLIPS

Scott A. Wallace and John E. Laird

Artificial Intelligence Laboratory
University of Michigan
Ann Arbor, MI 48109, USA
{swallace,laird}@umich.edu

Abstract. We propose a methodology that can be used to compare and evaluate Artificial Intelligence architectures and is motivated by fundamental properties required by general intelligent systems. We examine an initial application of this method used to compare Soar and CLIPS in two simple domains. Results gathered from our tests indicate both qualitative and quantitative differences in these architectures and are used to explore how aspects of the architectures may affect the agent design process and the performance of agents implemented within each architecture.

1 Introduction

Development of autonomous intelligent systems has been a primary goal of Artificial Intelligence. A number of symbolic architectures have been developed to support the low level, domain independent functionality that is commonly required in such systems. Although some studies have examined what types of agent behaviors are most appropriate within a given domain (e.g., Pollack and Ringuette [12]), only secondary attention has been paid to the complementary problem of determining which computational primitives should be supported by the architecture and how they should be implemented.

Researchers attempting to develop new agents are faced with a difficult problem: should they construct their own symbolic architecture for the tasks at hand, or should they reuse an existing architecture? Because symbolic architectures tend to be large and complex, development requires a significant amount of time and effort. This is especially true if the resulting architecture is intended to be flexible enough for reuse in diverse situations. On the other hand, because of the lack of research examining how different architectures support a particular type of behavior, the selection process is often haphazard, which as Stylianou et al. [15] state, can lead to inefficiencies in the development and the resulting agent.

Our goal is to develop a methodology for evaluating symbolic AI architectures and to examine how differences in these architectures affect both the agent design process and the capabilities of the resulting agent. This information will allow developers to make more informed decisions about the suitability of an architecture to a specific domain, and will enable future architectures to fill specific needs more effectively. Moreover, the design of any agent–based system will profit from information that indicates

which computational primitives are most effective for a particular class of problems. This paper describes a methodology for conducting such evaluations and our initial experience applying the methodology to Soar (Laird [7]) and CLIPS [1].

2 Architectures, Intelligence, and Knowledge

The class of AI symbolic architectures we are interested in are those that support the development of general, intelligent knowledge–rich agents. Following Newell's description [10], an architecture is the fixed set of memories and processing units that realize a symbol processing system. A symbol system supports the acquisition, representation, storage, and manipulation of symbolic structures. An architecture is analogous to the hardware of a standard computer, while the symbols (which encode knowledge) correspond to software. The role of a general symbolic architecture is to support the encoding and use of diverse types of knowledge that are applicable to various goals and actions. Some examples of these architectures include: Atlantis (Gat [4]), CLIPS [1], O-Plan2 (Tate [16]), PRS (Georgeff [6]), Soar [7].

The basic functions performed by an architecture usually consist of the following (from Newell [10] p. 83):

- The fetch-execute cycle
  - Assemble the operator and operands
  - Apply the operator to the operands using architectural primitives
  - Store results for later use
- Support access structures
- Input and output

Architectures are distinguished by their implementation of these functions, and the specific set of primitive operations supported. For example, many architectures choose the next operator and operand by organizing their knowledge as sequences of operators and operands, incrementing a program counter to select the next operator. They also have additional control constructs such as conditionals and loops, but depend on the designer to organize the knowledge so that it is executed in the correct order. Other architectures, such as rule-based systems, examine small units of knowledge in parallel, selecting an operator and operands based on properties of the current situation. Architectures are often further distinguished by the inclusion of additional functions, such as interruption mechanisms, error-handling methods, goal mechanism, etc.

3 Evaluation Methodologies

As analysts, we are faced with a difficult issue, how should we evaluate architectures and their corresponding functions? A first approach would be to try to compare architectures based on which types of knowledge they can encode, or what functions they directly support, possibly finding tasks that one architecture can be used for, but another cannot. Evaluations that follow this approach often focus on high level categorical differences, which are usually motivated by examining the demands of a certain class of

application domains. In this type of evaluation, differences such as support for forward and backward chaining, the availability of a GUI, or the ability to maintain real-time commitments may be assessed (e.g., Gevarter [5], Lee [8], Mettrey [9], Stylianou [15]). Because most architectures are Turing complete, they can often achieve identical functionality by interpretation with the addition of knowledge. For example, hierarchical planning systems are likely to have special architectural constructs to support the semantics of goals and actions. Simple production systems, like OPS5, however, can be designed to implement the same behavioral strategies as hierarchical planners with the addition of productions that encode knowledge about the missing constructs. However interpretation has two potential costs: an increase in execution time, and an increase in knowledge required to specify the new behavior. On the other hand, adding functionality to the architecture itself is also likely to introduce complementary performance overheads. For example, an agent that is built upon a specialized hierarchical planning architecture may not be able to excise the superfluous components. As a result, it may end up paying a performance penalty in simple domains where hierarchical planning is not exploited. It is an empirical question as to which cost dominates, and the costs might be different for different mixtures of knowledge, goals, and possible actions. Therefore, our methodology will include measuring both the required knowledge and required execution time of different agents in multiple domains.

A second possible approach to architecture evaluation is to create a set of benchmark tasks and use independent teams of expert programmers to implement the task on each architecture. Evaluation occurs by directly comparing these implementations to one another (Schreiber [14]). Unfortunately, this approach can lead to a confounding between the contribution of the architectures and the knowledge that was encoded in them. For example, if we have two teams of robot soccer players and one soundly beats the other, is it because the one that won had a better architecture, or is it because the programmers encoded better programs? Without the ability to examine how programs are implemented at a fine grained level, it is difficult or impossible to determine. Gaining this ability, however, is typically at odds with using complex, real-world problems as benchmarks.

Plant and Salinas [11] attempted to avoid the problem of confounding the contribution of knowledge and architecture by providing an extremely controlled design specification for their benchmark that reduced the differences between implementations almost completely to minor syntactic variations. Although this helps ensure that knowledge remains consistent between implementations, it suffers from its own drawback. Adhering to this approach is likely to result in examining only those implementation methods that are commonly available. This means that architecturally specific properties, such as control mechanisms, may not be fully explored and the architecture as a whole may be misrepresented by the test's results.

To avoid these problems, we propose a methodology where knowledge is encoded in two architectures so that the resulting systems produce the same behavior. As part of this methodology, it is critical that the agents are exposed to identical situations and thus receive identical input from their sensors. In many cases, especially simple ones, this is straight forward. In cases where the agent's input or output contains stochastic elements, however, the processes that govern this behavior must be replaced with computationally

equivalent deterministic processes to allow the agent's exact behavior to be reproduced on each iteration of the experiment. This modification is necessary because an agent's performance may vary drastically between iterations if nondeterministic elements are used.

Thus, we do not intend to measure which architecture is able to generate the best solution, but instead measure which architecture uses fewer resources in terms of the amount of knowledge required to achieve a specified behavior and the amount of time required to generate that behavior. Minimizing knowledge is advantageous because it gives a first order approximation as to how much effort is required to encode (or learn) knowledge for a task, which is directly related to how well the architecture supports the knowledge level. By measuring the execution time, we are measuring how fast the architecture can employ knowledge, which in turn gives a measure of interpretation (if it is required) or the overhead from other architectural features. A third measure that would also be useful would be memory space; however, we have concentrated on the first two in our initial investigations.

Application of these metrics will yield information on how the architecture as a whole supports different behaviors. This is done by first decomposing an abstract architecture into a set of capabilities that can be used to develop agents for different domains. Because each architecture provides different computational primitives, the mixture of native constructs and interpreted knowledge required to implement each capability will vary. An architecture's performance profile indicates how well each capability is supported through interpretation of knowledge and the use of architecturally native mechanisms. Thus, although divergent architecture may be best suited to implement a subset of the possible capabilities, and may be unable to support others, our methodology could be applied to any universal architecture in theory. Moreover, it allows comparisons between very different architectures through examination of their performance profiles.

From the agent's perspective, selection and application of operators and operands constitute decisions that must be made and acted upon. We identified five methods that can be used to implement the decision making process. These methods differ in terms of what types of knowledge they use and how their knowledge is organized. In general, two types of knowledge are used in decision making: applicability knowledge, and control knowledge. The first of these determines when an action *can* be applied to a particular state (e.g., by specifying a minimal set of conditions that must be met for the action to be feasible). The second type of knowledge determines when action *should* be applied to a particular state (e.g., by specifying a maximally specific set of conditions that must be met for the action to be pursued). The five methods described below are linearly ordered such that control knowledge is increasingly segregated from applicability knowledge. Agents that do not attempt to differentiate control and applicability knowledge, do not benefit as much from knowledge generalization. In the extreme, such agents will have difficulty acquiring new rules (e.g., through learning) because of the specificity required to properly encode their conditions. Moreover, large systems designed in this manner may not function because computational resources are exceeded.

The methods below are not meant to be completely comprehensive, rather they are part of the total set of decision making methods that in the limit include selections based on analogy, utility, planning, etc.

1. *Mutually Exclusive Actions:* Both applicability and control knowledge are merged so that actions are considered only if they should be selected. To prevent race conditions, the programmer must ensure that the conditions of each action are mutually exclusive.
2. *Segregation of Control Knowledge:* Control knowledge is somewhat segregated from applicability knowledge, but the control knowledge is restricted to situation independent selection heuristics. Thus, situation specific control knowledge is not possible.
3. *Two Phase Decision Process:* Two distinct phases occur in the decision process. During the first, all possible actions are proposed, and during the second phase one of these actions is selected and then applied. This allows the preconditions for considering actions to be less specific than in either of the previous cases, and allows situation-dependent control knowledge to be used for selecting the best action.
4. *Three Phase Decision Process:* Actions are first proposed, then evaluated and finally selected. Three phases allows knowledge to be segregated further. This creates four general classes of knowledge that each rule may encode: proposal, preference, selection and application.
5. *Goal Constrained Selection:* The agent uses high level goals to pursue problem solving and to control search. This includes standard means-ends analysis where an action may be selected even when it can not immediately apply, and its selection can be used to constrain the selection of subsequent actions.

Although our ultimate goal is to apply this methodology to a wide variety of architectures, we need to start somewhere. Given the potential pitfalls inherent to empirical evaluation, it behooves us to start with architectures that are not radically different. Given our experience with Soar, it made sense to pick another rule-based system. CLIPS was an obvious choice given its wide use and free availability. The following section describes the Soar and CLIPS architectures and provides a motivation for examining particular components of these systems. Our implementation of this methodology with respect to Soar and CLIPS is discussed in Section 5.

4 Architectural Description

Soar and CLIPS are similar in many respects; they are both forward chaining productions systems based on the Rete matching algorithm (Doorenbos [2], Forgy [3]) and implemented in the C Programming Language. Our evaluation, however, focused on four critical architectural properties in which overt distinctions could be observed: knowledge representation, knowledge access, knowledge deployment, and decision making. Differences in knowledge representation, access and deployment impact the expressiveness of the design language, and this contributes to the relative ease or difficulty of producing certain behaviors. Similarly, a less general decision making process may result in less robust behavior. The following two sections describe the CLIPS and Soar architectures with respect to these issues.

4.1 CLIPS

In CLIPS, short term declarative knowledge is stored as a list of facts[1]. A fact consists of a name and a series of slots that can either be ordered or unordered (Figure 1). Facts with ordered slots can be thought of as lists in which the first element is the name. A series of values follows the name, and the semantics of a particular entry is determined implicitly by its position. Alternatively, unordered facts can be used to explicitly define a fact's attributes and values. An unordered fact has a series of named slots, each of which can be set to accept either a single value or multiple values. Neither of these options, however, provide a good mechanism for encoding data objects that are incompletely specified. In this case, slots that are unused must be filled with place holders, and system resources must still be devoted to their representation.

Short Term Memory	A Matching Rule
(month January) (day Saturday) (task (name get-car) (priority 2))	(defrule when-to-get-car (month January) (task (name get-car)) => (assert (do (what get-car) (when now))))

Fig. 1. Knowledge Representation in CLIPS

Long term knowledge is stored as rules that contain conditions on the left hand side, and consequences on the right hand side (Figure 1). Rules become activated when all of their conditions are matched by the current contents of short term memory (facts). CLIPS syntax allows a wide variety of conditions to be expressed. For example, fact values can be variablized and required to satisfy arbitrary predicates, and both conjunction and disjunction of conditions is accepted. The right hand side of rules can be used to change to contents of sort term memory or to invoke procedural knowledge through a variety of mechanisms.

CLIPS deploys knowledge by firing productions serially. During execution, matching rules are placed on the agenda, and the rule on the top of the agenda is fired. This results in a recalculation of matching rules, and thus a possible modification to the agenda. Two factors contribute to a matching rule's position in the agenda. The first of these is salience, which defines a rule-level preference. The salience value may be computed before or during execution; rules with higher values are placed closer to the top of the agenda. The second mechanism is called the search-strategy. This defines how rules of equal salience are placed relative to each other. CLIPS offers many possibilities for this setting including depth-first, breadth-first and random.

CLIPS does not provide architectural mechanisms specifically to support all of the decision making processes listed earlier. As a result, the more complex schemes must be implemented through the use of additional knowledge.

[1] CLIPS also allows short term knowledge to be stored as Objects, via use of the CLIPS Object Oriented Language, however this was not examined in our experiment.

4.2 Soar

Soar's short term memory is a graph that is organized as a set of states (Figure 2). Each state has an arbitrary number of slots and corresponding values. Slots can be named with any alphanumeric string, and their values can be either a symbol or a link to another slot. The states are hierarchically organized and linked together with the architecturally created `superstate` slot that points to the parent state.

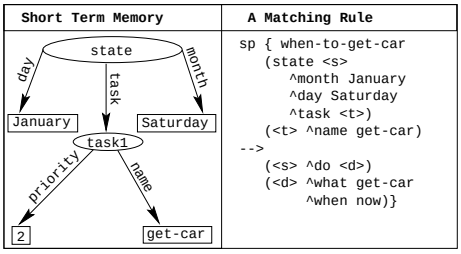

Fig. 2. Knowledge Representation in Soar

Like CLIPS, Soar stores long term knowledge as rules whose left hand sides contain a series of conditions that match against the current contents of short term memory. Although Soar supports standard numeric predicates such as $<, >, =$ etc as well some for determining set membership, it does not allow user defined predicates or functions to be used in the left hand side of a rule. Such matching requires at least two distinct rules, and rule firings. Soar does, however, allow a state's attribute name, as well as their value to be bound to a variable—a property that can reduce rule complexity in some situations. Similar to CLIPS, the right hand side of a Soar rule can be used to modify the contents of short term memory, and to invoke external procedural knowledge. Some rules, however, are used to take advantage of Soar's built in decision making scheme by proposing architectural constructs called operators, or by determining the relative suitability of a set of operators.

Unlike most production systems, Soar deploys its long term knowledge by firing all activated rules in parallel. Once a matching set of rules is calculated, all rules are fired and their consequences are carried out. This sequence, which is repeated until exhaustion, is called the elaboration phase. Parallel rule firing offers two potential benefits to serial rule firing. Foremost, is that all relevant knowledge is brought to bare in each situation, allowing Soar to consider all relevant paths before making a commitment. Secondly, parallel rule firing obviates the need for a rule level conflict resolution mechanism such as CLIPS's search strategy. This means that Soar forces control knowledge to be explicitly represented in the rules and helps free the developer from worrying about race conditions.

Soar supports decision making with the operator construct and an architecturally supported decision cycle. Operators represent high level actions or goals and are selected and applied in serial order. This is supported by Soar's decision cycle, which occurs in a series of phases. After Soar completes the elaboration phase, it examines which,

if any, operators have been proposed, and computes their relative preferences. Finally, the best operator is selected and the cycle ends. During subsequent cycles, the primitive actions of the selected operator will be carried out and new operators will be proposed. At certain times, Soar will be unable to choose an appropriate operator, or will not know immediately what actions are required to achieve the goals of the selected operator. In such instances, the architecture will form a substate in which more primitive reasoning can occur. This hierarchical nature facilitates problem solving at multiple levels of abstraction.

4.3 Summary of Architectural Differences

We have outlined four areas (knowledge representation, knowledge access, knowledge deployment, and decision making) in which the Soar and CLIPS architectures are significantly different. Knowledge representation in CLIPS is restricted to list like structures, whereas Soar supports more general graph structures. Knowledge is accessed by conditions on the left hand sides of rules in both systems. CLIPS provides the ability to use arbitrary function or predicate calls as a part of the matching process whereas this requires two distinct rules in Soar. On the other hand, Soar allows slot names, as well as slot values, to be variablized and matched. Replicating this mechanism in CLIPS is difficult. Knowledge is deployed via rule firings in both systems, but Soar fires all matching rules in parallel whereas CLIPS fires rules serially. Finally, Soar supports decision making with a three phase process in which all relevant knowledge is examined and possible actions are proposed, the relative preferences are then examined and an action is selected. Because CLIPS has no comparable mechanism, this type of strategy must be implemented with rules.

5 Comparing the Architectures

In this section, we outline how our methodology was applied to these two systems and provide an analysis of the reliability of our timing results.

Remember, the goal is not to determine which is the one, best architecture. Our goal is to develop a methodology that can be used to answer questions such as: for this class of knowledge and goals, what are the costs of using each architecture. As we present the results, it will become apparent that we have only scratched the surface in our comparison of Soar and CLIPS, and that many of the possible “strengths” of Soar have yet to be explored.

5.1 Implementation

We implemented the methodology described in Section 3 by examining pairs of agents (each implemented in a different architecture) that utilized the decision making strategies previously described. Recall that our methodology specified that the agents encode the same knowledge and utilize the same strategies, are exposed to the same stimuli, and exhibit the same behavior.

5.2 Task Environments

Tasks were created to test the architectures in two simple domains. Limiting our initial investigations to simple environments, and thus simple agents, allowed us to examine programs closely and thus avoid programmer bias. The first environment we utilized was the classic Towers of Hanoi problem, the second was an interactive environment called Eaters, which resembles the arcade game PacMan. Both environments are deterministic and discrete, as per Russell and Norvig's [13] definition, but the environment state in the Towers of Hanoi problem is completely represented within the agent's memory, whereas the sensors of the Eaters agent only provide limited information about the environment.

The Towers of Hanoi consists of three pegs, and a number of disks, each with a distinct diameter. Originally the disks all begin on a certain peg, stacked from largest to smallest. The goal of the puzzle is to move the stack to a different peg. The only restrictions are that only one disk can be moved at a time, and a disk may never be placed on top of a smaller disk.

The Eaters domain consists of one or more Eaters (agents) that move around a grid-based world eating food pellets. Each Eater only perceives locations within a small radius, yielding incomplete information about the world. The Eaters environment supports a generalized agent communication protocol that is used by both Soar and CLIPS to provide the agent's input and output functions.

5.3 Benchmarking

A critical component of our analysis involves measuring the time required by each architecture to use the knowledge that its agents encode. To do this, an accurate timing mechanism is needed. Our timers are implemented with the `getrusage` system call which returns the CPU time used by a particular process.

The times that we report in the following sections are referred to as *kernel times*. The kernel timer is turned off for the majority of the agent's input and output functions. It is only turned on in these situations for the period in which assertions or retractions are made to the agent's short term memory. Examining kernel time as opposed to total CPU time frees us from the possibility that differences in the optimization of a one architecture's input/output functions have any influence on the results.

Although the timers we implemented provide much higher reliability and accuracy than measurements of total CPU time, any timing mechanism is subject to errors that may skew the results. We identified a number of possible sources that may impact the timing results. These potential pitfalls and our methods of dealing with them are listed below:

– *False Time Reports:* Ideally, toggling a timer on and then immediately off would result in no accumulated time since each of these actions would be atomic, and there would be no delay between execution of the instructions. In reality, the function calls to turn timers on and off are not atomic, and a significant amount of *false* time, T_f, can be accumulated simply by turning the timers on and off repeatedly. Normalization of all timing results was computed by counting the number of kernel timer cycles K_c, and subtracting $K_c T_f$ from the reported kernel time.

– *Terminal I/O:* Output to the display (or to a file) has a significant impact on the kernel time values. All results that we report were conducted with the absence of such input/output, or with only a single output statement indicating the termination of the task.
– *User Interface:* Differences in the implementation of the user interface were observed to have an impact on timing results in both architectures. For the tasks examined, changing from GUI to command line interface improved CLIPS performance by more than a factor of two, as opposed to only $\sim 15\%$ for Soar. As a result, the measurements we report are using the CLIPS command line interface, but retain Soar's standard Tcl interface.
– *Other Factors:* It is possible that factors unknown to us may have impacted the results of our tests. In particular, little attention was paid to compile time options, and only slight modification to the standard Makefiles were used when building the executables.[2]

The following two sections present the experiments and results conducted in the Towers of Hanoi and Eaters domains. In all cases, the experiments were conducted on a Sun Ultra SPARC with 128M of RAM running Solaris 2.5.1. The times reported are kernel times and the we have attempted ensure their accuracy and reproducibility with the methods described above.

6 Towers of Hanoi

Within the Towers of Hanoi (ToH) domain, we implemented a set of five agents, each of which utilize a different set of decision making knowledge. Although internal differences among these classes of agents are significant, their external behavior is identical. They all implement the optimal strategy for solving the puzzle (that is, they require only the minimal number of moves).

Figure 3 shows the performance results for both Soar and CLIPS agents over the range of problem solving strategies we examined. Qualitatively, the shape of the performance curve is well matched between the two systems, indicating that at least in simple domains, CLIPS is fast enough to emulate more robust decision mechanisms such as those natively implemented in Soar without a serious cost to performance. Two exceptional points on the curve occur at either ends of the decision mechanism spectrum. On the left hand side, Soar performs less well than CLIPS, and even less well compared to a Soar agent using more segregated control knowledge. This difference may be attributable to the fact that Soar agents that segregate control knowledge by using the operator construct are able to constrain rule matching to improve overall performance. On the right hand side of the plot, two Soar agents have been implemented.

The first of these (1), outperforms half of all Soar agents tested, whereas the second agent (2) is more than a factor of two slower than its counterpart. Somewhat surprisingly, these agents differ only in how goals are represented. Agent 2 uses Soar's built

[2] Two compile time options were modified in Soar's default Makefile. The first prevents Soar from gathering unnecessary statistics about its memory usage, and the second results in the incorporation of only the timers that we implemented in CLIPS (kernel time and total CPU time).

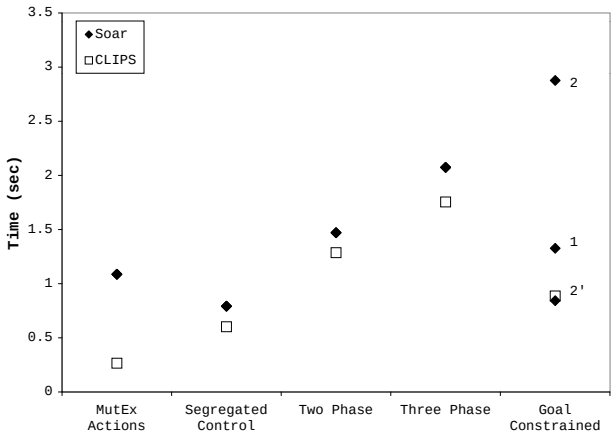

Fig. 3. Towers of Hanoi Performance (9 Disks)

in subgoal mechanism, whereas agent 1 is directly paralleled to the CLIPS implementation and uses rules to maintain a stack of goals. This illustrates one of the potential trade–offs which arise when functionality is built into an architecture, but is not fully used. In this particular example, using Soar's architectural subgoal stack is not beneficial because an extremely large number of goals are being produced. Each time a subgoal is created or destroyed, some amount of processing is done to support Soar's learning mechanism, even in cases such as this, when learning is explicitly turned off. The other agent (1) avoids this cost by maintaining its own data structure to represent the goal stack. The significant difference between the performance of these two implementations has led to the development of a new version of Soar, which we call Soar–Lite. Soar–Lite is a streamlined version of Soar which sacrifices some costly, and less frequently used, features in order to achieve an overall performance gain. The main difference between the current version of Soar–Lite and Soar is the removal of Soar's learning mechanism. The agent code for (2) was rerun in the Soar–Lite architecture and achieved more than a three–fold increase in speed. The performance difference is only substantial for the agent which uses architectural subgoaling, this data point is plotted in Figure 3 and labeled as $2'$. For all other data points, the performance difference is less than .15 seconds. The results of this series of tests indicate that there is a similar performance trade off when using more complex decision mechanisms in both Soar and CLIPS. At the same time, they also indicate that very little penalty is incurred for Soar's architecturally supported decision cycle and that the cost of subgoaling can be canceled by removing Soar's learning mechanism. Moreover, as the complexity of the domain increases, we expect that architectural based support for these features will provide a higher degree of efficiency than a rule based counterpart.

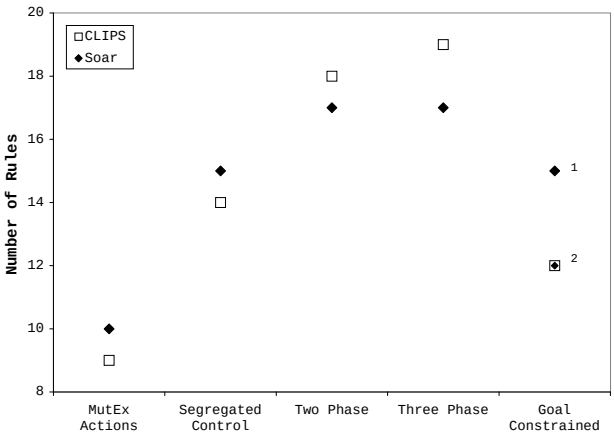

Fig. 4. Knowledge Requirements of Decision Making Strategies

Figure 4 shows the number of rules necessary to encode agents described above. For both architectures, a similar pattern can be recognized because the number of rules increases with the complexity of the decision mechanism. This is expected, because control knowledge is becoming increasingly distinct from applicability knowledge in each successive implementation. As the agents incorporate more and more knowledge, however, we would expect this result to diminish and perhaps reverse. This is predicted by the fact that smaller rules, which contain less control knowledge and are therefore more generalized, can be more easily reused in multiple situations. The two versions of the final Soar agent are also displayed in this plot. Using Soar's built in subgoal mechanism lowers the number of required rules, and when used with Soar–Light, this implementation achieves very good performance relative to other decision making strategies. The Soar agent that maintains a goal stack (2) requires slightly more rules than its corresponding CLIPS implementation because of the need for an additional operator (that of building the goal stack) that is not required by CLIPS. Both goal stack agents, however, are more efficient (both in terms of time and rules) than their corresponding two-phase or three-phase counterparts. This result can be attributed to two facts. First, the Towers of Hanoi is a naturally recursive problem and the ability to commit to an action before it is possible to actually carry out that action results in a simpler solution to the puzzle. Second, the goal stack provides a light–weight mechanism to support this implementation and yet does not require devoting computational resources to unutilized features.

Finally, because the complexity of the Towers of Hanoi problem has a simple, well known, closed form, we were able to examine how close the Soar and CLIPS implementations to this ideal. This analysis allows us to examine how well the architectures

scale as the required time to solve a problem increases. Figure 5 displays the scalability of Soar and CLIPS agents that use mutually exclusive productions and segregated control knowledge to make decisions. Exponential curves were fit to this data, providing fits of $0.0019 \cdot 2.025^n$ and $0.002 \cdot 1.941^n$ for Soar agents, and $0.0005 \cdot 2.015^n$ and $0.001 \cdot 2.058^n$ for CLIPS agents respectively. All of these fits come very close to the ideal growth rate of 2^n indicating a high potential for scalability as time to solution increases.

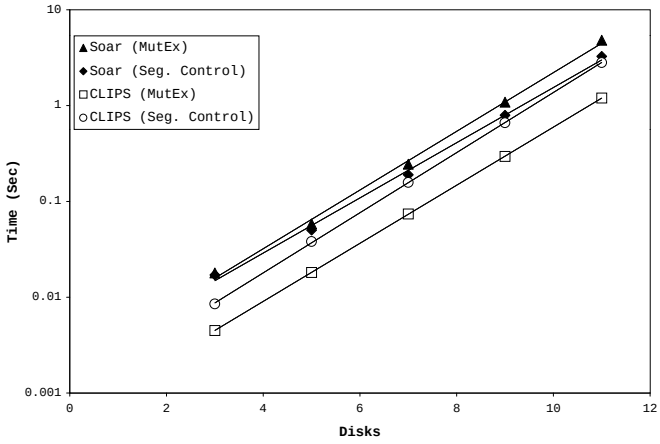

Fig. 5. Growth of Time Requirements in Towers of Hanoi

7 Eaters

To gain a deeper understanding of how Soar and CLIPS support higher levels of decision making, we examined how agents that use the operator and operator evaluation strategies perform as they are faced with a greater number of potential actions. In the Eaters environment, movement can only occur in cardinal directions. To examine situations in which more than four actions were under consideration, we modified the agent's precepts (represented by short term memory structures) so that it observed successively more directions in which it could move (4, 8, 16, 24, 32, 40). This allowed us to make only minimal changes to the agent's rules at each successive perceptual increase. A command from the agent to move in one of the new directions was simply mapped onto one of the four cardinal directions in a deterministic manner. Figure 6 shows the performance of the Soar and CLIPS versions of this agent as a function of the average

number of possible actions that were evaluated during the course of each decision. The agents considering between 0–4 candidates used 4 directional percepts, those considering between 4.1–8, 8.1–16, 16.1–24, 24.1–32 and 32.1–40 candidates used 8, 16, 24, 32, and 40 directional percepts respectively. The figure shows only results for agents which use a three phase decision process, however, agents using two phase decision process exhibit the same behavior, indicating that the performance curve is not an artifact of the decision making mechanism. The figure depicts three important phenomena. First, over all values of candidates, decision time in Soar remains almost constant, exhibiting only minor growth. Second, decision time in CLIPS increases sharply when the number of percepts increases. Third, given a constant number of percepts, the growth of decision time in CLIPS is also close to constant. Most likely, the discontinuity in CLIPS performance is a result of a sensitivity to short term memory structure (e.g., the number slots in an unordered fact) which is not apparent in the corresponding Soar agents. This indicates a potential problem for building agents in environments that require a high degree of perceptual complexity, or those in which the perceptual complexity may be increased significantly during the course of development.

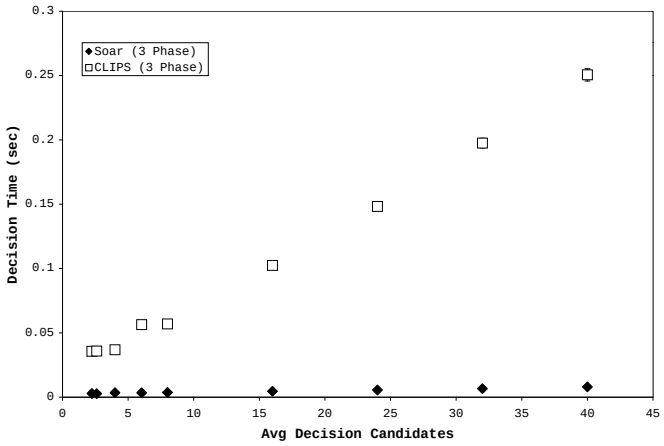

Fig. 6. Average Time Required Per Decision

8 Conclusion & Future Work

We have developed a methodology which can be used to evaluate the degree to which a universal architecture supports capabilities required by various agents. We have applied

this methodology to the Soar and CLIPS productions systems by examining their ability to support a variety of decision making mechanisms. Our results serve as a proof of concept that this approach can be used to build performance profiles for various architectures that can help illustrate the similarities and differences of these architectures.

Future work in this area requires increasing the breath of our study on two fronts. The first of these is to begin by examining a new set of capabilities distinct from those used for decision making such as the ability to support learning, use fuzzy logic, or make hard real-time guarantees. The second front involves exploring a larger number of architectures of progressively greater diversity. As the breadth of these two fronts increases, we hope to be able to establish trends and theories not only about the particular architectural implementations, but also about their more abstract underpinnings.

References

1. *CLIPS Reference Manual: Version 6.05*.
2. R. B. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Carnegie Mellon University, 1995.
3. C. L. Forgy. *On the Efficient Implementation of Production Systems*. PhD thesis, Carnegie Mellon University, 1979.
4. E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for mobile robots. In *Proceedings Tenth National Conference on Artificial Intelligence*, pages 809–15. AAAI Press, 1992.
5. W. B. Gevarter. The nature and evaluation of commercial expert system building tools. *Computer*, 20(5):24–41, 1987.
6. F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6):34–44, Dec. 1992.
7. J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 1987.
8. Jaeho Lee and Suk I. Yoo. Reactive-system approaches to agent architectures. In N.R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI — Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2000. In this volume.
9. W. A. Mettrey. A comparative evaluation of expert system tools. *Computer*, 24(2):19–31, 1991.
10. A. Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.
11. R. T. Plant and J. P. Salinas. Expert system shell benchmarks: The missing comparison factor. *Information & Management*, 27:89–101, 1994.
12. M. E. Pollack and M. Ringuette. Introducing the tileworld: Experimentally evaluating agent architectures. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, volume 1, pages 183–189. MIT Press, 1990.
13. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, chapter 2, pages 31–52. Prentice-Hall, Upper Saddle River, NJ, 1995.
14. A. Th. Schreiber and W. P. Birmingham. Editorial: the sisyphus-vt initiative. *International Journal of Human-Computer Studies*, 44(3):275–280, 1996.
15. A. C. Stylianou, R. D. Smith, and G. R. Madey. An empirical model for the evaluation and selection of expert system shells. *Expert Systems With Applications*, 8(1):143–155, 1995.
16. A. Tate, B. Drabble, and R. Kirby. O–plan2: An architecture for command, planning and control. In Mark Fox and Monte Zweben, editors, *Intelligent Scheduling*. Morgan Kaufmann, 1994.