Examining the Resource Requirements of Artificial Intelligence Architectures

Scott A. Wallace
John E. Laird
Karen J. Coulter
University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109-2110
734-647-1761
swallace@umich.edu

Keywords:
Architecture evaluation, Soar, CLIPS

ABSTRACT: *Over time, the range and complexity of behaviors in computer-generated forces has expanded as they move to include more autonomy and intelligence. To support the representation and execution of these behaviors, there is a tendency to add more and more features to the underlying software architecture. However, with the addition of these features, two potential costs may be incurred: increased execution time and additional memory requirements. As architectures progress, it is important to continually evaluate the cost and value of each new architectural feature. Seemingly very similar architectures may require significantly different resources; small changes to the features in a single architecture may have a large impact on its performance. Thus, it is necessary to understand and to quantify the resources consumed by different architectures and by the components of a single architecture. Unfortunately, there is no standard method for evaluating features of an architecture or for comparing sets of architectures. In this paper, we discuss a methodology for evaluating a specific architecture. We dissect the Soar architecture into a core set of functionality and examine how incrementally adding each of the features found in the original implementation affects the overall performance and resource requirements. Finally, we show how the same methodology can be used to compare two different architectures. We discuss initial results of a comparison that indicates both qualitative and quantitative differences between the Soar and CLIPS architectures.*

1. Introduction

As computer generated forces, and artificial intelligent agents in general, become increasingly robust and autonomous, the software underlying their behavior also becomes more and more complex. Success with simple agents in simple domains inspires research into the capabilities required to operate more efficiently and effectively. This in turn causes the software architectures to evolve, as functionality is added to support the new demands. Because this is a common process, many architectures have been developed incrementally over the course of many years as they become increasingly sophisticated.

Design decisions made at implementation time often play critical roles in the efficiency (both in time and space complexity) of the architecture. The impact is seen both when the new features are used by an agent and in some cases even when the features are not used. Thus, after a feature is added to an architecture, agents operating in complex domains and relying heavily on the new feature may operate more efficiently than was previously possible. At the same time, agents that do not rely on the new architectural feature may become less efficient. To properly assess the impact of an architectural modification, it is necessary to quantify the resource consumption of that particular modification. In most cases, it is extremely difficult to draw meaningful conclusions using analytical methods, although in some cases, a formula that relies on prior knowledge of a relatively few variables may be obtainable. Even in these instances, however, comparisons between two such formulas are further hampered by the fact that constant factor differences may have profound implications on their relative suitability in real-world tasks. As a result, we believe that empirical methods are currently the most suitable way to evaluate the impact of design decisions.

Additionally, two distinct agent architectures are likely to yield agents with differing efficiencies even if the architectures (and agents) appear otherwise similar. As architectures become increasingly divergent, it may become overtly obvious that the features of one architecture are better suited to a particular task than are those of another. In many cases, however, this is not necessarily clear a-priori. As a result, designers of intelligent agents, and of agent architectures may benefit from understanding the relative differences in resource consumption between two or more architectures. As in the single architecture case, empirical methods can yield approximate answers to these questions. Unfortunately, however, there are no standard methodologies for evaluating the resource consumption of a particular architecture or of components of a single architecture. In this paper, we discuss a methodology that can be used to examine the resource requirements of an architecture as a whole, or of particular aspects of that architecture. We present a practical example by applying this methodology first to components of the Soar architecture and then to the standard version of both the Soar and CLIPS architectures. Our results show both qualitative and quantitative differences between these two architectures and show how components of the Soar architecture contribute to its overall performance. An early version of some of the material and results in this paper appeared in [10].

2. Modular Architectures

Classically, agent implementations have been decomposed into two parts: architecture and knowledge [8]. In keeping with this distinction, we will use the term architecture throughout this paper to refer to a software platform used to design intelligent agents. At a minimum, the architecture must provide the basic functionality of input and output as well as an ability to represent and deploy knowledge. Knowledge representation may take the form of programmer supplied rules, frames or procedures; deployment refers to the mechanism with which the architecture determines what course of action should be pursued based on its current knowledge. In addition to these basic features, the architecture may provide functionality that exceeds the minimal requirements. For example, it could supply a learning mechanism or a method for calculating the utility of newly acquired knowledge. Succinctly put, the architecture is a domain-independent set of theoretical commitments which, when combined with domain-dependent knowledge, can be used to specify a range of agents, for use in a variety of environments.

At this point, it may be helpful to introduce a concrete, although somewhat contrived, example to illustrate the distinction between architecture and knowledge. Consider a factory that produces a variety of thermostats for different applications. Each thermostat is similar in that it has an input system, in this case the current and desired temperatures, and an output system, allowing it to supply either the furnace or air conditioner with a variable amount of power. In addition, each thermostat has the ability to examine and apply a set of stored operators in order to bring the room to the desired temperature. In certain applications, it may be most desirable to achieve the proper temperature by supplying power to either the air-conditioner or to the furnace in proportion to the difference between the current temperature, Tc, and the desired temperature, Td. In other applications, perhaps because of limitations on the heating and cooling devices, the desired temperature may need to be achieved by supplying full power to one of the devices, regardless of the difference between Tc and Td. In either situation, the method for achieving the proper temperature can be stored as a single operator that tests the current state. Moreover, the appropriate method (determined by the operator) can easily be programmed into the thermostat and may be changed without having to re-engineer the thermostat itself.

In this example, the architecture consists of the relatively unchangeable thermostat hardware. Knowledge is represented by the operator (or operators) stored in the thermostat and is deployed by the hardware to achieve the desired temperature. Notice that changing the knowledge can lead to a whole class of thermostats with different methods of achieving the desired temperature. Notice also that it may be possible to "hardwire" a particular method for controlling the temperature into the thermostat itself (i.e. incorporate a piece of knowledge into the architecture). However, this modification may be undesirable since the entire thermostat would have to be replaced if the requirements of the heating and cooling systems changed after installation.

This example provides an illustration of the minimal framework required by an architecture. It also illustrates the necessarily blurry distinction between knowledge and architecture. Because most agent architectures are Turing complete, features not supplied directly by the architecture can be emulated by the appropriate addition of knowledge, but with additional execution time overhead. What the previous example does not show is the concept of modularity. Architectures are modular in so far as features can be removed while still preserving the basic requirements of an architecture. Note that this is different from simply being able to refrain from using certain features because it suggests that the internal design of the architecture with and without a modular feature is different.

3. A Methodology for Agent Architecture Evaluation

The methodology we have employed to conduct our studies is straightforward in the case of examining the components of a single architecture. The first step begins with dissecting the architecture into constituent modules, leaving a core set of architectural features intact. In many cases, such as when an architecture is developed incrementally, certain features may be naturally modular. In other cases, a great deal of thought may be required before determining what aspects of the architecture can be removed while still allowing the core functionality to meet the design goals of the researchers. In either situation, modifications to the source code will undoubtedly be necessary to construct a set of architectural variants that combine different modular features with the core functionality.

The second step in our methodology consists of determining a class of situations in which to examine the architectural variants. Particularly interesting problem classes may be found at both ends of a spectrum from situations that typically do not rely on a specific architectural feature to those which typically rely very heavily on such a feature. Although any single study is likely to be limited to examining a relatively small problem class, as the number of studies increases, it is anticipated that general trends will emerge indicating which architectural variant is most suited for a particular class of problems.

The third step involves selecting an environment in which to examine the problem class selected in the previous step. Because there is no single environment that can be used to represent "environments" as a whole, selection must be made with care, and equal care must be used to ensure that results are not over generalized. Understanding how the environment fits within a typical taxonomy (e.g. from Russell and Norvig [10]) may help moderate this problem.

Fourth, for each architectural variant, an agent must be designed to solve the specific problem within the selected environment. Agents solving the same problem form a group. All agents within a group must utilize the same problem solving methods. The effect of this constraint is that any two agents within a group must not only have identical interactions with the environment, but must also utilize the same internal problem-solving methodology in so far as is possible. Proper implementation of this step is critical; otherwise there is a serious risk of confusing the contribution of different architectural aspects and different knowledge (i.e. problem solving methods) on the overall results. However, in certain circumstances this problem is eliminated because all of the agents within a group can be implemented using identical knowledge. This exceptional case occurs when architectural variants differ only in their inclusion or exclusion of unused features. Once a group of agents has been fully implemented, the performance of agent/architecture pairs can be directly compared.

4. Soar and Its Modular Components

The Soar [5] architecture is a forward chaining production system based on the RETE matching algorithm [2;3]. It contains a long-term memory (LTM) that stores production rules, and a short-term memory (STM) containing elements that are matched by the rules.

Short term, potentially volatile, knowledge is stored in STM in the form of a directed graph with labeled edges. Each memory element can be thought of as an ordered triplet whose slots refer to the parent node, the edge name and the child node respectively. Because this structure is so generic, it can be used to represent a multitude of more complex data structures.

Long term, stable knowledge, is stored in LTM as a set of productions. Productions are created explicitly by the programmer, or may be generated automatically by Soar's learning mechanism. The condition of a rule may contain either variables or constants, and variables may be bound to any of the three slots in a memory element's ordered triplet. This ability allows a large amount of flexibility in terms of how a rule is designed, but it can also greatly increase matching costs when it is used indiscriminately. The condition side of Soar's rules may also ensure that values bound to a variable satisfy one or more basic predicates (e.g. >, <, =). Generic predicates, however, are not supported in a rule's conditions. The right hand, or action side of a rule, can be used to modify the contents of STM. Additionally, it can propose architectural-constructs called operators or preferences for such operators.

In Soar, knowledge is deployed by rule firings. This process begins as follows:

- First, determine which rules, if any, match the current contents of STM.
- Next, fire all matching rules in parallel, by executing the instructions in their right hand side.

These two steps, called an elaboration cycle, are repeated until a quiescent state is reached in which no more rules can fire. Parallel rule firings allow Soar to make use of all relevant knowledge in a given circumstance. It also forces programmers to explicitly encode control knowledge into rules to select operators instead of relying on a potentially cryptic architectural mechanism to determine which rule among the current matches should actually be fired.

In addition to the basic execution supported by the elaboration phase, Soar also has an architecturally supported decision-making phase that occurs immediately after elaborations have ceased. During the decision phase, operators representing actions of higher-level goals, which have been proposed during the elaboration phase, are examined. The operators are ranked according to their relative preferences, which have also been specified during the elaborations. At this point, Soar selects the operator with the highest preference to be pursued.

Although the serial nature of pursuing operators may seem similar to productions systems that fire rules serially, this is not typically true. One important distinction is that in Soar, knowledge about proposed operators is explicitly declared, and is available to be used for further reasoning, whereas information about matched rules in a serial system is typically not available to be used in this way.

In certain cases, Soar may decide that it is no longer making progress on the current problem (e.g. the elaboration phase terminates without any rules being fired, or Soar cannot select between two operators). In such cases, Soar will react by creating a sub-state in which further reasoning can take place. Within this sub-state, operators can be proposed and pursued just as in the super-state. The sub-state vanishes when Soar has done enough reasoning to resolve the problem that triggered its creation. It is during the resolution of sub-states that Soar's learning mechanism creates new search control knowledge (in the form of a rule) and adds it to LTM so that similar sub-states (and the additional reasoning to resolve them) can be avoided in the future.

Although Soar has been developed incrementally over a number of years, the mechanisms needed to modularize the architecture were not completely in place. Nonetheless, some features were clear candidates for modularization, and these are listed below:

- Detailed Timing Facilities - Soar has the ability to keep track of the time spent on various aspects of execution. In many cases, however, only minimal timing information is required, and this feature is not necessary.
- Callbacks - Soar has the ability to invoke user-defined functions that have been registered with the architecture. Some of these callbacks are invoked many times per decision cycle, and even if no functions are registered with the architecture, some overhead is incurred due to looking up and testing one or more variables.
- Learning - Each time Soar completes reasoning within a sub-state, the architecture has the ability to learn a new rule. When using Soar to build computer-generated forces, however, learning has not been employed because these forces have been expected to perform at an expert level without undergoing a potentially costly training phase.
- Backtracing Mechanism - Soar also has the ability to keep (potentially elaborate) information as to how it reached a particular conclusion. The full power of this feature is used only during learning. Thus, as only a small portion of this mechanism is required for other purposes, significant amounts of source code can be removed or optimized when learning is also removed.

The four features we have identified above are only a subset of the features in the Soar architecture that could be modularized. However, this partitioning of the architecture was particularly suitable for our initial exploration because in certain testbed environments, a single set of knowledge could be used to examine all of the resulting architectural variants.

Once these components were identified, compiler directives were inserted into the source code that allowed only a specific subset of features to be included in the executable. Although with four modular features, many architectural variants were conceivable, we selected only a small subset of these, which, based on our experience, offer a practical combination of abilities.

Three variants of the Soar architecture were examined for our tests by including or excluding some or all of the modular features described above. Variant 1, which we will also refer to as the standard version of Soar, includes all of the modular features. Variant 2, removes the Detailed Timing Facilities as well as the Callback module. This variant preserves basic Soar's functionality including its learning capability. Variant 3, which we will also refer to as Soar-Lite, removes not just the Detailed Timing Facilities and Callback module, but also the Learning and Backtracing Mechanisms as well.

5. Decision-Making Strategies

The class of problems we have selected for the initial implementation of our methodology is what we refer to as decision-making strategies. Most, if not all, agents are similar in that they must examine their current state and decide which of the many possible options to pursue. This process can take place in a variety of ways. In particular, one set of methodologies which can be used by Soar (as well as by a potentially large set of agent architectures) focuses on the individual pieces of knowledge which must be brought to bear in order to make the most appropriate decision about the next action. Some agents, for example, may use knowledge that directly ties a particular state or set of states to the most appropriate action. If the preconditions for each action are disjoint, only a single piece of knowledge will be brought to bear in any given situation, and the decision will essentially make itself. This is analogous to the operation of a lookup-table. Other agents may bring multiple pieces of knowledge to bear in order to make their decision. As the knowledge becomes hierarchically organized, the agent will go through an increasing number of refinement steps (reflected by a path in the tree from the root to a leaf) before it is able to select the most appropriate action for the circumstances. It is this general process of refinement that we have used as the basis for this study. Below is a list of decision-making strategies in which the refinement process is increasingly complex:

- Simple, Declared Actions - Actions are represented declaratively to the system, in Soar this is done using operators. The programmer supplies enough knowledge to guarantee that only one action is applicable at any given time, thus no conflicts between courses of action can arise.
- Three-Phase Decision - The decision takes place in three distinct phases. In the first of these, actions are proposed, in the second phase actions are ranked according to their relative preferences and finally the most preferred action is selected and then pursued. This allows for multiple layers of refinement in the decision making process, potentially decreasing the size and complexity of the knowledge base.
- Goal Directed - A goal is a subtask that requires the application and pursuit of a sequence of one or more actions (operators). In this strategy, goals are selected the same manner as primitive actions. Once a goal has been selected, it may significantly constrain the subsequent problem solving, resulting in fewer matches. Soar expresses goals with high-level operators, and uses sub-states to perform the reasoning needed to achieve these goals.

6. Towers of Hanoi

The Towers of Hanoi problem is well known to the AI community and has an equally well-known optimal solution. Although it is a relatively simple problem, it is complex enough to examine the class of decision-making strategies outlined in the previous section. Moreover, the simplicity of the problem combined with the fact that all agents interact with the puzzle identically (i.e. they follow the optimal solution) means that the differences between the agents' knowledge is limited to exactly what is required to implement each decision making strategy. It is important to remember that we intend this environment to be used as a starting point for further investigation, and as a proof of concept. No single domain can claim to be representative of all situations an agent may face in general.

Figure 1 shows the runtime performance of the Soar architectural variants described in section 5. Across all problem-solving strategies, significant timesavings are achieved between variants 1 and 2 as unused features are removed from the architecture. Further savings are achieved in the Tower of Hanoi subgoaling agent because the differences between variants 2 and 3 affect the efficiency of the architectural subgoaling process in situations where learning is not employed. Based on these

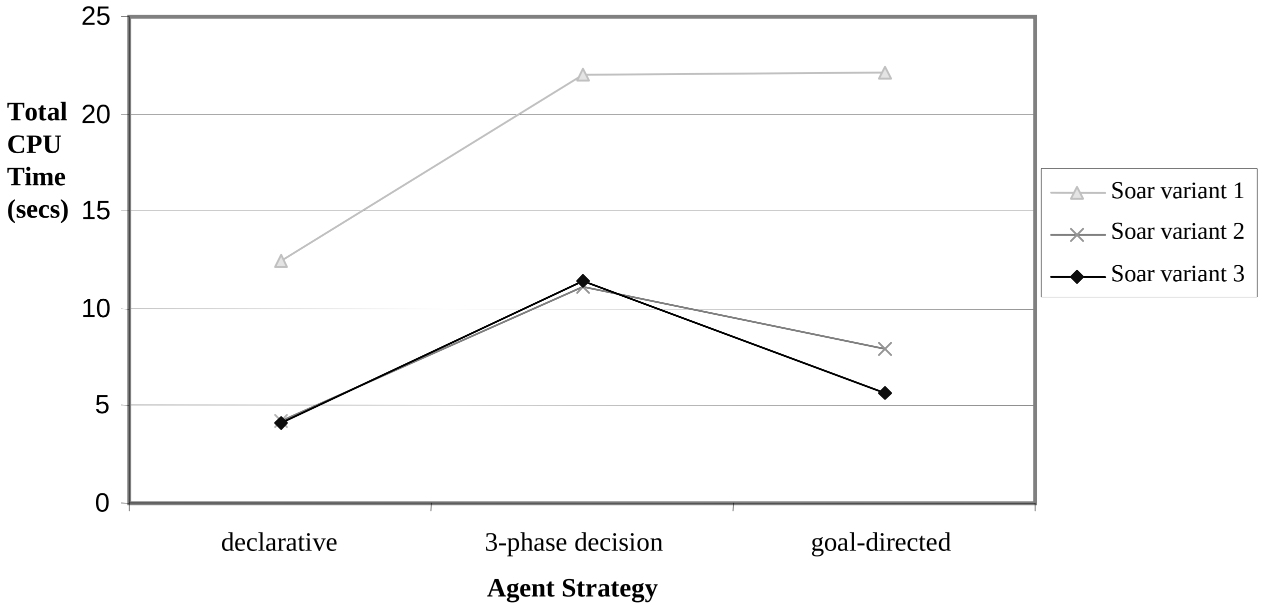

Figure 1: Soar Runtime Performance for Architectural Variants.

results, and knowledge of how the architecture was modified, we expect that all Soar agents that do not require learning will achieve some performance savings by using the more streamlined architectural variants. Moreover, we further expect that agents which will be most enhanced will be those that solve problems similarly to the Towers of Hanoi subgoaling agent above. That is, by using a large number of subgoals, each of which requires relatively little reasoning to resolve on its own.

7. Complex Real-Time Task: Quake II

The tests we conducted in Section 6 seemed to indicate that a substantial savings could be gained in situations that do not require learning. As noted earlier, the computer-generated forces previously developed by the Soar group have all shared this attribute. Moreover, a benchmark in a complex, real-world, environment such as those typically used with computer generated forces seemed to be an ideal counterpart to our earlier benchmark in the relatively simple Towers of Hanoi domain.

The computer generated force we selected for this set of tests was an obvious choice. Constructed by one of us (Laird) to run with the latest version of Soar, it is suitably complex (employing ~600 rules) and operates in the highly dynamic environment of the Quake II computer game. Although Quake II shares few, if any, attributes with the Towers of Hanoi puzzle, application of our evaluation methodology within this new domain was straightforward. As in Towers of Hanoi, a single set of knowledge could be used to test all of the Soar architectural variants, and testing followed the same basic procedure. The only significant difference resulted from the fact that in the Quake II environment, exogenous events are possible. Unless the world's events unfold in exactly the same manner between tests of two architectural variants, it is impossible to determine whether the agents interacting with the world underwent the same processes of reasoning. As a result, whether or not the performance of the architecture/agent pairs is comparable also depends on the ability to ensure that the world's events unfold in a repeatable manner.

To ensure that this did happen, the agent was initially allowed to operate in the Quake II environment by competing against a human opponent for a predetermined amount of time. During this phase, the agent's sensory inputs were recorded and stored in a file. During benchmarking, however, agents did not actually communicate with Quake II. Rather, their sensory input was replayed in exactly the same manner as occurred during the initial recording phase. Not only did this allow us to ensure that agents always performed the same reasoning, but because agent inputs were read from disk and stored in an array for fast retrieval prior to benchmarking, it also guaranteed that timing results would reflect Soar's true performance, and not be skewed by a communication bottleneck with the environment.

Figure 2 shows the run time performance in Quake II for the standard version of Soar (variant 1) and Soar Lite (variant 3). The performance was measured by recording

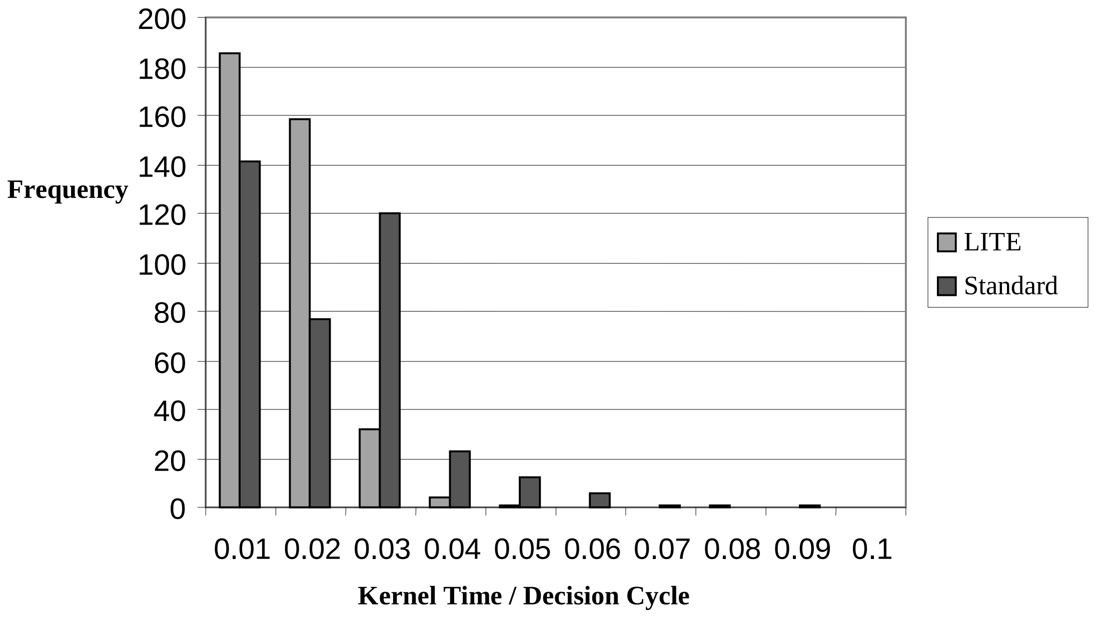

Figure 2: Execution histogram of standard Soar and Soar Lite for 380 decision cycles in the Quakebot.

the time required to complete each of 380 successive decision cycles. The histograms in Figure 2 show the number of cycles that were performed within the various times. The best behavior is to have all of the decision cycles execute in the minimal amount of time (to the left). As this behavior is difficult to achieve, a secondary goal is to have a low variance without any outliers so that there are not decisions that disrupt the overall system execution. In the figure, the standard version of Soar does have a high variance and lots of outliers. In contrast, the Soar Lite version shifts the histogram to the right so that almost all of the decisions execute in .03 seconds or less. There is one significant outlier at .08, but that is the first decision when working memory is initialized and it is irrelevant to the overall runtime performance of the bot. This demonstrates that not only does Soar Lite improve the aggregate execution time (in this case there is a factor of 3 improvement in average execution time) but it improves it at the level of individual decisions in a such a way as to decrease the overall maximum computational requirements of any single decision.

8. Comparing Multiple Agent Architectures

The methodology described in section 3 and that we have employed to examine the performance of the Soar architecture and some of its variants can also be used to examine or compare two distinct architectures directly. The same steps are applied as outlined previously, but the architectures need not be split into modules. The most difficult aspects of using our method for distinct architectures are deciding what class of problems to examine and how to implement the agents. The difficulties stem from the fact that problem definitions must be highly constrained so that each agent's knowledge is extremely similar, if not identical. At the same time, however, these problem definitions are likely to require more flexibility than in the single architecture case, because perfect behavioral analogues may not exist between two architectures. Thus, the burden is on the research team to ensure that agents are appropriately similar and that they encode the same knowledge. As in the single architecture case, once agents have been created, their performance in the problem domain can be measured and compared.

8.1 The CLIPS architecture

As an initial choice of a second architecture with which to conduct our evaluation, we have selected CLIPS [1]. Like Soar, CLIPS is a forward-chaining production system based on the RETE matching algorithm. In CLIPS, short term, potentially volatile, knowledge is stored in STM in the form of lists. Each list is given a name, or type, which is essentially the first element in that list. The remaining elements are labeled either explicitly or implicitly by referring to their position in the list. Each element is also a constant value, either numeric or string, and there is no architectural mechanism for referring to the contents of another list, or pointing to another slot.

As in Soar, long-term knowledge is stored as rules that are defined by the programmer. The conditions of these rules match against the contents of STM. Conditions can contain combinations of both constants and variables; however, variables may not be bound to list names or to slot labels. CLIPS, however, is not limited to using simple predicates in the right hand side of a rule as is Soar. A large variety of predefined predicates, as well as user defined predicates and functions can also be used as conditions. The action side of a CLIPS rule is used to modify the contents of STM or to execute external procedures.

CLIPS deploys knowledge via serial rule firings. The basic execution cycle consists of two steps:

- First, rule matches are calculated by comparing the conditions of each rule to the contents of STM.
- Second, successfully matched rules are placed into an ordered list such that the instantiated rule at the top of the list has highest priority.

Priority is defined using two methods. The first of these is a rule level conflict resolution mechanism called salience, which can either be a constant value, or a value calculated at run time. Rules with higher salience are placed higher in the list. In many cases, salience alone is not enough to determine a single highest priority rule. In these cases, CLIPS defers to one of a few user selected architectural mechanisms called search strategies, which orders rules of equal salience. At this point, the first rule in the list is fired and the entire process repeats itself. When no more rules are able to fire, the system halts.

Although there are many similarities between the Soar and CLIPS architectures, the differences are equally significant. These differences occur in each of the three architectural areas we have discussed: knowledge representation, knowledge deployment, and execution cycle. Recall for example, that Soar stores short-term knowledge in a directed graph structure and can perform variable binding on any slot in a memory element. CLIPS, on the other hand, stores short-term knowledge in lists, and cannot bind variables to the list name or to the names of its slots. Moreover Soar fires all matching rules in parallel whereas CLIPS fires only the highest priority rule. An additional difference is that Soar natively supports the decision making process within its execution cycle whereas CLIPS does not.

8.2 Towers of Hanoi revisited

We have examined CLIPS in the Towers of Hanoi domain using the same parameters that were used in our earlier evaluations of the Soar architecture. Note, however that the absolute timing data is not the same as in the first runs. These runs were done on different machines and measure total CPU time, not just Soar kernel time. Below, we briefly review the decision-making strategies of each agent and discuss the particularities of the CLIPS implementation. Notice that two additional categories have been added to further constrain the implementations and to examine areas that may be more amenable to the CLIPS architecture.

- Mutually Exclusive Reactions - Action conditions are mutually exclusive, and no symbol is declared to represent the action being pursued. In both Soar and CLIPS this is done by the construction of individual rules which specificity the preconditions of an action and its effects. Actions are applied sequentially within the world, and the programmer must ensure that no conflicts arise between two actions.

- Simple, Declared Actions - Similar to the first category, but in this case the action being pursued is declaratively represented. In Soar this is done using operators to represent the action. In CLIPS a fact is asserted which describes the current action being pursued. Once again however, the programmer must ensure that action preconditions are mutually exclusive.

- Two-Phase Decision - Two distinct phases are used to make the decision. In the first phase, actions are proposed via declarative symbolic representation. In the second phase one of these actions is selected and pursued. Note that this means that preferences corresponding to a specific action must be expressed simultaneous to the creation of the action symbol (e.g. within the same rule). In Soar, this is done using the architecturally supported decision phase, and the same rule is used both to propose an operator as to express its preference. In CLIPS, partitioning knowledge into a salience hierarchy supports the two phases. This guarantees that the first phase (action proposal) completes before the second phase (selection) begins.

- Three-Phase Decision - Three distinct phases are used to make the decision: proposal, preference and selection. Distinguishing these phases ensures that preferences can be asserted by independent rules, and that these preference rules can make use of knowledge about all currently proposed actions if necessary. Three distinct phases helps support situation dependent preference structures without an explosion of individual rules. CLIPS uses a three-stage salience hierarchy to implement this mechanism.

- Goal Directed - High-level actions, possibly requiring more than one action to complete, are used to constrain rule matching. In CLIPS, goals are maintained declaratively and represented in a stack. Two Soar implementations were examined, one using Soar's native mechanism as demonstrated in the previous trials, and the other using a declarative stack similar to that used in the CLIPS implementation.

Figure 3 shows CLIPS and Soar performance in the Towers of Hanoi domain. Qualitatively, performance is very similar between the architectures except at the end points. On the left-hand side of the graph, the Soar agent

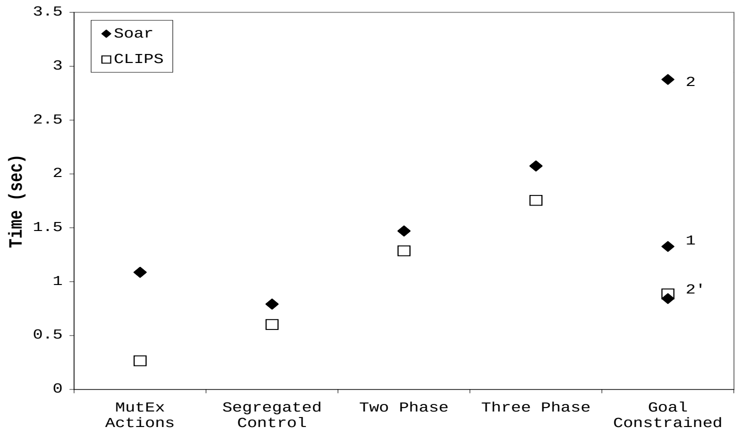

Figure 3: Execution time of Soar vs. CLIPS using various decision-making strategies for Towers of Hanoi.

performs markedly worse than the corresponding CLIPS agent. This performance difference can likely be explained by the fact that this Soar agent does not use the operator construct. As a result, it does not benefit nearly as much from constraining the rule matching as the other Soar agents do, and thus suffers an increase in execution time. At the other end of the graph, Soar and CLIPS behavior are once again divergent. In CLIPS we can attribute the performance increase to the fact that the problem's recursive nature allows the proper puzzle-solving knowledge to be easily expressed with a goal stack, and results in highly constrained rule matching. In the standard version of Soar (point 2), however, we have already seen that the benefits of subgoaling are dominated by the costs of Soar's architecturally supported subgoaling mechanism. However, when performance is re-examined using architecturally supported subgoals in the Soar-Lite variant, point (point 2') or when using a declarative subgoal stack similar in nature to the CLIPS implementation, point (point 1) the difference between the Soar and CLIPS agent's performance is minimal. In all, the similarity of performance between the declarative goal stack implementations in both Soar and CLIPS, and the architectural implementation in Soar-Lite, indicate that in simple environments such as Towers of Hanoi, declarative subgoaling provides a sufficiently lightweight and efficient means of problem solving. As tasks become increasingly complex, however, we expect that the rule-based techniques employed by these implementations will become significantly less efficient than the lightweight architectural counterpart of Soar-Lite.

9. Related Work

Examining differences between agent architectures has received relatively little attention compared to the complementary task of examining how different agent strategies are more or less suited to a particular problem. Nonetheless, a variety of approaches have appeared in the literature. The majority of architectural evaluations can be placed into a single group that we refer to as categorical comparisons [4,6,7,12]. Within this body of work, architectures are evaluated at a high level, in a domain-independent manner, typically based on whether they natively support certain features (e.g. backward or forward chaining, or the ability to make real-time commitments). The benefits of this approach are that the concise tabular data, representative of these studies, may allow architectures to be quickly assessed as having or not having the minimal necessary capabilities to perform the task at hand. Categorical evaluations are most useful when they examine aspects of the architecture that are extremely difficult, or impossible, to emulate using additional, programmer supplied, knowledge. Examples of these aspects may include (depending on particularities of the architectures themselves) the ability to support real-time commitments or distributed processing.

However, the high-level approach of categorical evaluations can also be a short-coming, In particular, because most architectures are Turing complete, it is likely that many architectural features can, in fact, be successfully emulated with addition knowledge. Categorical comparisons, however, rarely comment on this possibility. Moreover, these studies typically do not incorporate benchmarks and as a result, there is no indication as to the relative performance of different architectures or their underlying features.

In contrast to high-level categorical comparisons, the Sisyphus-VT initiative examined the problem of implementing a complex real-world problem on a number of different architectures [11]. Although the pursuit of complex, real world, problems as test bed domains is a laudable undertaking, the implementation overhead is extremely high. As a result, independent teams of programmers who were expert in one particular architecture, carried out the implementations. A critical difference between the methodologies used in the Sisyphus-VT study, and the one we have presented is that we emphasize that the problem solving methods used by two comparable agents should be strictly specified and adhered to. Sisyphus-VT, on the other hand, allowed relative freedom in this area. Although this freedom allows programmers to use a problem solving method which they feel is best suited to their architecture, it also means that differences in two agents' performances might be attributable more to differences in their knowledge, than to differences between the architecture which serve as their foundations. Plant and Salinas attempted to circumvent the problem of confounding the contribution of knowledge and architecture to the overall performance rating in their 1994 study [9]. Under their methodology, agents were constructed in a generic manner so that they had minimal reliance on architecturally specific constructs. This allowed them to create agents for each architecture based primarily on syntactic transformation of a single, handcrafted, agent. This methodology certainly adheres to our requirement of strictly specifying the agent's underlying problem solving methods. At the same time, however, it deviates from our requirements because it does not examine a range of these underlying methods. As a result, it is less likely that the benchmarks will include near-optimal implementations for any architecture, especially since reliance on architecturally specific constructs is purposely minimized.

10. Discussion

The methodology we have presented allows the performance of two architectures, as well as variations of a single architecture to be compared directly. Our methodology is an evolution of prior research, and emphasizes aspects of the benchmark design (e.g. problem-solving specification), which help ensure that agents built using two different architectures use equivalent knowledge. An initial application of our comparative approach has shown significant differences between 3 variations of the standard Soar architecture when Soar's learning capabilities are not required. This hypothesis was further supported by examining the performance of a computer-generated force in the complex, real-world, environment of Quake II. The broader implication of this finding is that knowledge both about the domain and about the implementation of the agent should play a role in deciding what architecture (and what architectural features) are most suitable for a particular circumstance.

We have also shown that the same methodology used to compare variations of a single architecture can also be used to compare two distinct architectures. We have illustrated this application with an initial comparison of Soar and CLIPS. Results from this set of tests indicated both qualitative and quantitative differences in their performance, and have also illustrated the potential performance savings that can be achieved by an architecture whose features are well suited to the current task.

We believe that the work presented in this paper provides a good foundation for addressing the question of what are the resource requirements of architectural properties, or, which properties of an architecture are most suitable for a given situation. Because the needs of computer generated forces often simultaneously push architectures to support a wide array of features and to be highly efficient in terms of run-time performance, an improved understanding of the answers to these basic questions is important.

11. Acknowledgments

This research was funded in part by grant N61339-99-C-0104 from ONR and NAWCTSD.

12. 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] W. B. Gevarter. The nature and evaluation of commercial expert system building tools. Computer, 20(5):24-41, 1987.

[5] J. E. Laird, A. Newell, and P.S. Rosenbloom. Soar: An architecture for general intelligence. Artificial Intelligence, 1987.

[6] J. Lee and S. I. Yoo. Reactive-system approaches to agent architectures. In N.R. Jennings and Y. Lesperance, 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.

[7] W. A. Mettrey. A comparative evaluation of expert system tools. Computer, 24(2): 19-31, 1991.

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

[9] R. T. Plant and J. P. Salinas. Expert system shell benchmarks: The missing comparison factor. Information & Management, 27:89-101, 1994.

[10] S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach, chapter 2, pages 31-52. Prentice-Hall, Upper Saddle River, NJ, 1995.

[11] A. Th. Schreiber and W. P. Birmingham, Editorial: the Sisyphus-VT initiative. International Journal of Human-Computer Studies, 44(3): 275-280, 1996.

[12] 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.

[13] S. A. Wallace and J. E. Laird, Toward a Methodology for AI Architecture Evaluation: Comparing Soar and CLIPS, in N.R. Jennings and Y. Lesperance, 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.

Author Biographies

SCOTT A. WALLACE is a Ph.D. candidate at the University of Michigan's Computer Science program. His research interests include empirical analysis of A.I. architectures, knowledge engineering and machine learning. He received his B.S. in Physics and Mathematics from the University of Michigan in 1996.

JOHN E. LAIRD is a Professor of Electrical Engineering and Computer Science at the University of Michigan. He received his B.S. from the University of Michigan in 1975 and his Ph.D. from Carnegie Mellon University in 1983. He is one of the original developers of the Soar architecture and leads its continued development and evolution. From 1992-1997, he led the development of TacAir-Soar, a real-time expert system that flew all the U.S. fixed-wing air missions in STOW-97.

KAREN J. COULTER is a Systems Research Programmer in the Department of Electrical Engineering and Computer Science at the University of Michigan. She received her BS in Physics from the University of Michigan in 1984 and her MS in Computer Science from the Illinois Institute of Technology in 1993. Since 1997, she has been the principle maintainer of Soar, integrating architectural changes, refining the user interface, and providing software releases and support to the general Soar community. From 1995-1997, she supported development of TacAir-Soar, developed a graphical user interface for configuring missions, and participated in STOW-97.