Abstraction in Problem Solving and Learning*

Amy Unruh
Computer Science Dept.
Stanford University
701 Welch Rd., Bldg. C
Palo Alto, CA 94304

Paul S. Rosenbloom
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292

Abstract

Abstraction has proven to be a powerful tool for controlling the combinatorics of a problem-solving search. It is also of critical importance for learning systems. In this article we present, and evaluate experimentally, a general abstraction method — *impasse-driven abstraction* - which is able to provide necessary assistance to both problem solving and learning. It reduces the amount of time required to solve problems, and the time required to learn new rules. In addition, it results in the acquisition of rules that are more general than would have otherwise been learned.

1 Introduction

Abstraction has proven to be a powerful tool for controlling the combinatorics of a problem-solving search [Korf, 1987]. Problem solving using abstract versions of tasks can provide cost-effective search heuristics and evaluations for the original, or "full", tasks which significantly reduce their computational complexity, and thus make large problems tractable [Gaschnig, 1979, Kibler, 1985, Pearl, 1983, Valtorta, 1981].

Abstraction is also of critical importance for learning systems. Creating abstract rules can reduce the cost of matching the rules, thus improving their operationality [Keller, 1988, Zweben, 1988]. Abstract rules can transfer to a wider range of situations, thus potentially increasing their usability and utility. Abstract rules may also bo easier and/or cheaper to create, thus simplifying the learning process and/or making it more tractable.

In this article we are concerned with abstraction techniques that assist in both problem solving and learning.

*We would like to thank John Laird and Rich Keller for providing valuable ideas and discussions about this research, and Gregg Yost for his help in re-conceptualizing and rewriting RI-Soar. This research was sponsored by the Hughes Aircraft Company Artificial Intelligence Center, and by the Defense Advanced Research Projects Agency (DOD) under contract number N00039-86C-0033. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Hughes Aircraft Company, the Defense Advanced Research Projects Agency, or the US Government.

The four key requirements such a technique should satisfy are:

1. Apply in any domain.
2. Reduce problem solving time.
3. Reduce learning time (therefore help in intractable domains).
4. Increase the transfer of learned rules.

The first requirement implies that the technique must be a *general weak method* that is applicable to domains without additional domain-specific knowledge about how to perform the abstraction. Most problem solvers that utilize abstractions do so only when the appropriate abstractions have been prespecified for them. The second requirement implies that, on average, the time to solve problems with abstraction should be less than the time without. Implicit in this requirement is also that this should be true even if only one problem is being solved; that is, abstraction should help immediately, on the first problem seen in the domain. The third requirement implies that abstraction should be integral to the rule creation process. If the problem-solving time necessary to learn a rule is to be reduced, an approach that simply abstracts the output of the normal learning algorithm will not be sufficient. The fourth requirement implies that abstraction should result in the creation of generalized rules. Even if the rule creation process is a justified method, such as explanation-based learning [Mitchell *et al.*, 1986], this can lead to a form of unjustified induction (though a useful one).

In this article we describe and evaluate an abstraction method which meets these four requirements. The following sections provide a description of the basic method, a discussion of how abstraction propagates through a problem, experimental results from an implementation of the method in two domains, and a set of conclusions and future work.

2 The Abstraction Method

The abstraction method is based on the integration of learning and problem solving found in the Soar system [Laird *et al.*, 1987]. In Soar, problems are solved by search in problem spaces. Decisions are made about how to select problem spaces, states, and operators, plus how to apply operators to states to yield new states (operator implementation). Decisions are normally based on

knowledge retrieved from memory by the firing of productions. However, if this knowledge is inadequate, an *impasse* occurs, which the system then tries to resolve by recursive search in subgoals. This leads to hierarchical processing in which control decisions can be based on multiple levels of look-ahead searches, and complex operators can be implemented by multiple levels of simpler operators (an operator aggregation hierarchy). Learning occurs by converting subgoal-based search into rules that generate comparable results under similar conditions. This *chunking* process is a form of explanation-based learning in which the explanation is derived from a trace of the search that led to the results of the subgoal [Rosenbloom and Laird, 1986].

Abstraction occurs in this framework in the service of control decisions. If an impasse occurs because of a lack of knowledge about how to make a selection, the resulting search is performed abstractly. Consider a simple example from a toy robot domain. Suppose that among the operators in the domain are ones that allow a robot to push a box to a box *(push-box-to-box(BOX1*,130X2)) and go to a box *(goto(BOX)).* The preconditions of the *push-box-to-box*operator are that the robot is next to the first box and that the boxes are in the same room. The precondition of the *goto* operator is that the robot is in the same room as the box that it wants to reach. The goal is to reach a state where the two boxes (boxl and box2) are next to each other. In the initial state, the robot and the two boxes are in the same room together, but at different locations.

Given this initial state, a control decision must be made that will result in the selection of an operator. With partial means-ends control knowledge (encoded as rules), the system can determine that the *push-box-to-b*ox(boxl,box2) operator is one of the possible alternatives, but it may not be able to eliminate all of the other alternatives, leading to an impasse, and thus a subgoal. In the subgoal, a search will be performed by trying out each alternative until one is found that leads to the goal. When the *push-box-to-box* operator is tried, it will fail to apply because one of its preconditions — that the robot is next to the first box — is not met. However, if this precondition is abstracted away, then the operator can apply abstractly, as the robot itself couldn't actually do this and the goal of the abstract search will be achieved. From this abstract search, the information that the *push-box-to-box* operator is the right one to select is returned, and used to make the original control decision. Simultaneously, a control rule is learned which summarizes the lesson of the abstract search. Figure 1 illustrates this process.

Though the basic idea of abstracting within control searches is simple, its consequences are far-reaching. One consequence is that the abstract search is likely to be shorter than the full search would have been because less now needs to be done. If the abstract searches are shorter, yet still return adequate control knowledge, then the time to solve the problem will be reduced (Requirement 2) — as in the toy robot example. Additional consequences arise because learning occurs via the chunking of subgoal-based search. If chunking is done over an abstract search, then the time required to learn about the task is reduced because of the reduced time to generate the explanation (Req. 3). In addition, because abstract searches lead to abstract explanations, the rules acquired by chunking abstract searches will themselves be abstract, and thus be able to transfer to more situations (Req. 4). These generalized control rules effectively form an *abstract plan* for the task. Though these rules may not always be completely correct, limiting abstraction to control decisions ensures that unjustified abstractions will not lead to incorrect behavior — control knowledge in the Soar framework affects only the efficiency with which a goal is achieved, not the correctness.

The actual abstraction of the control search occurs by *impasse-driven abstraction.* When an impasse occurs during the control search, it is resolved by making an assumption, instead further problem solving in another level of subgoals. Impasse-driven abstraction belongs to the general class of abstractions that involve *removing,* or abstracting away, some aspects of the problem in question. (In the taxonomy provided by [Doyle, 1986], our techniques fall under the category of *approximation.)* For example, in the toy robot example above, when the precondition of the *push-box-to-box* operator failed during the control search, leading to an impasse, the system simply assumed that the precondition was met, and continued the abstract search as best it could. (Another way of looking at this is that the system didn't care if the precondition was met). Without abstraction,

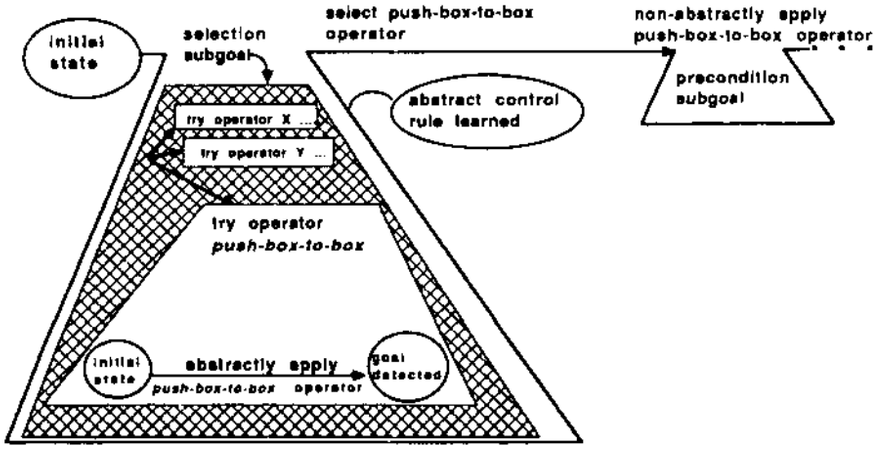

Figure 1: Abstract problem solving and learning (toy robot example).

the impasse would lead to a subgoal in which the system would search for a state to which the operator could legally apply (by applying the *goto* operator).

Impasse-driven abstraction is a general technique that can be applied to arbitrary domains without domain-specific abstraction knowledge; that is, it is a general weak method (Requirement 1). With it, the default abstraction behavior for the problem solver is to abstract away those parts of an operator which are not already compiled into rules, and which therefore generate impasses and require subgoals to achieve. This behavior results in abstraction of operator preconditions and abstraction of operator implementations. The former leads to a form of abstraction similar to that obtained in Abstrips [Sacerdoti, 1974], while the latter leads to behavior that is best described as successive refinement [Stefik, 1981]. As an example of the latter, consider what happens when there is a complex operator for which a complete set of rules does not exist a priori about how to perform it. When such an operator is selected, some rules may fire, but an impasse will still occur because of what is left undone. Without abstraction the system would enter a subgoal where it would complete the implementation by a search with a set of sub-operators. With abstraction, the system assumes that what was done by the rules was all that needed to be done. It then proceeds from the abstract state produced by this abstract-operator implementation.

Another way to understand what impasse-driven abstraction is doing is to look at its effect on the explanation structure created as a byproduct of abstract search (and upon which the learning is based). Figure 2 shows a simplified version of the explanation structure for the toy robot example. Without abstraction, the rule learned from this explanation is:

operator is *push-box-to-box(b1* ,b2) A
in~same-room(bl,b2) A
in-same-room(bl,robot) A
¬ next-to(bl,robot)
=> goal success.

With abstraction, information that would normally be needed for the generation of the result is essentially ignored, and some subtrees of the unabstracted explanation tree — the circled substructure in the figure — no longer need to be expanded for the goal to be "proved". (Another way of looking at this is that some nodes in the proof tree are effectively replaced with the value *TRUE.* Alteration of a proof tree in this manner has been proposed by [Keller, 1988] as a method of forming approximate concepts.) The abstracted rule becomes:

operator is *push-box-to-box(bl* ,b2) A
in-same-room(bl ,b2)
=> goal success.

Alteration of the explanation structure in this way has made the rule more general, and thus able to apply to a larger number of situations.

The same abstraction techniques extend, with no additional mechanism, to multi-level abstraction of both preconditions and implementations. The levels of refinement grow naturally out of the dynamic hierarchy of subgoals that are created during problem solving. Consider multi-level precondition abstraction, for example. In the toy robot problem above, the abstract search that was performed was at the most abstract level — the search was cut off at the highest level of precondition subgoals. Once this search is done, and the *push-box-to-box* operator is selected, it is necessary to do another search to determine what sequence of operators will satisfy its preconditions. In this particular example, the *goto* operator would be among the candidates. Here no impasse of the *goto* operator application would occur, because its preconditions are already met. However, if an impasse did occur during this new search, it would lead to abstraction in the search. This new abstract search is one level more detailed than was the original one. The same cycle continues until a complete plan is generated in which nothing has been abstracted.

Note that there is nothing in the impasse-driven abstraction techniques which prevents the problem solver from making use of additional domain-specific knowledge about what to abstract. The existence of such knowledge can certainly improve performance. However, domain-specific, abstraction knowledge is not often available. If it is not, then the impasse-driven techniques, as a weak method, are able to provide useful abstract problem-solving behavior when it would not otherwise have been possible.

3 Abstraction Propagation

Thus far, we have presented the effects of impasse-driven abstraction on problem solving and learning. However, this is only part of the picture. An important feature

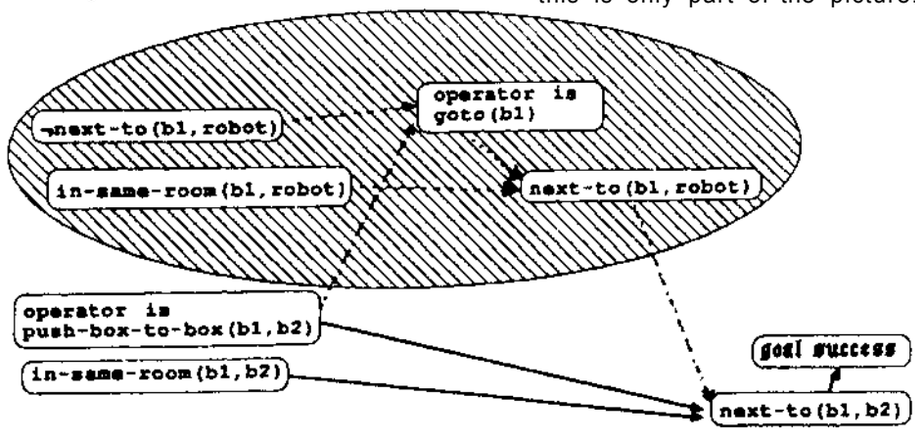

Figure 2: An explanation structure for the next-to(box1, box2) goal.

of impasse-driven abstraction is the way in which the abstraction occurs *dynamically* during problem solving. Each time an impasse occurs during a control search, some aspect of the problem gets abstracted. However, these bits of abstraction initially happen only locally — just because part of one particular operator application gets abstracted during one search step does not necessarily mean that the rest of the problem space will automatically be abstracted in a compatible fashion. Once some part of a problem has been abstracted away, the effects must be *propagated* to later aspects of the problem, including the goal test. Consider, for example, what would happen if the goal in the toy robot domain was to have two boxes adjacent and in the same room, but all of the "in-room" information in the problem space was abstracted away. If the "full", non-abstract goal test was used during abstract search, it would never succeed, and the abstract search would never terminate (unless all options became exhausted, or some monitoring process decided to kill it). It would be more desirable if the goal test of the abstract search was to be compatibly abstracted, so that it cared only about whether the two boxes were adjacent.

The general approach that we have taken is to develop a set of restrictions on the construction of problem spaces which, if followed, ensure appropriate propagation of the abstraction. The two restrictions -— *problem-space factorization* and *assumption-based goal tests* — do not limit what can be expressed, only how it is expressed.

A problem space is factored if it is designed so that the descriptions of problem space components (states, operators, or goals) are separated into any independent subparts which compose them; for example, by creating one production per sub-part. When problem-space components are factored, they may still be partially applicable to the task at hand, even if some of the problem-space knowledge is missing or ignored. For example, if an operator is composed of a number of sub-actions, and if each sub-action is described separately, some of the subactions may be able to apply even though there is not enough information available to allow the operator to apply in its entirety. In this way the operator applies abstractly.

For an example of operator factorization, consider the following simplified "robot domain" operator, which moves a robot through a door to a new room, and in the process keeps track of how many robots are currently in each room. The operator's preconditions are not shown here. If it is true that the operator "may apply", its preconditions are either met or have been ignored through abstraction. Unfactored, the operator is:

```
operator is go-through-door(robot,door,new-room)
  ∧ the operator may apply
  ∧ inroom(robot,old-room)
  ∧ #-of-robots-in-room(old-room,n1)
  ∧ #-of-robots-in-room(new-room,n2)
⇒ add(inroom(robot,new-room))
  ∧ delete(inroom(robot,old-room))
  ∧ change(#-of-robots-in-room(new-room,n2+1))
  ∧ change(#-of-robots-in-room(old-room,n1-1))
```

If the operator is factored, then each independent subaction is considered separately:

```
operator is go-through-door(robot,door,new-room)
  ∧ the operator may apply
  ∧ inroom(robot,old-room)
⇒ delete(inroom(robot,old-room))
and:
operator is go-through-door(robot,door,new-room)
  ∧ the operator may apply
⇒ add(inroom(robot,new-room))
and:
operator is go-through-door(robot,door,new-room)
  ∧ the operator may apply
  ∧ #-of-robots-in-room(new-room,n2)
⇒ change(#-of-robots-in-room(new-room,n2+1))
and:
operator is go-through-door(robot,door,new-room)
  ∧ the operator may apply
  ∧ #-of-robots-in-room(old-room,n1)
⇒ change(#-of-robots-in-room(old-room,n1-1))
```

Thus, if information about the number of robots in either room is not available, the rest of the operator can still apply. Additionally, if because of abstraction the previous location of the robot was unknown, it can still be "moved" to its new room. Factorization enables abstract problem-solving behavior to be propagated dynamically; whatever can be done will be done, while what can't be done because of previously abstracted information is simply ignored. Then, when part of a process is ignored, this in turn may cause new problem-space information to become abstracted. (There is some indication that factorization is not specifically an abstraction issue — if a problem space is factored, then more generalized learning can occur regardless of whether or not abstraction takes place). Note that a factorization determines what *may* be abstracted in a problem space — the set of possible abstractions. It is the impasses that arise during problem solving which determine what actually *is* abstracted.

Assumption-based goal testing refers to the problem-solver's ability to make assumptions about whether or not goals have been achieved during abstract problem solving. To do this, it is necessary to be able to detect that a goal has *not* been met, in addition to being able to detect that it has been met. Under normal circumstances, the problem solver has enough information about a state to determine one or the other; that is, that the state either does or does not achieve the goal. However, when the problem is abstracted, neither test may succeed. Under these conditions, the problem solver needs to make a default assumption as to whether the goal is met or not. Such default assumptions can be made about a goal as a whole, or if it is factored, about individual conjuncts of the goal. To do this properly, the problem solver needs to be supplied with additional information about which goals, and goal conjuncts, should be assumed true and which should be assumed false. Rare termination conditions, for example, should be assumed by default to be unmet. This additional assumption information is not knowledge about what to abstract, or any particular abstraction. Rather, it plays a part in determining the behavior of the system

once abstraction has occurred.

The restrictions which support abstraction propagation are independent of *what* is abstracted, or what is expected to be abstracted. In fact, they are independent of whether problem information is missing because of deliberate abstraction, or because of some other reason (such as bad instrument readings, etc.). Therefore, the problem spaces in which these restrictions have been followed could provide a more robust support for problem solving in noisy domains, and make assumptions based on the best data at hand, regardless of whether or not abstraction is deliberately used.

4 Experimental Results

Experiments have been run with impasse-driven abstraction in two distinct task domains: a Strips-like robot domain and a computer-configuration domain (RI-Soar) [Rosenbloom *et al.*, 1985]. The robot domain is similar to the one in the example presented earlier, but slightly more complicated: there are two robots and two rooms, with two doors between them, as well as two boxes. The RI-Soar computer-configuration domain was based on a re-implementation of a portion of the classic RI expert-system [McDermott, 1982]. The two domains were chosen because they cover both a classical search/planning domain (the robot domain) and a classical expert system domain (computer configuration). Moreover, the domains also differ in that the robot domain stresses abstractions based on operator preconditions, while the RI-Soar domain stresses abstractions based on operator implementations.

To achieve further variation, two different problems were run in the robot domain, with the same goal, but with different initial states. In both problems the conjunctive goal was to have the two boxes pushed next to each other, and to have the two robots "shake hands" (to do this the robots had to be next to each other). The key difference in the initial states was that in the second problem one of the doors was locked, and there was no key. (This second problem should cause some additional complexity if the system abstracts away whether the doors are unlocked.) For each problem, the amount of search-control knowledge that was directly available to the problem solver was also varied. In one version, the problem solver started with means-ends knowledge that allowed it to directly recognize which operators helped solve which subgoals. In the other version, the problem solver could detect when a subgoal had been solved, but knew nothing directly about which operators helped solve which subgoals. In the RI-Soar domain, two computer-configuration problems were also run. Once again, the goal was the same — to have a configured computer — but the initial states were varied.

The problem spaces for these domains were designed according to the restrictions discussed in Section 3. The key issues to be addressed by these experiments are the degree to which irnpasse-driven abstraction meets the four abstraction requirements presented in the introduction.

The first requirement was that the method should be applicable in any domain. The evidence to date is that the abstraction method has been applied to these two quite different domains. In both domains it was possible to apply the impasse-driven abstraction techniques. In the robot domain it was not necessary to add any abstraction-specific knowledge. With RI-Soar, it turned out that although the method was applicable, it was necessary to add a small amount of additional knowledge about the abstraction, to prevent random behavior. RI-Soar is designed so that complex operators are implemented by multiple levels of simpler operators, to form an operator aggregation hierarchy. If abstraction occurred at the level of the top operator, then there was not enough information remaining in the problem space (all configuration work occurred in lower subgoals) to make an informed control decision. That is, the decisions became random. Therefore, we instructed the problem-solver not to abstract at the top level in RI-Soar. Default abstraction behavior at other levels of the operator hierarchy was not affected. It would be preferable for the problem-solver to be able to determine more intelligently (through experimentation, and the current amount of chunked vs. unchunked knowledge in the system), a useful level at which to begin abstraction. We are currently working on an abstraction method which builds on the impasse-driven abstraction techniques, and allows the problem solver to make such a determination.

	No Abstr. (dec.)	Abstr. (dec.)	Abstr./ No Abstr.
ROBOT DOMAIN:			
Problem 1			
Means-ends knowl.	99	80	.81
No means-ends knowl	> 3000	599	< .20
Problem 2			
Means-ends knowl.	120	97	.81
No means-ends knowl.	> 3000	> 3000	
RI-SOAR:			
Problem 3	1260	707	.56
Problem 4	1255	900	.72

Table 1: Number of decisions to solve problems.

The second requirement on the abstraction method was that abstraction should reduce the problem solving time required. Table I shows the number of decisions that the problem solver required to solve each of the problems, and the ratio of the performance with abstraction to that without[1]. Because several of the problems were completely intractable, an arbitrary cutoff was set at 3000 decisions.

The overall trend revealed by these results is that abstraction does reduce the problem solving time, when measured in terms of number of decisions. Moreover, the harder the problem, in terms of the amount of search required without abstraction, the more abstraction helps. Even in the second robot problem, where the problem solver does indeed abstract away the test of whether it can get through the locked door, abstraction helps. It

[2]In the RI-Soar runs, a few chunks learned were altered to avoid problems generated by the way the current version of Soar copies information to new states. This difficulty is unrelated to the abstraction issues, and will be fixed in the next version of Soar.

	No Abstr. (secs.)	Abstr. (secs.)	Abstr./ No Abstr.
Problem 3	4,161	1,535	.37
Problem 4	4,641	2,146	.46

Table 2: Time (seconds) to solve R1-Soar problems.

turns out that this abstraction does not make the problem solver noticeably less efficient when doors are locked, since when it does not use abstraction it is still forced, during its search, to go to the door and try to open it before it realizes this is not possible. What abstraction was not able to do was to make all of the intractable tasks tractable.

Hidden in the RI-Soar numbers is another interesting phenomenon. In problem 4, abstraction reduced the amount of time required to generate a configuration, but the configuration was not as good as the one generated without, abstraction. The goal test for RI-Soar is that there be a complete and correct configuration. Not tested is the cost of the configuration. Instead, RI-Soar uses control knowledge to guide it through the space of partial configurations so that the first complete configuration it reaches is likely to be a cheap one. This use of control knowledge to determine optimality is a soft violation of the constraint that control knowledge not determine correctness, and thus abstraction can (and does) have a negative impact on it. A recoding of RI-Soar to incorporate optimality testing into the goal test could avoid this, or it could simply be lived with as an effort/quality trade-off.

To return to requirement 2, the normal assumption in Soar is that the time per decision is fairly constant, so the decision numbers should be directly convertible into times. However, it turns out that decisions for deep searches are considerable more expensive than ones for shallow searches because of the amount of additional information in the system's working memory. Table 2 shows the actual problem solving times for the two RI-Soar problems, with and without abstraction. These numbers show that when actual run times are compared, the advantage of abstraction is even greater.

The third requirement was that abstraction should reduce the time required to learn. To evaluate this, we need to look at how long it takes to acquire control chunks, with and without abstraction. Table 3 presents the relevant data. It shows the number of decisions that occurred before the control chunk for the first operator tie was learned, for one robot problem and one RI-Soar problem. In both cases, abstraction greatly reduced the amount of effort required before the control rule could be learned.

The fourth requirement was that abstraction should increase the transfer of learned rules. Rather than evaluate transfer directly, what we shall do is illustrate this effect by comparing a corresponding pair of abstract and non-abstract chunks from the robot domain (Figure 3). The two have identical tests up to a point; however, the non-abstract chunk cares whether the robot is next to the box to be pushed, and whether the robots, rooms, and

	No Abstr. (decisions)	Abstr. (decisions)	Abstr./ No Abstr.
ROBOT DOMAIN:			
Problem 1			
Means-ends knowl.	92	19	.21
RI-SOAR:			
Problem 3	1248	101	.08

Table 3: Number of decisions to learn control rule for first operator tie.

ABSTRACT:
If a push-box-to-box operator has been suggested,
and its boxes are pushable,
and the two boxes and one robot are all in the same room,
and the desired state is to have the two boxes adjacent
and to have the robots shake hands,
then mark the operator as being "best".

NON-ABSTRACT:
If a push-box-to-box operator has been suggested,
and its boxes are pushable,
and the two boxes and one robot are all in the same room,
and the desired state is to have the tow boxes adjacent
and to have the robots shake hands
and the operator has the precondition that
the robot be next to the box to be pushed,
and the other robot is in the other room,
and there is a door that connects the two rooms,
and the door is closed and unlocked,
then mark the operator as being "best".

Figure 2: Abstract and non-abstract chunks from the robot domain.

doors are arranged so that the robots will later be able to get together to shake hands. These extra conditions limit the domain of applicability of the non abstract rule with respect to the abstract rule.

Together these experimental results provide support, though not yet conclusive support, for the ability of impasse-driven abstraction to meet the four key requirements on an abstraction method.

5 Conclusions and Future Work

In earlier work we showed how an abstraction, once chosen, could be made to dynamically propagate through a problem space [Unruh *et al.*, 1987]. In this article we have built on that work, by turning it into a general weak method that does not require manual specification of how the problem spaces and goal tests are to be abstracted; the key ideas being impasse-driven abstraction and restrictions on problem space construction. We have also shown how this technique can yield multi-level abstraction and successive refinement.

Another important way that the earlier results have been extended is by the performance of a set of experiments in two task domains. These experiments provided evidence for the satisfaction of four key requirements on an abstraction mechanism: that it should be applicable in any domain, that it should reduce problem solving time, that it should reduce learning time, and that it should increase the transfer of learned rules. However, in the RI-Soar domain, the problem solver was provided with additional abstraction knowledge beyond the default which prevented it from abstracting at the highest

level of the domain's operator hierarchy. This knowledge was necessary to make the abstraction useful; it prevented random control decisions stemming from too little information.

Despite progress with the general weak method presented here, a number of issues remain to be addressed. The most important issue is how the weak method can be strengthened by using additional knowledge about domains and their abstractions. Impasse-driven abstraction does appear to be a plausible technique to use in many situations. Due to the experiential nature of chunking, those parts of the problem space that are familiar will be encoded as compiled knowledge, and thus won't generate the impasses which initiate abstractions. If the heuristic holds that "familiar" is "important", the default abstraction behavior may be quite useful.

But because the current method is weak, there must, be many circumstances in which it will not cause the most appropriate behavior to occur. We plan to try to use the combination of the weak method and experiential learning (chunking) to bootstrap the system to a richer theory of abstractions by learning about the utility of the abstractions that the system tries. One promising avenue of current research is the technique referred to in Section 4, by which the system tries to determine through experimentation a helpful level of abstraction for a given problem context. There are many other ways to learn about an abstraction's utility as well. One possibility is empirical observation over a sequence of related tasks. Alternatively, the problem solver might notice that an abstraction has caused a problem in a particular context, and "explain" to itself why this is the case, using its domain knowledge (failure-driven refinement of the abstraction "theory".) A final option would be for the problem solver to analyze its domain, if it has time to do so, and attempt to come up with a partially pre-processed abstraction theory, as in [Benjamin, 1989, Ellman, 1988, Knoblock, 1989, Tenenberg, 1988].

A second item of future work is the extension of the experiments, both in breadth and depth. We will be looking at abstraction in a number of domains, and trying to empirically evaluate how domain characteristics impact the utility of abstraction.

A final item will be to evaluate the extent to which the restrictions on problem space construction presented in Section 3 can improve the robustness of problem solving in noisy domains.

References

[Benjamin, 1989] D. P. Benjamin. Learning problem-solving abstractions via enablement. In *AAA I Spring Symposium Scries: Planning and Search,* 1989.

[Doyle, 1986] R. Doyle. Constructing and refining causal explanations from an inconsistent domain theory. In *Proceedings of AAAI-86,* pages 538 544, 1986.

[Ellman, 1988] T. Ellman. Approximate theory formation: An explanation-based approach. In *Proceedings of the Seventh National Conference on A rtifical Intelligence,* 1988.

[Gaschnig, 1979] J. Gaschnig. A problem simihlarity approach to devising heuristics. In *Proceedings of IICAh 79,* pages 301-307, 1979.

[Keller, 1988] R. Keller. Learning approximate concept descriptions. Technical Report KSL-88-57, Stanford University, Knowledge Systems Lab, 1988.

[Kibler, 1985] D. Kibler. Generation of heuristics by transforming the problem representation. Technical Report TR-85-20, ICS, 1985.

[Knoblock, 1989] Craig A. Knoblock. Learning hierarchies of abstraction spaces. In *Proceedings of the Sixth International Workshop on Machine Learning.* Morgan Kaufmann, 1989.

[Korf, 1987] R. E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence,* 33:65-88, 1987.

[Laird *et al,* 1987] J. E. Laird, A. Newell, , and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence,* 33:1 -64, 1987.

[McDermott, 1982] J. McDermott. RI: A rule-based configurer of computer systems. *Artificial Intelligence,* 19:39-88, 1982.

[Mitchell *ct al,* 1986] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view *Machine Learning,* 1, 1986.

[Pearl, 1983] J. Pearl. On the discovery and generation of certain heuristics. *AI Magazine,* pages 23-33, 1983.

[Rosenbloom and Laird, 1986] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto Soar. In *Proceedings of AAAI-86,* Philadelphia, 1986.

[Rosenbloom *et al,* 1985] P. S. Rosenbloom, J. E. Laird, .1. McDermott, A. Newell, and E. Orciuch. RI-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 7(5):561-569, 1985.

[Sacerdoti, 1974] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence,* 5:115-135,1974.

[Stefik, 1981] M. Stefik. Planning and meta-planning (molgen: Part 2). *Artificial Intelligence,* 16:141-169, 1981.

[Tenenberg, 1988] J. Tenenberg. *Abstraction in Planning.* PhD thesis, University of Rochester, 1988.

[Unruh *ct al.,* 1987] A. Unruh, P. S. Rosenbloom, and J. E. Laird. Dynamic abstraction problem solving in Soar. In *Proceedings of the AOG/AAAIC Joint Conference,* Dayton, Oil, 1987.

[Valtorta, 1981] M. Valtorta. A result on the computational complexity of heuristic estimates for the A* algorithm. Technical report, University of North Carolina, 1981.

[Zweben, 1988] M. Zweben. Improving operationality with approximate heuristics. In *AAAI Spring Symposium Series: Explanation-Based Learning,* 1988.