Experimentation-Driven Knowledge Acquisition for Planning

Kang Soo Tae
Department of Computer Engineering
Jeonju University, Korea
kstae@www.jeonju.ac.kr

Diane J. Cook and Lawrence B. Holder
Department of Computer Science Engineering
The University of Texas at Arlington
{cook, holder}@cse.uta.edu

Abstract

Knowledge engineering for planning is expensive and the resulting knowledge can be imperfect. To autonomously learn a plan operator definition from environmental feedback, our learning system WISER explores an instantiated literal space using a breadth-first search technique. Each node of the search tree represents a state, a unique subset of the instantiated literal space. A state at the root node is called a seed state. WISER can generate seed states with or without utilizing imperfect expert knowledge. WISER experiments with an operator at each node. The positive state, in which an operator can be successfully executed, constitutes initial preconditions of an operator. We analyze the number of required experiments as a function of the number of the missing preconditions in a seed state. We introduce a naive domain assumption to test only a subset of the exponential state space. Since breadth-first search is expensive, WISER introduces two search techniques to reorder literals at each level of the search tree. We demonstrate performance improvement using the naive domain assumption and literal-ordering heuristics. To learn the effects of an operator, WISER computes

the delta state, composed of the add list and the delete list, and parameterizes it. Unlike previous systems, WISER can handle unbound objects in the delta state. We show that machine-generated effects definitions are often simpler in representation than expert-provided definitions.

1 Introduction

A planning system should have provably rigorous operator definitions for application domains. However, knowledge engineering for planning may be expensive, and the resulting knowledge can be incomplete and incorrect [21]. To automate the acquisition of planning domain knowledge, we are building an experimentation-driven operator learning system, WISER, that is designed and implemented to run on top of CMU's PRODIGY planning system [2].

WISER learns operator preconditions and effects from environmental feedback with or without help from incomplete and incorrect expert knowledge. To learn the preconditions of an operator, WISER first explores a state space to search for a state in which the operator can be successfully executed. The first successful state is called a *positive state*. This state contains all the positive and negative[1] literals necessary, but not necessarily sufficient, to execute the operator. WISER experiments with an operator in each node of the breadth-first search tree to detect missing positive or negative necessary literals in a given state. We simulate expert knowledge and generate an expert-assisted state to be used as a root node. The number of missing necessary literals in a root node is crucial in the performance of a system using a breadth-first search. We will present an analysis of the number of required experiments to learn an operator definition as a function of the missing necessary literals. We introduce a *naive*

[1] The literals which should be absent.

domain assumption to reduce the size of the search space. Since there is a huge range between the best and the worst case computation requirements of breadth-first search, we introduce two search techniques that reorder literals at each level of the search tree to reduce the number of experiments. We will compare analytical and empirical results and show the performance improvement obtained by introducing the assumption and the heuristics respectively.

The initial set of preconditions learned from a positive state is overly-specific. WISER refines this definition by removing irrelevant literals and introducing necessary negative literals. The refined definition is then type-generalized for the relevant parameters of the operator. To learn the effects of an operator, WISER computes the delta state, the difference between the previous state and the resulting state after the operator execution. WISER learns operator effects by parameterizing the delta state. Unlike OBSERVER [22] and the method introduced by Oates and Cohen [13], WISER handles objects which are not bound in the preconditions by utilizing an unbound variable as in the expert-provided definition. We will show the advantage of using the machine-generated definition of the effects of an operator over the expert-provided definition which can be difficult to comprehend.

2 Related Work

Much of the existing research in planning assumes that domain operators are perfect. However, since in reality those expert-provided theories are incomplete, a fundamentally important area in planning is to autonomously learn and/or refine incomplete domain knowledge. Carbonell and Gil [3, 8] explain a plan failure in terms of the inherent incomplete domain knowledge. Gil's EXPO system refines the incomplete definitions by discovering missing literals through experiments when a planner's internal expectation and the external observation of the simulated environment differ.

EXPO is applicable only for incomplete knowledge.

Wang's OBSERVER [22], running on top of the PRODIGY system, like EXPO, autonomously learns a new operator by observing an expert's problem solving method. The purpose of OBSERVER is to avoid the knowledge engineering problem. An initial incomplete operator is refined through the integrated cycles of planning, execution and plan repair. OBSERVER is applicable for both incomplete and incorrect knowledge. Learning in OBSERVER requires many training data sets or observations. Learning in WISER requires smaller training data sets by introducing the naive domain assumption.

desJardins [5] uses an operator editor to build (and modify) incomplete operators and employs an inductive learning system to acquire planning knowledge for missing preconditions via feedback from simulators or users. The learning system, however, does not generalize redundant arguments, and it learns a new operator only from a user. Benson's agent [1] adopts an inductive logic programming algorithm to learn operators from its own experience and from its observation of a domain expert. In the presence of noisy instances, the agent produces approximate preconditions of an operator, called a *preimage* of an action model. It cannot, however, deal with noise problems caused by unavoidable perceptual aliasing [23].

Oates and Cohen [13] learn planning operators with context-dependent and probabilistic effects. They randomly explore the operator space, composed of precedent and successor states, and record state descriptions to find the dependencies between pairs of precedent and successor states that reflect the true structure of the agent's interactions with its environment. Their method is a kind of observation-based learning that passively processes a given stream of data just as Wang does [22]. Wang uses 2-value logic and cannot learn negative preconditions. Oates and Cohen use 3-value logic in learning operators: (+ - *). A sensor is equivalent to a predicate or a proposition in

our model, assuming one of two values: "+" or "-". They expand a sensor to include the value "*", which represents irrelevancy. A multitoken space, composed of the cross product of all the possible values of the tokens, is equivalent to a state space in our model. Their method performs a general-to-specific best-first search. The symbol "*" contributes significantly to the complexity explosion. As the search space gets larger, their method uses pruning techniques. However, the techniques are only related to handling "*". Recognizing which sensor is irrelevant to an operator in advance is difficult. In our approach, we use a two-value logic and can deduce "*" by experimenting with two opposite cases. The size of search space is significantly reduced.

EITHER [15] and RAPTURE [11] are theory revision systems. Given an imperfect expert-supplied rule base and a set of correctly labeled examples, EITHER sends the false positive and the false negative examples to ID3 [16], an inductive machine learning method using an information gain heuristic. EITHER revises its rule base as follows: False positive examples are specialized by removing rules from the rule base and adding conjuncts to rules. False negative examples are generalized by adding new rules, deleting conjuncts from rules, or adding new disjuncts. When examples cannot be classified correctly, RAPTURE [11] uses ID3's information gain heuristic to learn new terms and to change the structure of the rule base. Even though there has been much research in the area of rule revision, little of this work relates to extending the research to the area of planning. Our work represents an attempt to integrate these two areas.

3 The WISER Learning System

The power of a system depends on the expressibility of its description language. WISER's language uses the strongly-typed first order logic used in the PRODIGY system. The type hierarchy forms a tree structure. Every object in a domain is speci-

fied by its type. A literal has n typed arguments and is assigned a Boolean value. A state describes a world state using a conjunction of instantiated literals. The built-in robotic actions are functions that map one state to another state. The operators, modeling the actions, are *if-then* rules specifying the legal robotic actions in terms of the preconditions and the effects. An operator description includes a list of type-constrained parameters, $op(v_1, .., v_m)$, for $m \geq 0$. The generalized parameters enable a single operator definition to apply to a range of situations [2].

To learn operator definitions, WISER adopts Craven and Shavlik's concept learning method which employs two oracles [4]: the EXAMPLES oracle to produce, on demand, examples and the SUBSET oracle to answer queries about the membership of the generated examples for target concepts. For WISER, the Gen-States oracle generates, on demand, a state as an example. When an operator is queried (executed) by WISER in an oracle-generated state, the Environ oracle, like SUBSET, returns feedback from the environment answering whether the execution (query) is successful and thus the state is a positive example for the operator. Feedback, either positive or negative, acts like the membership function of SUBSET. Each execution of Environ forms an experiment, which will be defined later in the paper.

4 Designing the Experiments

4.1 Two Types of Seed States

A state in which a robot experiments with an operator is called an initial state *IS*. If *IS* contains all the necessary positive and negative preconditions of the operator, the operator can be *successfully* executed, and the state is called a positive state. WISER searches for a positive state from the space of states. The root node of the search tree is a seed state, $\hat{S}$. The missing positive and negative necessary literals in $\hat{S}$ are called

errors of a state. A positive state contains no errors.

The Gen-States oracle generates a new *IS* by changing some literals in $\hat{S}$. Each *IS* constitutes a node in a tree. A seed state usually contains a number of errors. The *number* of errors in $\hat{S}$ is important in WISER's performance due to using breadth-first search. WISER can utilize two types of seed states. The first type is an arbitrary state in the environment, which represents the current observable state. Oates and Cohen [13] use this approach. This type of state is simple to generate but is likely to have a large number of errors. This state is used for experimentation as the raw data when no background knowledge is available. The second type is generated by transforming the raw data into a less error-prone state by exploiting the expert's imperfect knowledge.

4.2 Method to Generate a Probabilistic State

The expert's imperfect knowledge is probabilistically generated. To provide an experimental platform for this paper, we simulate an expert's approximate (or imperfect) knowledge AK by initially assuming idealistically perfect domain knowledge *IK* as in Gil's approach [8]. Given a set of operators $Ops = (op_1, .., op_n)$ and literals $Lits = (l_1, .., l_m)$, their cross product, (Lits × Ops), generates a space Ψ of mn (literal, operator) pairs. A pair $\psi_{ij} \in \Psi$, referring to l_i mapped to op_j, is assigned a value by a mapping function: $M(\psi_{ij}) \rightarrow V \in \{+ - *\}$. V can have one of three possible values: positive(+), negative(-), or irrelevant(*). A positive value means that the literal is a positive precondition of the corresponding operator. Let v_{ij} be the value of ψ_{ij} in *IK*, and v'_{ij} be the value of ψ_{ij} in AK. v_{ij} is always correct in *IK*. We probabilistically map v_{ij} to v'_{ij}. To model the imperfect knowledge, we randomly select ε pairs from Ψ and inject error to their values in AK. The imperfect knowledge can be either incomplete and/or incorrect. If v is positive and v' is negative, we call the knowledge incomplete. If v is negative and v' is positive, we call it incorrect. An expert assigns a probability

to each mapped value v'. The probability reflects the expert's certainty of the mapped value $v'_{ij} \in \{+, -, *\}$ for ψ_{ij}.

When assigning probability to each v', if AK is as accurate as IK, the v'_{ij} will denote a true relation of a literal l_i as a precondition for op_j, and v' will be assigned with 1.0 probability. If the expert's knowledge is almost perfect, the probability for v' is assigned near 1.0. As the knowledge becomes less perfect, the probability for v' decreases. Since AK is imperfect, v' will be assigned with less than 1.0 probability. l of ψ may or may not be injected with error and be assumed incorrectly to be $\neg l$ by an expert. If v is $+$ in IK and the expert thinks incorrectly that v' is $-$ in AK, $\neg l$ may be assigned to v' in AK with a probability far less than 1.0. WISER heuristically generates a probably-best seed state S for an operator op_j, $1 \leq j \leq n$, based on v'_{ij}, $1 \leq i \leq m$ in AK: If v'_{ij} is positive, l_i is in S, and if v'_{ij} is negative, l_i is not in S for op_j.

4.3 Three schemes of Assigning Probability

Suppose the expert knows the domain well. If he/she provides a statement *S1* with high certainty, we can say that the statement is very reliable. If the expert is not certain about another statement *S2*, *S2* is less reliable. In contrast, if an expert's statement is actually true in IK, he/she is more likely to be certain of the same statement in AK. To model the imperfect knowledge, A probabilistically biased wheel is used as follows:

$$P_i = \frac{i^k}{\sum_{j=1}^{100} j^k} \tag{1}$$

where i is the probability, $0 \leq i \leq 1$, and k is a constant. An observer (or learner in this case) will judge that a statement is more likely to be true as he/she is more certain. Even though both statements are correct, if something goes wrong, an observer will suspect S2 first.

No matter the level of knowledge, the expert's knowledge can still contain errors in a complex domain. When we need to detect errors and are given two probabilities,

ϕ_1 and ϕ_2, $\phi_1 > \phi_2$, we suspect that ϕ_2 is more likely to be the source of the error and check ϕ_2 first. We use three types of incorrect expert knowledge in this paper: accurate, uniform and blurred. We are using three types of biased wheels to model the types of incorrect expert knowledge and compare how quickly each can detect the incorrect knowledge.

First, suppose the expert knows a domain very well (accurate expert knowledge). When the statement is actually false in *IK*, the expert will find the errors rather quickly because the expert's certainty of the statement is likely to be very low in AK, which contrasts with the high certainty of the correct statement. The following wheel of distribution models this incorrect mapping:

$$P_i = 1 - \frac{i^k}{\sum_{j=1}^{100} j^k} \tag{2}$$

On the other hand, suppose the expert does not know the domain well (blurred expert knowledge). When the statement is actually false in *IK*, the expert cannot find the errors quickly. The expert's certainty for an incorrect statement is likely to be as high as that for a correct statement in AK. and the expert may suspect another source of the error. In the extreme case, it may not be relevant how certain the expert is of a statement. To model the fact that the expert cannot quickly distinguish between the correct and incorrect statements quickly, we use the same wheel used in mapping the correct statement in (1).

$$P_i = \frac{i^k}{\sum_{j=1}^{100} j^k} \tag{3}$$

Thus, the blurred knowledge does not give any information.

As a compromise between these two cases, a uniform distribution is used to assign probability uniformly.

$$P_i = \frac{1}{100} \tag{4}$$

As shown later, the behavior modeling the accurate knowledge is close to the best case

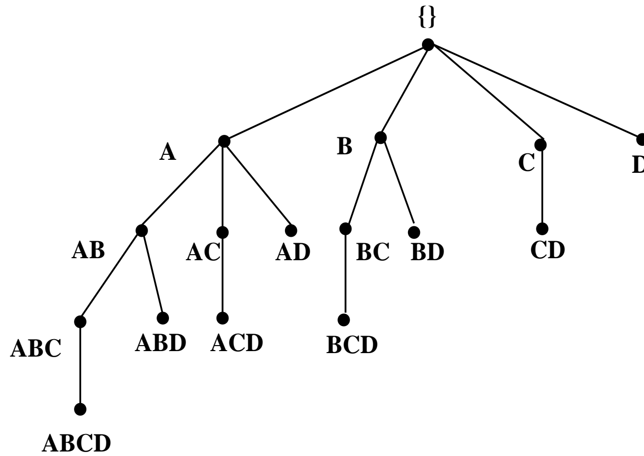

Figure 1: Breadth-first Error Search Tree

of the theoretical analysis, and the behavior modeling the blurred knowledge is close to the average case.

4.4 Size of Search Space

Oates and Cohen rely on random exploration to learn operators assuming that it is easy to observe a positive example in a natural state. However, as an operator gets more complicated in the real world, finding a positive example (or a correct hypothesis) can become very expensive or even impossible. For this kind of problem, OBSERVER relies directly on an expert. However, the expert's knowledge may not be available. To find a positive state, WISER uses an experimentation-driven breadth-first search as shown in Figure 1. When the root node is not a positive state, WISER needs a set of states on which to experiment. This section will define the size of the space of all possible initial states.

If all the parameters of a literal are instantiated, the literal is an instantiated literal. Otherwise, it is a parameterized literal. Let n represent the number of parameterized literals known to the learner. A state S is represented by a subset of the instantiated literals L*. $\bar{S} \subseteq$ L* contains literals which are not true in S. In other words, L* is

composed of a state and its complement: L*=$S \cup \bar{S}$. We know that L* is given for a system. Since $\bar{S}$ can be derived from L* and S, the state space constitutes the learning space for WISER. Gil [8] calls this space a hypothesis space. Wang does not include the negative literals in $S \cup \bar{S}$ in the search. The problem with this approach is described in Section 6.2 of our paper. Oates and Cohen include irrelevant literals in the search, thus the size of the search space is overwhelming.

Given n parameterized literals L=$\{l_1, ..., l_n\}$, let the number of parameters for a literal l_i, $1 \leq i \leq n$, be $|arg_i|$. Let the jth parameter, $param_{i_j}$, be constrained to have $|type_{i_j}|$ possible types. Let the kth type of $param_{i_j}$, $ty_{i_{j_k}}$, have $|ty_{i_{j_k}}|$ objects. N, the number of all instantiated literals L*, is

$$N = |L*| = \sum_{i=1}^{n} \prod_{j=1}^{|arg_i|} \sum_{k=1}^{|type_{i_j}|} |ty_{i_{j_k}}|. \quad (5)$$

Each instantiated literal has a Boolean value in a state and the size of the state space is 2^N, which grows exponentially as a function of N.

4.5 Generating Training Examples

Since the size of the state space is exponential, an intelligent system should carefully design a series of experiments, testing different subsets of L* that are relevant to learning. To actively explore its hypothesis space, WISER requests, in a non-random manner, Gen-States to change the value of certain requested literals in $\hat{S}$. The requested literals are generated as a function of WISER's knowledge K_{op} about the operator: Request-lits(K_{op}). K_{op} can be empty, injected from an expert or *IK*, or acquired through practice. *IK* is not available to WISER. We will later compare performance of these three approaches. WISER applies the operator to the oracle-generated state: Gen-States($\hat{S}$, Request-lits(K_{op})). For example, if $\hat{S}$ is {ABCD} and Request-lits(K_{op}) is {AB}, then Gen-States($\hat{S}$, Request-lits(K_{op})) is $\{\bar{A}\bar{B}$CD}.

A learner-specified state constitutes an initial state IS for an experiment. In using the Strips assumption, an operator fires only if the preconditions are met. For the purpose of learning operator definitions, we remove this assumption and an operator can attempt to fire in any state. In WISER, application of an operator *op* is represented as a function from a state S to another state defined by the following operation on S:

$$\begin{cases} \text{Apply(op, S)} = \{\text{S} \cup \text{add(op)} \setminus \text{del(op)}\}, & \text{if pre(op)} \subset \text{S} \\ \text{Apply(op, S)} = \text{S}, & \text{if pre(op)} \not\subset \text{S} \end{cases}$$

The application of an operator in IS constitutes an experiment. If all the actual preconditions are met, the operator is successfully executed in IS, generating a new state NS. If the preconditions are not met, the operator is not executed and generates no new state. The Environ oracle returns true to WISER if a new state is generated; otherwise, Environ returns false:

$$Environ(op, IS, NS) = \left\{ \begin{array}{ll} true, & \text{if IS} \neq \text{NS}, \\ false, & \text{otherwise.} \end{array} \right\}$$

An experiment is formally defined by a triple composed of an operator, an initial state, and the environmental feedback: (OP, Gen-States($\hat{S}$, Request-lits(K_{op})), Environ(OP, IS, NS)). WISER generates new training examples until a positive state is encountered [5].

4.6 Complexity of Learning Initial Preconditions

Learning an operator's preconditions is cast as a search problem for finding relevant literals from L* [13, 22]. An example from the state space constitutes a hypothesis and can be either positive or negative. A positive example always contains all the necessary (positive and negative) preconditions and is sufficient for learning the operator preconditions [20, 22].

WISER executes an experiment in each node of the search tree as shown in Figure 1. To perform an efficient systematic search over the space, children of search nodes are expanded in a manner that ensures that no node can ever be generated more than once. To analyze the complexity of WISER's learning behavior, we calculate the number of necessary experiments as a function of the number of errors in the root state. To detect the errors in the seed state, WISER at first changes a single literal at the first level of the search tree. If it cannot find a positive state by changing i, $i < N$ literals at the i-th level, it changes 1 more $(i+1)$ literals at the next level. If there are ε errors in $\hat{S}$, for $0 \leq \varepsilon \leq N$, the depth of the search tree is ε, and the number of experiments required at the ε level is 1 in the best case and $\binom{N}{\varepsilon}$ in the worst case. We use a random-number generator in our simulator to make every combination of ε literals at the ε level equally likely to contain errors. Thus, the average case is calculated as follows:

$$\frac{\sum_{i=1}^{\binom{N}{\varepsilon}} i}{\binom{N}{\varepsilon}}$$

or, since each node containing a combination is monotonically increased by 1 in the breadth-first search, it is the average of the best and the worst cases:

$$\frac{\binom{N}{\varepsilon}+1}{2}$$

In addition, the number of unsuccessful experiments summed from the first through (ε − 1) levels is $\sum_{i=1}^{\varepsilon-1}\binom{N}{i}$. Thus the total number of experiments τ is in the range

$$\sum_{i=1}^{\varepsilon-1}\binom{N}{i} < \tau \leq \sum_{i=1}^{\varepsilon}\binom{N}{i}. \tag{6}$$

The average case is

$$\frac{\left(\sum_{i=1}^{\varepsilon-1}\binom{N}{i}+1\right)+\sum_{i=1}^{\varepsilon}\binom{N}{i}}{2} \tag{7}$$

We will compare this average case analysis with the empirical results later. The average case is a good estimate as will be shown.

Interestingly, Oates and Cohen use the same search tree in Figure 1 for automatically learning planning operators. However, while we explore a state space to find

possible missing preconditions, they explore the operator space to find the dependencies between pairs of precedent and successor states. Since their method handles two states at each node, the search space is twice as big: $(2^N)^2$. In a deterministic approach, each operator contains one and only one successor state. Observe that our state search tree is a special case of the operator search tree. We will compare the two approaches.

First, Oates and Cohen execute random actions and record state descriptions to find the true structure of the agent's interactions with its environments. They assume that it is easy to find positive states for an operator. To learn the probabilistic effects for the operator, they use a heuristic evaluation function which counts the number of times in history that the precursor of a node, which denotes the preconditions, is followed by the successor, which denotes the effects [13]. However, if the observed data does not contain the true structure, it can be hard to generate a dependency that we want to learn. They point out that a random exploration may be extremely inefficient in this situation. Our search is a non-random exploration. We use only one example and generalize using the naive domain assumption as explained later.

Second, their method is a kind of observation-based learning that passively processes a stream of data. Wang uses 2-value logic and cannot learn negative preconditions. In contrast, Oates and Cohen use 3-value logic in learning ops: (+ - *). A sensor is equivalent to a predicate or a proposition in our model. A sensor can assume one of two values: V or not-V. They expand a sensor to a token which can take 3 values by including "*". Let a sensor L1 assume values from {A, not-A}, and another sensor L2 be assigned values from {B not-B}. A token L1* can take a value from {A, not-A, *}, where "*" represents an irrelevant sensor. A multitoken space is composed of the cross product of all the possible values of the tokens. The size of multitoken space, composed of two tokens L1* $\times$ L2*, is 9. A multitoken, composed of a set of token

values, is equivalent to a state in our model, represented as a node in the search tree shown in Figure 1. If a multitoken contains more "*" symbols, it is more general. The root node is composed of a set of "*" symbols corresponding to each sensor. Oats and Cohen perform a general-to-specific best-first search starting from a root node.

The use of "*" contributes significantly to the complexity explosion. Pruning techniques are used to reduce the size of the search space, but these techniques only handle "*" symbols. One of the problems is that it is hard to know which sensor is irrelevant to an operator in advance — instead, the "*" symbol may be more appropriately used to represent unknown or noisy data, similar to the use of null values in databases [24, 14]. In our approach, "*" is not provided in a state description. We deduce "*" by experimenting with two opposite cases: if both states, (A B) and (A not-B), have the same effect on an operator, then B is irrelevant to the operator and we can deduce (A *). Since we can deduce "*", the size of search space is reduced to 4 from 9 by using the state space, L1 $\times$ L2, instead of the token space.

5 Naive Domain Method

Since a robotic experiment is costly, a method to reduce τ in Equation (6) is imperative. We will use the number of errors ε as the independent variable in our experiments. To reduce N, we introduce a *naive domain assumption* to select a subset of L*.

Imagine a simple world with 3 boxes: BOXa, BOXb, and BOXc. To learn the operator (Pickup box), a learner may experiment with every box. However, if the learner has limited resources and time, this method is very impractical in the real world. When the world is regular [18], where you can assume that each box is very similar to other boxes, the learner may experiment with only one box to learn the operator. WISER assume a regular world. If we consider the relation among the objects of the BOX type with respect to the operator, we can notice that the boxes

are functionally homogeneous [20].

Definition: A homogeneous relation $\sim$ on the set of objects in a type exhibits the equivalence relation.

The equivalence relation among boxes measures the functional equality of the boxes with respect to the operator, namely pickable. If $\sim$ is satisfied on a subset of ty, defined by $[obj_e]=\{obj_i \in ty | obj_e \sim obj_i\}$, then $[obj_e]$ forms the subtype of the type. The n equivalence relations, $\{[obj_1], .., [obj_n]\}$, partition the set of objects of a type into n subtypes. We can choose any object randomly from a subtype $[obj_e]$, and use it as a typical instance for the subtype. We will call a chosen object a target object. The target objects are bound to the parameters of an instantiated operator at run time. If a target object is experimented as a parameter of the operator, no additional objects of the subtype need to be tested further.

Definition: A *naive domain* is a domain where every object in a type belongs to one and only one subtype.

WISER adopts the assumption of the *naive domain* to learn an operator definition from one example. Only one instantiated literal needs to be tested and parameterized for each type. This abstraction drastically reduces the size of the search space. Gil and Wang also adopt a version of the naive domain assumption by requiring only two objects to generalize all the objects of a type [20]. In our approach, when only one object of a type is considered in a naive domain, such that $|ty_{i_{j_k}}| = 1$, the number of instantiations for n parameterized literals is

$$\lambda = \sum_{i=1}^{n} \prod_{j=1}^{|arg_i|} \sum_{k=1}^{|type_{i_j}|} |ty_{i_{j_k}}| = \sum_{i=1}^{n} \prod_{j=1}^{|arg_i|} |type_{i_j}|. \tag{8}$$

The complexity of the naive state space is 2^{λ} instead of 2^{N} in (5), and $\lambda \ll$ N as n, the number of parameterized literals, grows.

The naive domain assumption forces users of the system to supply type hierarchies

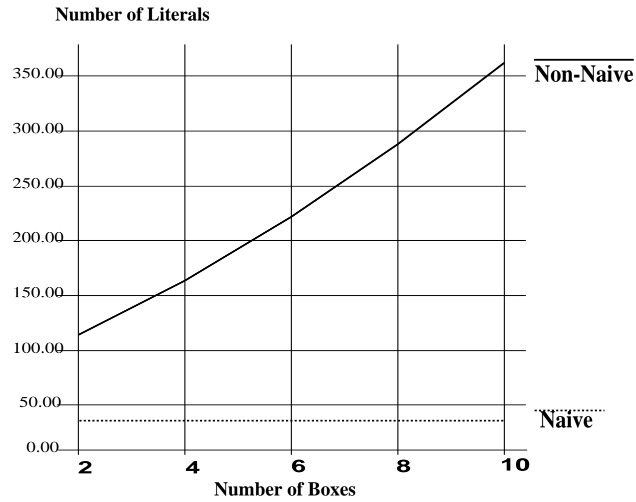

Figure 2: Comparison of the number of literals in Extended Strips domain

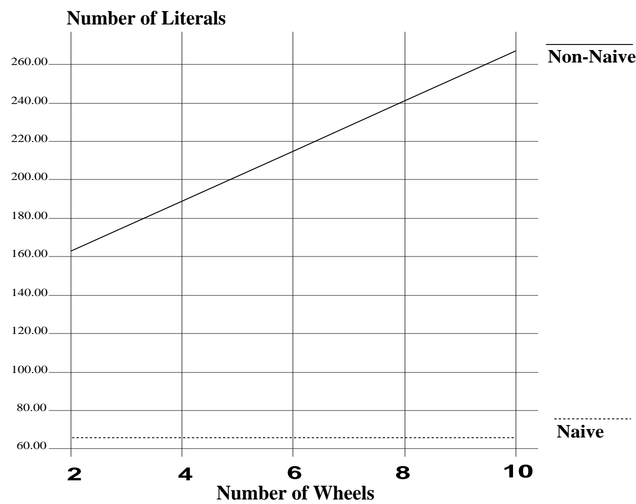

Figure 3: Comparison of the number of literals in Flat Tire domain

for each domain of application. In addition, the type hierarchies must be detailed enough to guarantee that each object within a particular class is functionally equivalent to all other objects in the same class. In complex domains with a great variety of objects, this assumption may be unrealistic. Two possible methods of overcoming the limitation are to add uncertainty values to operator descriptions based on the likelihood that an object of a particular class will behave in a predictable way (Kushmerick et al describes an approach for planning with probabilistic operators [9]), or to use C4.5 to learn more detailed object class descriptions based on experience (we will describe this method later in the paper).

5.1 Performance Evaluation

To show the domain-independence of the naive domain assumption, we tested WISER in multiple domains. We modified the original Extended-Strips domain and the Flat Tire domain in the PRODIGY system to include more literals. The world in the modified Extended-Strips domain has a set of interconnected rooms containing a robot, boxes and keys. Doors can be open or closed and locked or unlocked. The robot can open a door and move objects from one room to another. This domain has 36 parameterized literals. Each literal has 2 arguments on average. In the modified Flat Tire domain, the world is composed of a robot, wheels, a container, and many tools. A robot can open the container and fetch the tools from the container to fix a flat tire. The robot uses a jack to lift a hub on the ground and uses a wrench to loosen or tighten a nut on a hub. It removes a wheel from a hub and uses a pump to inflate a tire. This domain has 66 parameterized literals. Each literal has 1 argument on average.

Figure 2 shows that in the Extended-Strips domain, all other conditions being equal, as the number of objects of the box type increases, N increases drastically but λ stays the same. Figure 3 shows the same phenomena in the Flat Tire domain.

We have previously performed a theoretical analysis to derive τ as a function of ε and N. We will now compare the empirical results with the theoretical analysis in the two domains and demonstrate that we have almost identical results for both domains. We will also show the effects of using the naive domain assumption. In each of these experiments we will measure the number of needed experiments as the number of errors increases from 1 to 5 (this is sufficient to see the trends). The number of literals varies in each case as is dictated by the nature of the problem.

The preconditions of the Unlock-door operator in the modified Extended-Strips Domain contain 9 positive literals and 3 negative literals: (INROOM key room), (INROOM robot room), (IS-KEY door key), (DR-TO-RM door room), (HOLDING key), (DR-CLOSED door), (LOCKED door), (NEXT-TO door robot), (NEXT-TO robot door), ¬(DR-OPEN door), ¬(ARM-EMPTY), and ¬(UNLOCKED door).

We ran WISER to learn the operator preconditions for cases of using 4-Boxes and 8-Boxes, both in the naive domains and in the non-naive domains. Note that both of these cases will perform like 1-Box in the naive domains because we are only concerned with a target object.

Figure 4 shows the empirical results, averaged over 5 tests, in a non-naive domain with 4 boxes. For example, for ε=3 and N=164, the number of experiments τ is theoretically in the range of

$$\sum_{i=1}^{2} \binom{164}{i} < \tau \leq \sum_{i=1}^{3} \binom{164}{i}$$

The figure shows that there is a huge range between the best case and the worst case. It shows that empirical result matches closely to the average case of the formal analysis of WISER. The figure also indicates that τ drastically increases as a function of ε, for $0 \leq \varepsilon \leq 5$.

Next, in the naive domains of using 4-Boxes, and 8-Boxes for ε=3 and N=36, the

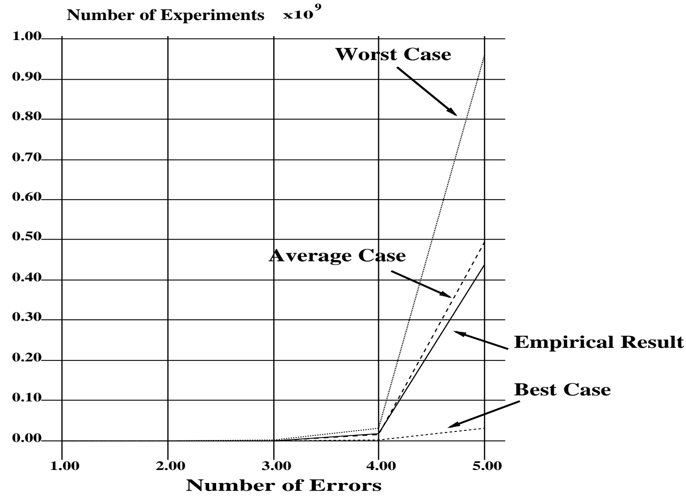

Figure 4: Formal Analysis and Empirical Result in 4-Boxes Model

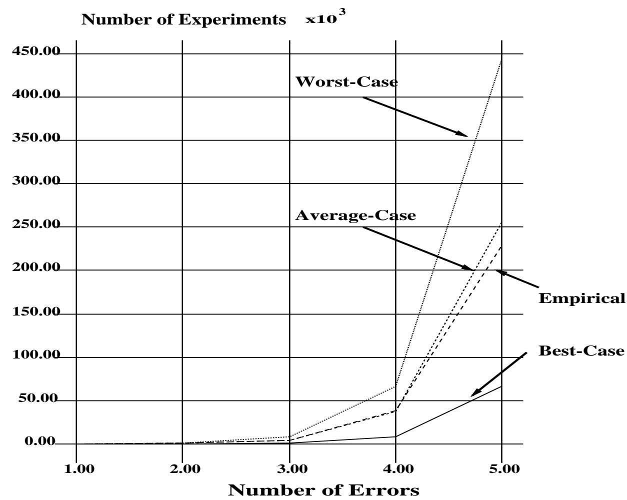

Figure 5: Formal Analysis and Empirical Result in Naive E-Strips Domain

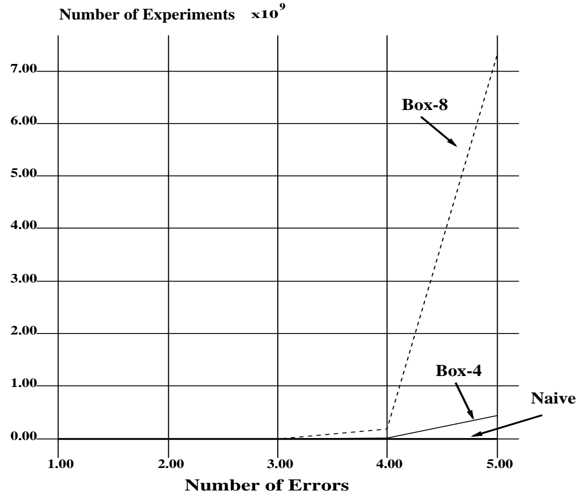

Figure 6: Comparison of Naive, Box4, and Box8 Domains

number of experiments is theoretically in the range of

$$\sum_{i=1}^{3-1} \binom{36}{i} < \tau \leq \sum_{i=1}^{3} \binom{36}{i}$$

Figure 5 shows that the empirical result matches closely to the average case of the formal analysis and that τ drastically increases as the function of ε.

Finally, we compare the system's performances in three cases: 1-Box (Naive), 4-Boxes and 8-Boxes, where N=288 as shown in Figure 2. The empirical results reveals the drastic effect of using a naive domain in Figure 6. (Naive performance overlaps the x-axis.)

We ran similar experiments to learn the Fetch operator in the modified Flat Tire Domain. The preconditions of the Fetch operator contain 3 positive literals and 4 negative literals: (arm-empty), (¬ (out nut container)), (in nut container), (¬ (slippery nut)), (¬ (closed container)), (¬ (glued container)), and (open container).

First, we tested in the non-naive domain with 4 wheels, where N=189. Figure 7 shows that the empirical results match a little less closely to the average case of the formal analysis of WISER. They also indicate that τ drastically increases as a function of ε as in the above domain. For ε=3 and N=189, the number of experiments is

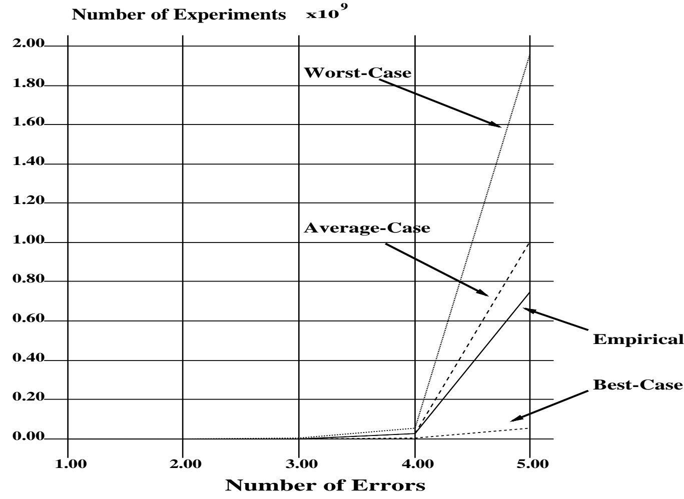

Figure 7: Formal and Empirical Results in Non-Naive Models in Flat Tire Domain

theoretically in the range of

$$\sum_{i=1}^{3-1} \binom{189}{i} < \tau \leq \sum_{i=1}^{3} \binom{189}{i}$$

Figure 8 shows similar results in the naive domain. Finally, we compare the system's performances in three domains: 1-wheel (naive), 4-wheels and 8-wheels where N=241 as shown in Figure 3. The empirical results demonstrates the drastic effect of using a naive domain in Figure 9.

6 Heuristics of Ordering Literals

We note that there is a huge difference in the number of experiments between the best case and the worst case: $1 \leq \tau \leq \binom{\lambda}{\varepsilon}$ in the ε-th level of search. We introduce two heuristics to order the literals so that WISER can search the probably most relevant literals first at each level of the search tree.

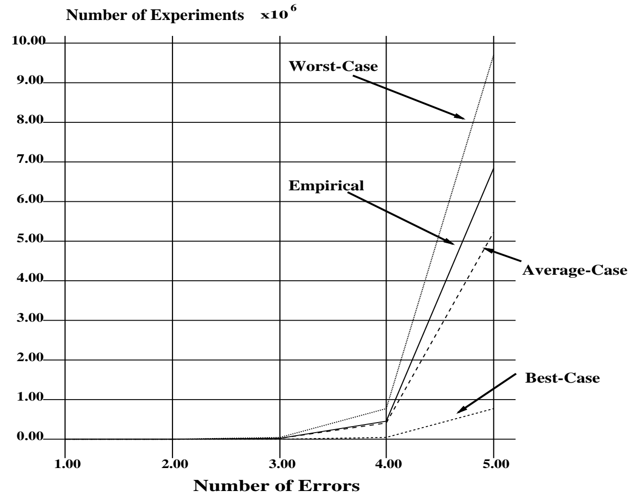

Figure 8: Formal and Empirical Results for τ in Tire Domain Naive Models

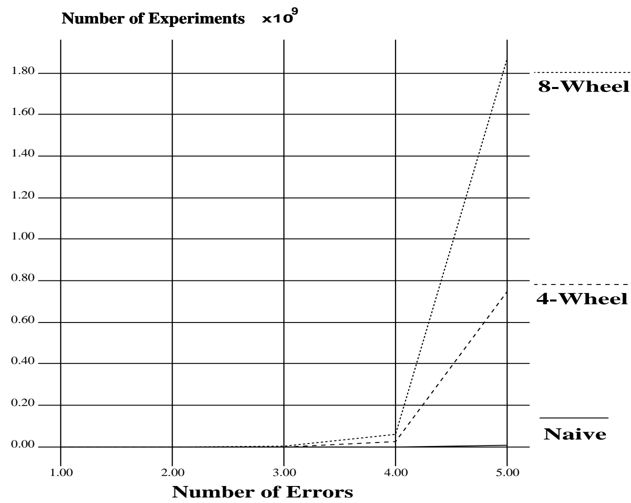

Figure 9: Naive, 4-Wheels and 8-Wheels Models

6.1 Least-Confidence-First Heuristic

WISER can use expert knowledge to efficiently find a probabilistically-best seed state. When $\hat{S}$ is not a positive state, WISER can use the Least-Confidence-First Heuristic to probabilistically order the literals according to the expert's confidence, experimenting first with the lowest-probability literals explained earlier. We introduce 3 possible kinds of expert knowledge: accurate, uniform and blur. To evaluate the performance of WISER using different expert knowledge, we measure the results averaged over 5 trials in the Flat Tire domain. The performance of WISER using accurate knowledge is shown in Table 1. The first column enumerates the number of errors. The next column shows the mean of 5 results, each result averaged over 5 trials. We compare the mean with the theoretical best and worst performance bounds.

Table 1: Using Accurate Knowledge

Errors	Mean	Best Case	Worst Case
1	5.4	1	66
2	308.6	67	2211
3	5831.4	2212	47971
4	64581.8	47972	768691
5	941668.0	768692	9705618

The performance of WISER using uniform knowledge is shown in Table 2. The mean of each experiment over 5 trials is provided, along with the theoretical average case result.

Table 2: Using Uniform Knowledge

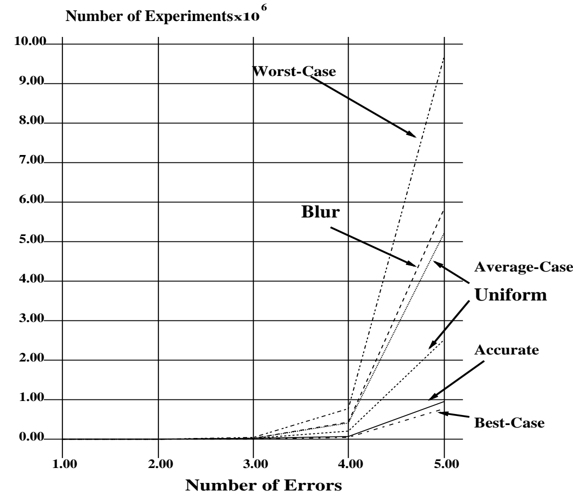

Figure 10: Three Expert Knowledge and Theoretical Bounds

Errors	Mean	Average Case
1	23.4	33.5
2	436.4	1139.0
3	10074.6	25091.5
4	187218.8	408331.5
5	2520362.0	5237155.5

As shown, the accurate knowledge closely matches the best case, and the blurred knowledge closely matches the average case in Figure 10. The performance of the uniform case is twice as good as the blurred case or the theoretical average case. As expected, the uniform case results fall between the best and average cases as shown in Figure 10.

6.2 Highest-Frequency-First Heuristic

When learning multiple operators simultaneously, we assume that a literal relevant to one operator is likely to be relevant to other similar operators. Given no background knowledge, WISER assigns higher weight to literals used more frequently in learning

previous operator definitions. WISER then searches the highest-weighted literals first. Empirical results from running WISER in a naive domain show that the number of experiments for learning preconditions for 13 operators is 56,700, averaged over 5 tests, for $\varepsilon = 3$ with no ordering heuristic. The number of experiments is reduced by half to 28,416 when using the *highest frequency heuristic*. The average number of experiments per operator with and without the highest frequency heuristic are shown and compared to the number for the Unlock-door operator in Table 3.

Table 3: Using Frequency Knowledge

Operator	No Knowledge	HFH
Averaged over 13 Ops	4631.6	2185.9

Our search tree, originally designed for learning a single operator, can be expanded to learn multiple operators by continuing until we learn the entire operator set. We experiment with all candidate operators at each node in the search space. The advantage of this new method is that we do not need to visit the same node more than once in learning multiple operators. Compared to the method of testing only one operator in a new state, this method is more efficient if the cost of changing a state is greater than the cost of executing operators, while our previous method is more useful if the cost of executing new operators is greater than the cost of changing a state. We can assume that the cost of changing a state is greater than the cost of changing operators in many cases. The empirical results of running 7 tests for learning 13 operators in the naive Extended Strips Domain show that the performance is improved on average by a factor of 1.133 using the frequency ordering heuristic, as shown in Table 4.

Table 4: Using Frequency Knowledge in Many Ops in One State

	No heuristic	Ordering heuristic
1	3856836	2557909
2	3277045	3762942
3	64418130	58160685
4	911893	1047349
5	42161474	4162115
6	1431294	2946720
7	35823095	23951608
Mean	2.169711e+7	1.9149768e+7

7 Refinement of Initial Operator

The preconditions of an operator should represent the most general description of all the states to which the operator is applicable. This is called the "weakest conditions" by Benson [1]. The initial definition of an operator learned from one positive state is often overly-specific (incorrect) as well as overly-general(incomplete). WISER applies three kinds of refinements. The empirical results demonstrating the effectiveness of these refinements will be shown in the next step.

7.1 Generalization by Literals Elimination

The preconditions for an operator are initialized as a positive state, S. The overly-specific definition should be refined by removing irrelevant predicates. This process requires many training examples in Wang's method. However, our insight is that one positive training example (or one observation from an expert) is sufficient for generalizing initial preconditions in a naive domain. We adopt Craven and Shavlik's bottom-up search generalization method [4]. While the preconditions are overly-specific, WISER chooses and negates each literal, $l \in S$, transitioning to a new state $S' = \{S - l\}$.

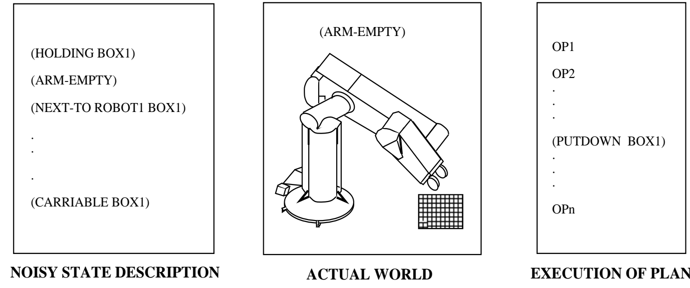

Figure 11: Noisy State and Plan Failure

WISER tests if the operator is still applicable in S'. If applicable, we delete the irrelevant literal and generalize S into $\{S - l\}$. WISER keeps on generalizing until S' contains only the necessary preconditions. The size of the search is λ in a naive domain. Note that the irrelevant literal is assigned "*" a priori in Oates and Cohen's work while we can detect "*" by experimentation.

7.2 Learning Negative Preconditions

Most current planners cannot handle unavoidable noisy data. A noisy state causes a plan failure in Figure 11. The robot's arm is empty in the actual world, but the robot believes that it is holding *box1* and tries to execute a plan step, (Putdown box1). To make a planner robust in the noisy state, we propose a method to add negative constraints to a domain theory.

Knowledge acquisition is the mapping of human knowledge to a machine. One type of implicit knowledge is the knowledge which the expert believes the machine possesses after the mapping, but the machine actually does not possess. Implicit negative knowledge is difficult to detect and make explicit. For example, given that *(arm-empty)* is known to both a human and a machine, $\neg$*(holding x)* is known to the human, but not known to the machine. However, negative knowledge is crucial in handling complex real world problems. Carbonell and Gil [3] show an incomplete domain theory in which

a hidden negative constraint causes a plan failure. The failure is the inevitable result of an expert's poorly-designed domain theory. We propose a novel method to learn the necessary negative information at the initial design stage of a domain theory by unifying two types of logic systems as follows: A state description uses two-value logic: true or false. As with many reasoning systems, we adopt the Closed-World Assumption (CWA): if p is not in the state, $\neg p$ is inferred. However, WISER operator descriptions use three-value logic: true, false, or irrelevant. If p is not in the preconditions, p is irrelevant. If p should not be in the state, $\neg p$ must appear in the preconditions. The knowledge acquisition of a learner is delimited by its language. The WISER description language is a typed first-order logic that allows conjunction and negation.

Let L represent all the predicates known to WISER and P the predicates that are true in S, a positive state. S is represented as the conjunction of predicates. This raw input data constitutes the potentially overly-specific preconditions for the operators in OBSERVER. We adopt three-value logic and transform the positive state to its closure to include the negative information before initializing it to overly-specific preconditions. F, the predicates not true in S, is $\{L - P\}$ by CWA. Removing CWA, S transits to $S^* = \{P \cup \neg F\}$. S^* is identical to S and provides a more comprehensive description of the same state. S* is used for inducing preconditions in WISER. WISER successfully avoids inconsistent actions in a noisy state by learning more constrained preconditions.

Suppose an inconsistent initial state is supplied by a noisy fast-moving camera. A planner may generate a plan, and the plan execution sometimes succeeds and sometimes fails due to perceptual alias in the state [1, 20]. To prevent a catastrophic result in the real world, WISER successfully generates more constrained preconditions of an operator by using negative constraints.

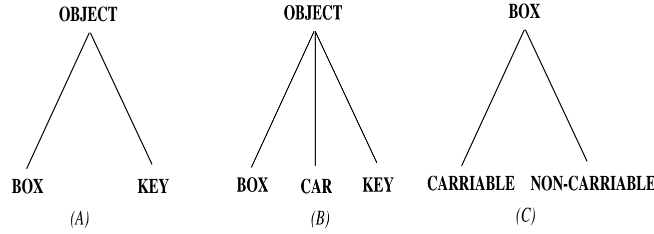

Figure 12: Three Different Type Hierarchies

7.3 Type Generalization

WISER learns the most specific type for each parameter of an operator. Thus, (Pickup Box) cannot be used to pick up a key. This degrades an agent's performance significantly. To generalize the specific type, WISER experiments with its sibling types in a type hierarchy provided for each parameter. If the operator definition works only for some of its sibling types, WISER generalizes the specific type to a more general type by using logical disjunction. WISER generalizes to its supertype only if the operator definition works for all sibling types. If the object type is composed of the box-type, the car-type and the key-type as in Figure 12-b, and (Pickup Key) works, but (Pickup Car) does not work, WISER introduces new disjunctive type: (or Key Box). If the object type is composed of the box-type and the key-type as in Figure 12-a, and (Pickup Key) works, (Pickup Object) is learned. The hierarchies can contain any number of generalization levels.

7.4 Type Specialization by Learning New Terms

A robot can expedite learning in simple environments first and then incrementally adjust an operator definition to complex environments by employing an inductive learning method. Let's start with an intuitive example of how a newborn baby learns to feed itself. Suppose the Feeding-World is composed of only two resources of an object-type, a bottle b and a finger f, and the baby is endowed with feeding capabilities. He first

starts with a naive world assumption where only one type composed of the two objects exists. He learns the preconditions {(in-mouth object) (hungry)} and the effects {(not-hungry)} after only one action is successfully applied. Soon, the baby may try *f*, its finger. The preconditions are satisfied, but the desired effect does not happen. Given that the baby is supplied with some reasoning ability, he may begin to distinguish between the objects of the bottle-type [b] and the objects of the non-bottle-type [f] after a number of examples are given. He learns the new concept for [bottle-type] and the new preconditions, {(in-mouth object)(bottle-type object)(hungry)}.

Wang assumes that all the predicates in the state are observable. Wang's approach results in the following potential problem. A robot has different physical capabilities than a human. A robot may carry an object up to 500 pounds in weight, while a human is limited to 50 pounds. Through observation, the robot will learn an overly-specific concept that it can carry the objects only up to 50 pounds. We will relax Wang's overly-simplistic assumption and divide the predicates into the observable (such as *height*) and the unobservable (such as *carriable*). For example, a robot cannot observe the weight of an object. Given a small number of objects, we can encode carriable(x) in our database by enumerating all carriable objects. However, if there are infinitely many objects in the domain, it is impossible to encode each object.

We introduce an autonomous method to learn these types of unobservable predicates incrementally. Each object is described as a vector of observable features, such as *(material shape color width length height)*. Each feature has the associated value. For example, the *shape* feature has a value from *(flat round angle)*. First, the robot learns overly-general operator definitions by assuming that every object is carriable in a naive domain. The robot experiences a plan failure due to the overly-general preconditions. Experimentation relies upon environmental feedback and produces a set of positive and negative (or false-positive) training examples for the preconditions of the faulty

operator, such as *(+ (wood flat red narrow short high))* and *(- (metal flat red wide long high))*.

Next we use C4.5, an information-theoretic system that induces a decision tree from pre-classified test cases, to classify a set of objects into homogeneous subclasses, say positive and negative examples, by selecting the features with the highest discriminating values in terms of the information gain [17]. As an example, when a set of boxes are described using features including the box material, the shape of the box, the color, the width, and the height, and these boxes are classified as carriable or not-carriable based on experience with this particular set of boxes, C4.5 learns a rule such as "if a box is made of wood, the bottom is flat, and it has width less than two feet, then the box is carriable".

The selected features are used for classifying unseen objects into respective subclasses. Our approach learns the functional concept composed of an unobservable predicate, such as *carriable*, by learning structural descriptions consisting of observable predicates as in Figure 12-c. This structural concept satisfies the operational criterion of Mitchell et. al [12] using only predicates which are easily observable by a robot. The benefit of learning an operational concept for an unobservable literal is that we don't need to rely on the information supplied by a human.

7.5 Empirical Results of Refinement Methods

To empirically demonstrate the improved accuracy after each stage of refinement of the initial operator definition, we first generate test problems for Box type objects with 70% probability distribution and 30% for Key type objects. We then inject noise into each problem with 10% probability[2]. We test 100 new problems in each trial.

[2]Since WISER cannot handle universal quantifiers to specify that if an agent holds a box it cannot hold any other box, we exclude this type of noise from testing. This type of reasoning would be a valuable

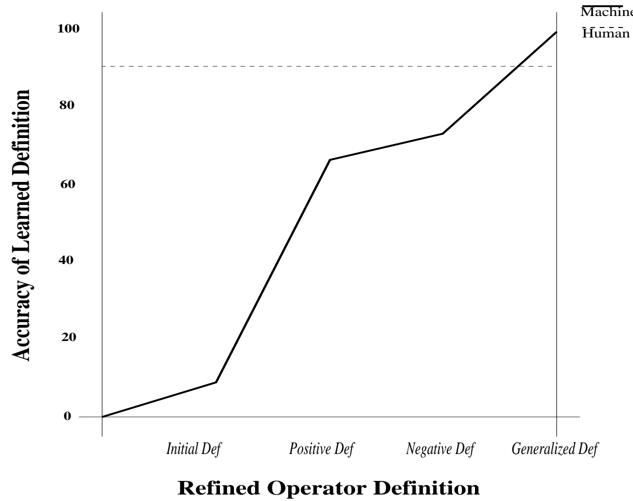

Figure 13: Accuracy of Refined Operator Definitions

The empirical results, shown in Figure 13, are averaged over 3 trials and demonstrate that an initially-learned operator is extremely overspecific and, after the elimination of irrelevant literals, the definition composed of only positive literals improves its performance by more than 50%. Type-generalization improves the performance by about 30% solving the problems for the new subtype, Key type. The human-supplied definition, obtained from a PRODIGY domain, does not have negative constraints. It cannot handle the problems of noisy states. The machine-generated generalized definition outperforms the expert-provided definition since it handles noisy states. Empirical results using C4.5 will be available in our next step.

8 Learning Effects of Operators

An operator describes what changes after the action, not what stays the same. When an operator is successfully executed and generates NS, the literals which belong to IS but not to NS constitute the operator's deleted list: $\Delta_D \leftarrow \{IS \setminus NS\}$. The literals

extension to our current work

in NS but not in IS constitute the operator's added list: $\Delta_A \leftarrow \{NS \setminus IS\}$. The new state is represented as $\{IS - \Delta_D + \Delta_A\}$. The delta state is composed of Δ_D and Δ_A: $\Delta \leftarrow \{\Delta_D + \Delta_A\}$.

Suppose an operator, (OP box roomx roomy), is applied in *IS* to move *box1* in *room1* to *room2* and generates *NS*:

IS:	**NS:**
(connects dr12 rm1 rm2)	(connects dr12 rm1 rm2)
(dr-to-rm dr12 rm1)	(dr-to-rm dr12 rm1))
(inroom robot rm1)	(inroom robot rm2)
(inroom box1 rm1)	(inroom box1 rm2)
(inroom box2 rm1)	(inroom box2 rm1)
(arm-empty)	(arm-empty)
(unlocked dr12)	(unlocked dr12)
(dr-open dr12)	(dr-open dr12)
(next-to robot box1)	(next-to robot box1)
(next-to box1 robot)	(next-to box1 robot)
(next-to dr12 box1)	
(next-to box1 dr12)	
(next-to box1 box2)	
(next-to box2 box1)	
(next-to robot dr12)	
(next-to dr12 robot)	
(next-to robot box2)	
(next-to box2 robot)	

The delta state Δ is obtained from IS and NS:

(DEL (next-to dr12 box1)

```
        (next-to box1 dr12)
        (next-to box1 box2)
        (next-to box2 box1)
        (next-to dr12 robot)
        (next-to robot dr12)
        (next-to robot box2)
        (next-to box2 robot)
        (inroom robot rm1)
        (inroom box1 rm1))
   (ADD (inroom box1 rm2)
        (inroom robot rm2))
```

Note that the classical frame problem is addressed by virtue of the fact that PRODIGY assumes all aspects of the state description remain the same unless explicitly noted otherwise in the operator effect list, and WISER generates an effect description using only differences between state descriptions.

In this section, we will first show that we cannot use the naive domain assumption to learn the effects of an action and secondly show that Wang's method of simply parameterizing each object in Δ to its own type is also problematic. Finally, we introduce a new learning method which uses an unbound variable for parameterizing objects.

8.1 Problems with Learning Effects Using the Naive Domain Assumption

Target objects are bound to an operator's parameters. Since the change in the status of the non-target objects may affect the next action of the robot, the effects of an

operator should consider the non-target objects as well as target objects. The problem of using the naive domain assumption is that it ignores the non-target objects. For example, if the robot is next to box1 in roomx in S1, and it moves to roomy in S2 by (Go-thru-dr roomx roomy), the robot is not next to box1 in S2. If the robot wants to pickup box1, it must know that it is not next to box1 any more. Suppose that the parameters are bound to the target objects as follows: {(box box1) (roomx room1) (roomy room2)}, and box2 is unbound. Since box2 is ignored by the naive method, the effects are learned from the delta state Δ as

```
(DEL  (next-to robot door)
      (next-to door robot)
      (next-to box door)
      (next-to door box)
      (inroom robot room)
      (inroom box room))
(ADD  (inroom box roomy)
      (inroom robot roomy))
```

Using this incomplete definition learned by adopting the naive domain assumption, NS will be *erroneously* predicted as

```
(connects dr12 rm1 rm2)     (dr-to-rm dr12 rm1)
(inroom robot rm2)          (inroom box1 rm2)
(inroom box2 rm1)           (next-to robot box1)
(next-to box1 robot)        (next-to robot box2)
(next-to box2 robot)        (next-to box1 box2)
(next-to box2 box1)         (unlocked dr12)
(arm-empty)                 (dr-open dr12)
```

Suppose the next goal of the robot is (next-to robot box2). This goal is already true, and no plan will be generated. On the other hand, Wang [22] learns the effects of an operator by parameterizing Δ. However, certain effects cannot be learned by simply parameterizing each object in Δ to its own type. Since all the objects in a delta state are parameterized to types, box2 is parameterized to Box. The effects are learned from the delta state Δ as

```
(DEL (next-to robot box)
     (next-to box robot)
     (next-to robot door)
     (next-to door robot)
     (next-to box door)
     (next-to door box)
     (next-to box box)
     (inroom robot room)
     (inroom box room))
(ADD (inroom box roomy)
     (inroom robot roomy))
```

This incorrect definition erroneously deletes (next-to robot box1) from NS:

(connects dr12 rm1 rm2)	(dr-to-rm dr12 rm1)
(inroom robot rm2)	(inroom box1 rm2)
(inroom box2 rm1)	(arm-empty)
(next-to robot box2)	(next-to box2 robot)
(next-to box2 box1)	(next-to box1 box2)
(unlocked dr12)	(dr-open dr12)

8.2 Introducing an Unbound Variable for Unbound Objects

One problem in Wang's approach lies in not distinguishing target objects from non-target objects. Oates and Cohen also exclude the problem of unbound objects in the effect list – they allow effects to contain a value for a sensor only if the preconditions also specify a value for that sensor [13]. On the other hand, to handle non-target objects, the STRIPS system uses the "wildcard" feature in the effect list [6, 7]. The PRODIGY system implements the "wildcard" variable by introducing unbound variables. Unbound variables are the variables not included in the parameter list. All the objects not bound to operator parameters in the delta state are bound to the unbound variables in PRODIGY. Note that target objects are not explicitly bound only to operator parameters in PRODIGY. As a result, some of the target objects are bound to unbound variables in HR.

The human representation HR, which can be assumed to be (close to) *IK* in a simple model, may seem efficient, but HR is often inconsistent [19]. When the unbound variables are not consistently used in HR, it is sometimes complicated to determine which objects are in the scope of an unbound variable, and it is hard to understand the function of an operator. We claim that to solve this problem, only unbound objects must be bound to the unbound variables and each target object in the delta-state must be explicitly bound to a parameter in its own type using a type hierarchy as shown in Figure 12. The effects of an operator are learned by parameterizing Δ according to this new mapping scheme in WISER. The machine representation MR of the effects of the Pickup-obj operator learned by WISER is compared with the human representation HR described in [2] and it shows some advantages over HR: WISER uses only one unbound variable systematically.

```
operator PICKUP-OBJ (params <object>)
```

```
    (<object> (OBJECT))

Human Representation

    (<other> (or OBJECT DOOR))

    (<other2> (or OBJECT DOOR))

    (del (arm-empty))

    (del (next-to <object> <other>))

    (del (next-to <other2> <object>))

    (add (holding <object>))

Machine Representation

    (<other> (and (or OBJECT DOOR) (diff <other> <object>)))

    (del (arm-empty))

    (del (next-to <other> <object>))

    (del (next-to <object> <other>))

    (add (holding <object>))
```

HR has two unbound variables, ⟨other⟩ and ⟨other2⟩. Rather surprisingly, they subsume all the objects of either OBJECT or DOOR type including the *target* object which is supposed to be bound to ⟨object⟩ at run time. However, the unbound variable ⟨other⟩ in MR is bound to only the *unbound* objects by specifying *explicitly* that ⟨object⟩ is different from ⟨other⟩. Note that ⟨other⟩ is the same object for both predicates to satisfy symmetry as will be explained below. The machine-generated representation is more constrained than the human-generated one. Next, we compare the HR and MR representations for the Push-thru-dr operator.

```
operator PUSH-THRU-DR (params <box> <door> <room> <roomy>)

   Human  Representation

       ((<other>  (and (or OBJECT DOOR) (diff <other> <box>)))

        (<other2> (or OBJECT DOOR))
```

```
        (<other3> (and (or OBJECT DOOR) (diff <other3> robot)))
        (del (next-to robot <other>))
        (del (next-to <box> <other2>))
        (del (next-to <other3> <box>))
        ...
Machine  Representation
       ((<other> (and (or OBJECT DOOR)
               (diff <other> <box>) (diff <other> <door>)))
        (del (next-to <door> <box>))
        (del (next-to <box> <door>))
        (del (next-to <other> <box>))
        (del (next-to <box> <other>))
        (del (next-to <door> robot))
        (del (next-to robot <door>))
        (del (next-to <other> robot))
        (del (next-to robot <other>))
        ...
```

The *next-to* relation is symmetrical by its definition. This symmetry relation is obvious to a human, but it is not that obvious to a machine. Note that (next-to ⟨box⟩ ⟨other2⟩) and (next-to ⟨other3⟩ ⟨box⟩) are symmetrical, but (next-to robot ⟨other⟩) is not in HR. This inconsistency may cause problems in machine reasoning. In order to get around this problem, HR adopts a convention that the robot appears only as the first argument and does not use literals such as (next-to ⟨box⟩ robot) [10]. We enforce this convention by imposing the symmetry on all observable relevant literals.

HR with three unbound variables is somewhat confusing. If the three unbound variables can be unified into one unbound variable, the resulting form will be simpler

and more mechanical. We will show how HR can be transformed to MR by assigning the most specific constraints to unbound variables.

The expression (⟨other3⟩ (and (or OBJECT DOOR) (diff ⟨other3⟩ robot))) specifies that ⟨other3⟩ should be either an OBJECT or a DOOR type and cannot be instantiated to a robot. However, a variable of either OBJECT or DOOR type cannot be instantiated to a robot by a type hierarchy definition shown in Figure 12. Thus, the expression (diff ⟨other3⟩ robot), is always true and can be removed. ⟨other3⟩ is unified to ⟨other2⟩ and (del (next-to ⟨other3⟩ ⟨box⟩)) is transformed to (del (next-to ⟨other2⟩ ⟨box⟩)). We can even remove ⟨other2⟩ as follows.

Idealistically, the scope of unbound variables should not overlap with those of the parameters. Given the parameters ⟨box⟩ and ⟨door⟩, the most specific constraints to ⟨other⟩ are ((diff ⟨other⟩ ⟨box⟩) (diff ⟨other⟩ ⟨door⟩)). When both of the unbound variables are assigned with the most specific constraints, they can be unified resulting in only one variable. Now, the literals in Δ related with ⟨box⟩ or ⟨door⟩ should be explicitly coded, since they can no longer be represented by unbound variables. Note that (diff ⟨other⟩⟨box⟩) forces ⟨other⟩ to subsume {dr12 box2 box3 box4 key1 key2} in HR, and that (diff ⟨other⟩⟨box⟩) (diff ⟨other⟩⟨door⟩) forces ⟨other⟩ to subsume {box2 box3 box4 key1 key2} in MR. The following literals appear in Δ_D[3]:

(next-to dr12 robot)* (next-to robot dr12)

(next-to box2 robot)* (next-to robot box2)

are parameterized in HR to (del (next-to robot ⟨other⟩)), and parameterized in MR to

```
(del (next-to <door> robot))
(del (next-to robot <door>))
(del (next-to <other> robot))
```

[3] The literals with "*" do not actually appear in HR as explained.

```
(del (next-to robot <other>)).
```

Thus, the expressions,

```
(next-to <door> robot) and (next-to robot <door>)
```

are implicit in HR and become explicit in MR. They reveal the hidden relation between the door and the robot. Implicit knowledge, which the expert believes the machine possesses but the machine actually does not possess, can cause problems [5, 19].

Many operating learning systems handle only effects on target objects, which are bound by parameters of the operator. Reasoning about non-target objects after the action is important because the non-target objects may be relevant in choosing the next action. In this section we have demonstrated that this is possible if we use an unbound variable to parameterize the non-target objects. In some cases, the machine-generated definition may actually be more succinct than the expert-provided definition.

9 Conclusions and Future Research

As planning systems begin to solve more realistic problems, automated knowledge acquisition techniques become increasingly important. We have designed and implemented WISER, a prototype experimentation-driven operator learning system running on top of the PRODIGY system. WISER learns an initial operator definition from one positive training example. To reduce the number of experiments, we introduce a naive domain assumption and literal-ordering heuristics. We present analytical models and empirical results demonstrating how much experimentation is necessary to generate an initial operator definition as a function of the number of errors. We show the performance improvement obtained by using the naive method and literal orderings, making use of either background knowledge or the learner's own past experience.

This definition is refined by removing irrelevant literals. WISER induces negative literals by transforming the two-valued logic state description into three-valued logic. The learned negative literals prevent inconsistent planning problems in noisy states. We generalize a type into its supertype for parameters of the learned operator. Empirical results demonstrate WISER's ability to generate more accurate operator definitions by performing more experiments. The naive domain assumption that we adopt causes overly-general operator definitions in a realistic domain. To cope with this problem, we use C4.5 to learn subtypes that can represent an unobservable concept with observable features. To learn the effects of an operator, WISER computes the delta state and parameterizes the delta state by handling the unbound variable unlike other systems.

One current limitation of WISER is that the system relies on oracles to generate state descriptions. In real applications, it is not always possible to move to any desired state in order to test an operator. As a result, future work will focus on generating only state descriptions that can be reached with known operators. In addition, our ongoing research includes methods for refining missing preconditions and effects for operators not just applied in isolation, but also embedded inside a large plan.

References

[1] S. Benson. Inductive learning of reactive action models. In *Proceedings of the 12th International Conference on Machine Learning*, 1995.

[2] J. G. Carbonell, J. Blythe, O. Etzioni, Y. Gil, C. Knoblock, S. Minton, A. Perez, and X. Wang. Prodigy4.0: The manual and tutorial. Technical Report CMU-CS-92-150, Carnegie Mellon University, Pittsburgh, PA, 1992.

[3] J. G. Carbonell and Y. Gil. Learning by experimentation: The operator refinement method. In *Machine Learning, An Artificial Intelligence Approach*, volume 3.

Morgan Kaufmann, San Mateo, CA, 1990.

[4] M. W. Craven and J. W. Shavlik. Using sampling and queries to extract rules from trained neural networks. In *Proceedings of the Eleventh International Conference on Machine Learning*, 1994.

[5] M. desJardins. Knowledge development methods for planning systems. In *AAAI-94 Fall Symposium Series: Planning and Learning: On to Real Applications*, 1994.

[6] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalizized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[7] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1972.

[8] Y. Gil. *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1992.

[9] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic least-commitment planning. In *Proceedings of the National Conference on Artificial Intelligence*, 1994.

[10] V. Lifschitz. On the semantics of strips. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, chapter 7, pages 523–531. Morgan Kaufmann, 1990.

[11] J. J. Mahoney and R. J. Mooney. Combining connectionist and symbolic learning to refine certainty-factor rule-bases. *Connection Science*, 5:339–364, 1993.

[12] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986.

[13] T. Oates and P. R. Cohen. Searching for planning operators with context-dependent and probabilistic effects. In *Proceedings of the 13th National Conference on Artificial Intelligence*, 1996.

[14] T. Oates and P. R. Cohen. Searching for structure in multiple streams of data. In *Proceedings of the 13th International Conference on Machine Learning*, 1996.

[15] D. Ourston and R. J. Mooney. Changing the rules: A comprehensive approach to theory refinement. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, MA, 1990.

[16] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[17] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[18] S. J. Russell. Preliminary steps toward the automation of induction. In *Proceedings of the National Conference on Artificial Intelligence*, PA, 1986.

[19] K. S. Tae and D. J. Cook. Learning rules from incomplete and redundant domain theory. *Journal of Computing in Small Colleges*, 10(5):242–250, 1995.

[20] K. S. Tae and D. J. Cook. Experimental knowledge acquisition for planning. In *Proceedings of the 13th International Conference on Machine Learning*, 1996.

[21] K. S. Tae and D. J. Cook. Knowledge acquisition for planning with incomplete domain theorie s. In *AAAI 1996 Spring Symposium on Planning with Incomplete Information for Robot Problems*, 1996.

[22] X. Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the 12th International Conference on Machine Learning*, 1995.

[23] S. D. Whitehead and D. H. Ballard. Active perception and reinforcement learning. In *Proceedings of the 7th International Conference on Machine Learning*, 1990.

[24] C. C. Yang. *Relational Databases.* Prentice Hall, Englewood Cliffs, NJ, 1986.