A Framework for the Analysis of Sophisticated Control in Interpretation Systems *

Robert C. Whitehair Victor R. Lesser

Technical Report Number 93–53

Abstract

This paper introduces a framework for the analysis of sophisticated search control architectures in AI systems. The framework is based on two formalisms, the *Interpretation Decision Problem (IDP)*, which models the characteristics and problem structure of a domain, and the *UPC* formalism, which provides a general model of control and problem solving. Using these models, the problem structures of disparate domains and the problem solving architectures constructed to exploit these structures can be viewed from a unified perspective where control and problem solving actions can be considered a single class of problem solving activity. Models built from this unified perspective offer advantages for describing, predicting and explaining the behavior of *interpretation* systems and for generalizing a specific problem solving architecture to other domains. Use of the IDP and *UPC* formalisms also supports the synthesis of new, more flexible problem solving architectures. Examples based on formalizing uncertainty and subproblem interaction are used to illustrate the power of the IDP/*UPC* framework.

*This work was supported by the Office of Naval Research contract N00014-92-J-1450. The content does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Contents

List of Tables

List of Figures

1 Introduction

Many research projects have shown that "sophisticated[1]" control and problem solving techniques, such as "meta-level processing," can be applied in complex problem solving domains to improve system performance [7, 16, 17, 21, 37, 39]. However, for many of these techniques, there are open issues regarding why the technique works, how to use a particular technique properly in its original domain, and how to generalize the technique to other domains. These open issues result from a lack of understanding associated with both the properties of a domain that determine the effectiveness of a particular problem solving technique and the assumptions (implicit and explicit) about the properties of a domain that are incorporated into a problem solving technique.

One class of problems where meta-level processing has been shown to be advantageous is that of *interpretation problems.* Intuitively, interpretation problems are tasks where a stream of input data is analyzed and an explanation is postulated as to what domain events occurred to generate the signal data – the problem solver is attempting to interpret the signal data and determine what caused it [4, 12].

Given the success of some "sophisticated" interpretation problem solving techniques, there has been an obvious desire to understand the underlying principles and domain properties in order to generalize the techniques to other domains. Such techniques include the focus of control mechanisms introduced in the Hearsay-II speech understanding system [12, 20] and the Distributed Vehicle Monitoring Testbed (DVMT) [2, 6, 8], sophisticated control techniques such as goal processing [4, 5, 26, 27], and abstracting and approximating computational domain theories [9, 28].

This paper presents a formal framework for investigating and analyzing the relationship between the performance of search-based interpretation problem solving systems and the inherent properties, or *structure*, of problem domains in which they are applied. The analysis framework models the structure of interpretation *domain theories* (a domain theory is the computational theory that is the basis for a problem solver's functionality) in terms of four feature structures: *component* (or *syntax*), *utility* (or *credibility*)[2], *probability* (or *distribution*), and *cost*.

The different feature structures that are defined by domain theories are combined into a unified representation by expressing them in terms of formal grammars and functions associated with production rules of the grammar. The formal grammar and functions associated with a domain theory will be referred to as the *domain grammar.* This unified approach allows search paths to be represented graphically as parse trees (or *interpretation trees*) of the grammar. By analyzing the statistical properties of the interpretation trees of a domain grammar, it will be possible to determine general characteristics of problem solving in the domain such as the expected cost for a random problem instance, the expected credibility of a solution, etc.

The statistical analysis is based on an approach where interpretation problems are viewed as discrete optimization problems, implying that the problem solver must consider, either implicitly or explicitly, every possible interpretation for a set of signal data and identify the best interpretation. This also implies that the problem solver must *connect* every path in the search space to either a dead end state or a final state representing a possible interpretation. (Connected paths are defined formally in Section 4.) It is important to note that the sophisticated control techniques that are the focus of this paper can

[1] *Sophisticated problem solving* will be formally defined in subsequent sections.

[2] In Section 4, credibility structures are formally linked to the semantics associated with full and partial interpretations. Thus, a full or partial interpretation that has a high credibility can intuitively be thought of as having a highly consistent semantic interpretation and a full or partial interpretation that has a low credibility can intuitively be thought of as having an inconsistent or incomplete semantic interpretation.

implicitly enumerate search spaces efficiently and that connecting a space does not necessarily require every potential final state to be generated. Viewing interpretation problems as discrete optimization problems sets the proposed analysis framework apart from previous analysis techniques that are used to analyze problem solving in domains where the objective is to find the shortest, or lowest-cost search path, the highest-rated solution, or a solution path to a "winning position" [34].

The analysis framework formalizes the representation of problem structures and control architectures in such a way that the following issues can be addressed;

1. Is it possible to formally define "sophisticated control architectures" and "complex problem solving domains?"
2. What characteristics of a complex problem solving domain can be determined analytically?
3. How is this analysis done?
4. Can the analysis include consideration of actions that dynamically prune search paths?
5. Can the analysis techniques be extended to incorporate base-level and meta-level processing in a unified framework?
6. Can the analysis techniques include considerations associated with subproblem interactions?
7. How is this done?
8. Can the analysis techniques include considerations associated with the long-term affects of a problem solving action?
9. How is this done?

The analysis framework is composed of the *Interpretation Decision Problem (IDP)* and the *UPC* formalism (from *U*tility, *P*robability, and *C*ost) and will be referred to as the IDP/*UPC* framework. Each of these formalisms can be used to derive a distinct set of calculations that can be used to analyze different aspects of a problem solver's performance. The calculations we derive for the IDP/*UPC* framework are based on the dynamic local perspective of problem solving for specific states in a search space and the statistical properties of a domain derived from a formal IDP specification. The remainder of this section presents a general introduction to the major components of the framework and it includes examples that will be explained in full over the course of the paper. This section will not address the formal mathematics on which the framework is based. In the sections that follow, the components of the framework will be presented in a manner that will address the formal mathematical and theoretical elements in more detail.

Figure 1 shows a general overview of the analysis framework. As shown in the figure, the natural structure of an interpretation problem is mapped into a formal representation, or domain theory, based on the IDP formalism. The IDP formalism represents problem structures in terms of a characteristic grammar and functions associated with the production rules of the grammar. By representing a domain structure in terms of the IDP formalism, certain statistical information about the structure of the corresponding search space can be derived. In particular, significant relationships among subproblems and among subproblems and final solutions can be determined and, to an extent, quantified. For example, given an IDP specification of an interpretation domain, general characteristics associated with

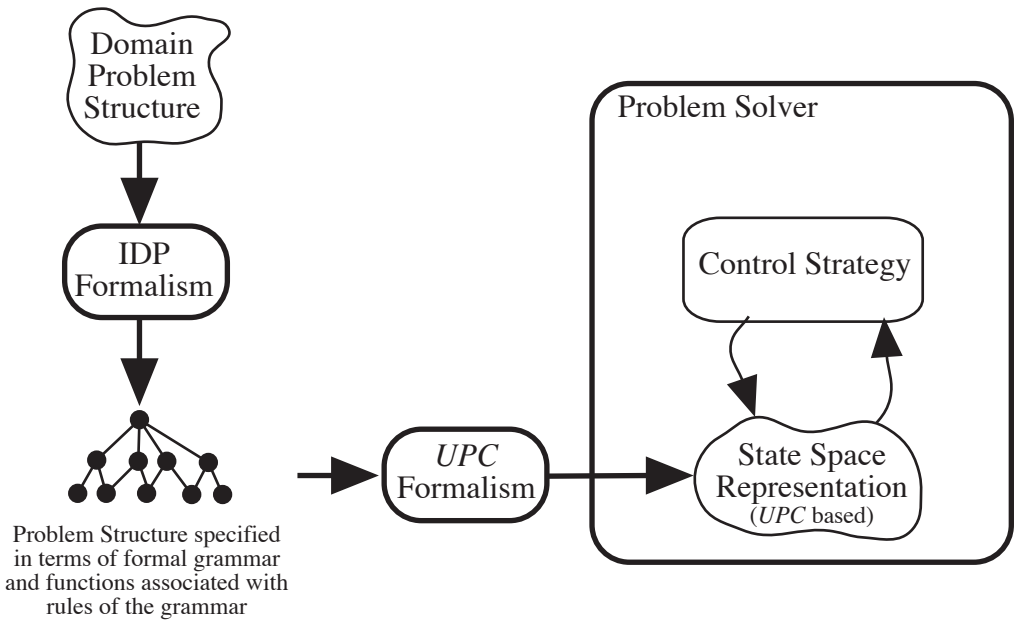


Figure 1: Overview of the IDP/*UPC* Framework for Analyzing Sophisticated Control

the complexity of problem instances from that domain can be determined. This includes characteristics such as the expected cost of problem solving for a random problem instance. To complement this, the *UPC* formalism explicitly represents subproblem relationships in a quantified way that can be used, either directly or in abstract form, by a problem solver's control component to schedule the execution of problem solving actions. (In a real system, the cost of accurately quantifying relationships may be prohibitively expensive, in which case it is necessary to use approximations or abstractions.) Thus, the *UPC* formalism can be used to determine which actions available to a problem solver are "optimal" from a local problem solving perspective and to explain why a problem solver chose a certain course of action in a given situation.

The *UPC* formalism can be used as a basis for understanding the control decisions made by a problem solver and for explaining and predicting the effects of a control algorithm on a problem solver's performance. The *UPC* formalism maps IDP structures into a state space representation where, for each intermediate state in the search space, certain relationships between the state and the final states that can be reached from the state are represented explicitly. This is shown in Fig. 2 and in Fig. 3. For a given intermediate search state, s, that corresponds to a partial solution, and for each of the final states that can be reached along paths from s, the relationships that are represented include the expected cost of reaching each final state from s, the expected utility of each of the final states, and the expected probability of successfully reaching each of the final states. These expected values will be referred to as the *UPC values* for a state. *UPC* values are determined dynamically based on a perspective of problem solving that is local to a given state and on the statistical properties of a domain derived from a formal IDP specification. The calculation of *UPC* values for a specific state does not take into consideration the existence or absence of any other state. Both Fig. 2 and Fig. 3 illustrate

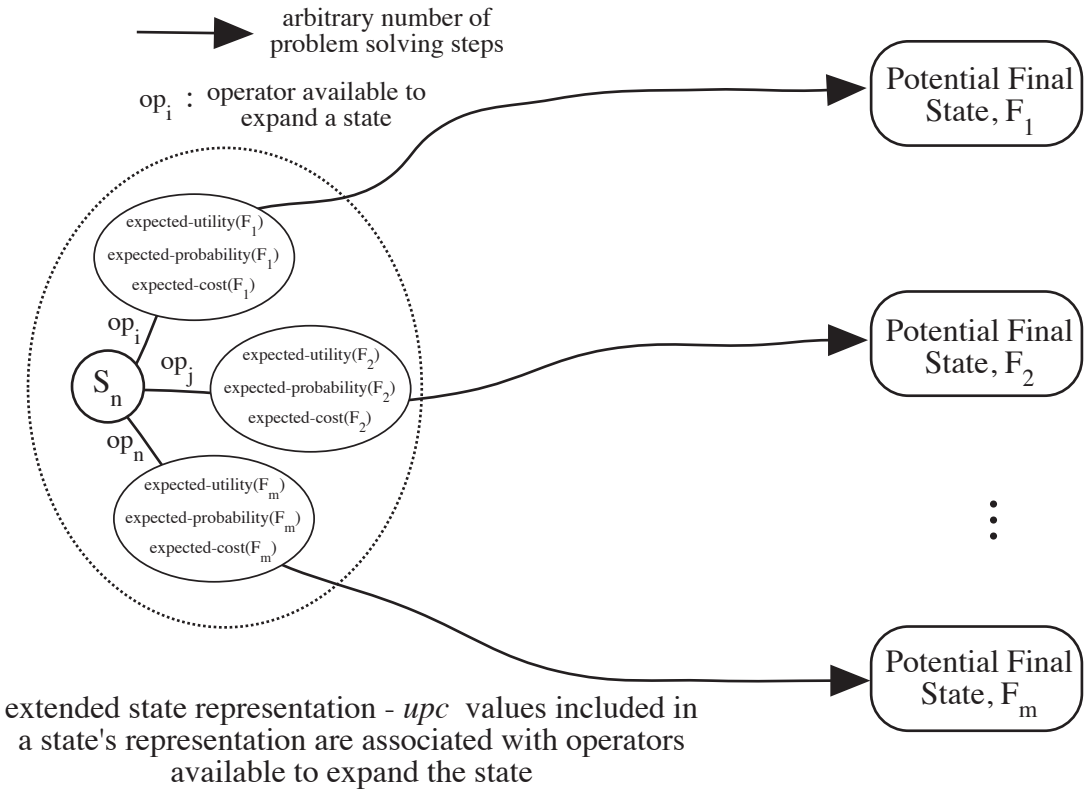


Figure 2: Derivation of *UPC* Values for a State

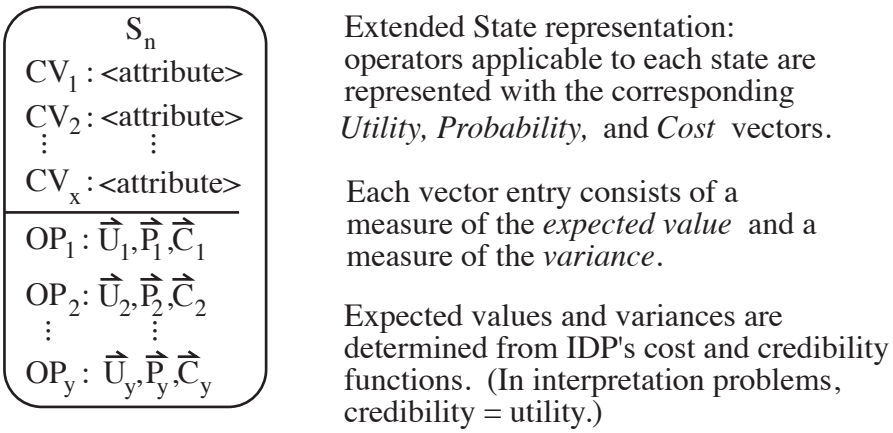


Figure 3: Representation of a Search State in the *UPC* Formalism

how the representation of a state is extended by associating vectors of *UPC* values with the operators available to extend paths from the state.

The analysis techniques incorporated in the IDP/*UPC* framework are motivated by, and based on, the fundamental principles that every AI problem domain has a corresponding *ideal domain theory* that can be used by a search-based problem solver's control component to make optimal decisions, and that a given domain theory is structured in a way that can be represented and exploited during problem solving. Decisions that are based on an ideal domain theory are optimal in the sense that, over time, they are statistically superior to all alternatives. For the comparative studies that will be performed with this analysis framework, an optimal decision making process can be used as a reference point. For example, in order to evaluate a given problem solving strategy, which will be referred to as a *control architecture*, it is necessary to compare it to something, either directly to a baseline optimal case or to another control architecture. In the latter case, the baseline optimal performance is used as a reference to help determine if performance differences in the two control architectures are significant. It is important to note that the term "ideal" in "ideal domain theory" is not meant to imply that there is no uncertainty associated with such a theory. In fact, quite the opposite is true. Though a problem solver can use meta-processing to limit the costs associated with the uncertainty in an ideal domain theory (as will be discussed), there is nothing a problem solver can do to eliminate these costs. This form of uncertainty will be referred to as *inherent uncertainty.*

Ideal domain theories are usually unknown and, when they are known, can require computations that are too expensive to use effectively in problem solving. As a consequence, control architectures are often based on *approximations* of an ideal domain theory. In general, the use of an approximate domain theory results in problem solving performance that is, in some sense, suboptimal. Formal definitions of optimal processing are given in Section 8. Two of the more prevalent effects of suboptimal processing in interpretation problem solvers are an increase in the processing time required and a decrease in the "quality" (i.e., correctness, completeness, precision, etc.) of interpretations. The proposed IDP/*UPC* framework is a tool for analyzing the performance implications associated with the use of approximations to the ideal domain theory.

The framework involves the use of two distinct grammars that reflect the concept of an ideal domain theory and an approximation of the domain theory used in problem solving. These grammars consist of a *generational grammar, IDP*$_G$, and an *interpretation grammar, IDP*$_I$. IDP_G corresponds to the ideal

domain theory in the sense that it can be used to generate problem instances that correspond to the actual events that occur in a domain. IDP_I is a representation of the problem solving actions available to a problem solver, including abstract and approximate operators used by the control mechanism.

In order to minimize the negative effects associated with the approximations used by a problem solver, it is necessary to develop a better understanding of the relationship between the problem solver's control architecture (represented by IDP_I), the problem structure (represented by IDP_G), and the problem solver's performance. This is because a given approximation technique is likely to work well only in certain situations and it is necessary to have some way of formally describing these situations. To accomplish this, the IDP/*UPC* framework will serve as a formalism for specifying the relevant characteristics of a search space so that the approximations and assumptions used by a control architecture can be explicitly represented and so that the performance of different control architectures can be compared and analyzed in terms of the properties of the domains in which they are applied. In particular, in terms of problem structures resulting from interacting subproblems. (Interacting subproblems are formally defined in Section 4. For now, two or more processing tasks are interacting subproblems if the results they generate can be used during the problem solving activities of a subsequent task.) Used in this manner, the framework supports the identification of general classes of problem domains and general classes of control architectures, and the description of the expected performance of classes of control architectures when applied to specific classes of problem domains. Ultimately, the objective is to use this information to construct design theories for search control architectures.

The IDP/*UPC* framework assumes that the phenomena, or individual problem instances, associated with a specific problem domain can be defined to occur in principled, or structured ways. Phenomena such as noise and missing data that make interpretation a difficult task do not "just occur," rather there are laws and principles that govern their occurrence and a problem solver can exploit these laws and principles in order to improve its performance. For example, a problem solver might be able to exploit its knowledge of certain recurring subproblems in a way that significantly reduces the overall cost of problem solving or that increases the quality of the solutions it generates. A domain's problem structure is a description of the causes of phenomena in the domain and knowing this structure is critical. For example, it is not enough to know that an interpretation domain experiences "noise" phenomena in order for a problem solver operating in that domain to successfully use a control architecture that was built to deal with noise in another, different domain. It is necessary to know if the characteristics of noise in the two domains are the same or similar enough that the control architecture can be extended to the new domain.

In IDP models, the different feature structures that are defined by domain theories are combined into a unified representation by expressing them in terms of formal grammars and functions associated with production rules of the grammar. Nonterminals of the grammar represent intermediate problem solving states, terminal symbols represent raw sensor input, and the production rules of the grammar represent potential problem solving actions. The grammar rules of IDP models specify the component structure of a domain and each production, p, has associated cost and utility functions, g_p and f_p, that define the cost and utility structures. In addition, IDP models explicitly represent aspects of inherent uncertainty in a domain with the distribution function, ψ, that defines the probability structure of the domain. (i.e., ψ, along with other mechanisms, define inherent uncertainty in a domain.) For a given production, p, the frequency of the occurrence of p's right-hand-side (RHS) is specified by the distribution function $\psi(p)$. Thus, p can have multiple RHSs, RHS_1 through RHS_n, and the distribution of the RHSs is defined by $\psi(p)$. Finally, each production rule, p, is associated with a semantic function, Γ_p that is a function of the subtree components represented by the elements on the

Interpretation Grammar G'

grammar rule	distribution	credibility (utility)	cost
0.1 S → A	$\psi(0.1) = 0.2$	$f_S(f_A)$	$g_S(g_A)$
0.2 S → B	$\psi(0.2) = 0.2$	$f_S(f_B)$	$g_S(g_B)$
0.3 S → M	$\psi(0.3) = 0.2$	$f_S(f_M)$	$g_S(g_M)$
0.4 S → N	$\psi(0.4) = 0.2$	$f_S(f_N)$	$g_S(g_N)$
0.5 S → O	$\psi(0.5) = 0.2$	$f_S(f_O)$	$g_S(g_O)$
1. A → CD	$\psi(1) = 1$	$f_A(f_C,f_D,\Gamma_1(C,D))$	$g_A(g_C,g_D,C(\Gamma_1(C,D)))$
2. B → DEW	$\psi(2) = 1$	$f_B(f_D,f_E,f_W,\Gamma_2(D,E,W))$	$g_B(g_D,g_E,g_W,C(\Gamma_2(D,E,W)))$
3.0 C → fg	$\psi(3.0) = 0.5$	$f_C(f_f,f_g,\Gamma_{3.0}(f,g))$	$g_C(g_f,g_g,C(\Gamma_{3.0}(f,g)))$
3.1. C → fgq	$\psi(3.1) = 0.5$	$f_C(f_f,f_g,f_q,\Gamma_{3.1}(f,g,q))$	$g_C(g_f,g_g,g_q,C(\Gamma_{3.1}(f,g,q)))$
4. E → jk	$\psi(4) = 1$	$f_E(f_j,f_k,\Gamma_4(j,k))$	$g_E(g_j,g_k,C(\Gamma_4(j,k)))$
5.0 D → hi	$\psi(5.0) = 0.5$	$f_D(f_h,f_i,\Gamma_{5.0}(h,i))$	$g_D(g_h,g_i,C(\Gamma_{5.0}(h,i)))$
5.1. D → rhi	$\psi(5.1) = 0.5$	$f_D(f_r,f_h,f_i,\Gamma_{5.1}(r,h,i))$	$g_D(g_r,g_h,g_i,C(\Gamma_{5.1}(r,h,i)))$
6.0 W → xyz	$\psi(6.0) = 0.5$	$f_W(f_x,f_y,f_z,\Gamma_{6.0}(x,y,z))$	$g_W(g_x,g_y,g_z,C(\Gamma_{6.0}(x,y,z)))$
6.1. W → xy	$\psi(6.1) = 0.5$	$f_W(f_x,f_y,\Gamma_{6.1}(x,y))$	$g_W(g_x,g_y,C(\Gamma_{6.1}(x,y)))$
7. f → (s)	$\psi(7) = 1$	$f_f(f_{(s)},\Gamma_7(\,(s)\,))$	$g_f(g_{(s)},C(\Gamma_7(\,(s)\,)))$
8. j → (s)	$\psi(8) = 1$	$f_j(f_{(s)},\Gamma_8(\,(s)\,))$	$g_j(g_{(s)},C(\Gamma_8(\,(s)\,)))$
●	●	●	●
●	●	●	●
●	●	●	●

(s) = signal data $C(\Gamma_n(i,j,...))$ = cost of executing $\Gamma_n(i,j,...)$

Figure 4: Example of an IDP Grammar with Associated Functions

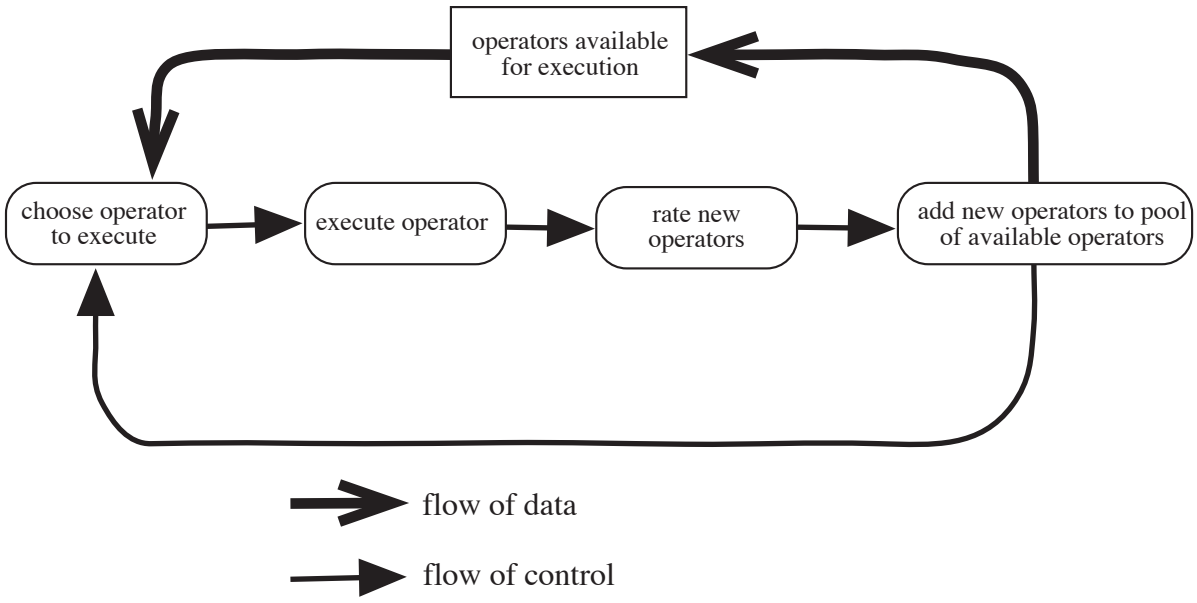


Figure 5: The Basic Control Cycle

right-hand-side of p. Γ_p measures the "consistency" of the semantics of its input data and returns a value that is included in the credibility function. For example, in a speech understanding domain, Γ_p would rate the consistency of the meaning of a sentence and return a value indicating whether or not the sentence made any sense.

Figure 4 is an example of an IDP grammar for a simple interpretation problem. Included in the figure are the functions associated with the grammar rules that define the structure of the domain. The "interpretations" associated with an A, B, M, N, or O are considered final solutions in this example.

The techniques that are used to analyze control architectures and, more generally, the performance of problem solving systems are based on formal IDP descriptions of the systems and on the resulting *UPC* values included in the extended representation of search states. The *UPC* representation was illustrated in Fig. 2 and in Fig. 3. As is discussed in Section 9, the utility, probability, and cost vectors associated with search operators can be derived from the corresponding IDP model. In essence, the IDP formalism explicitly represents the phenomena, such as functions defining distributions of domain events, sensor distortion, environmental noise, etc., that cause interpretation to be a complex task. These phenomena result in IDP representations that are *structured*. IDP structures are subsequently mapped to structures that appear in a search space and that can be exploited by the control component of a search-based problem solver. The *UPC* formalism explicitly represents these structures in a way that supports the analysis of a problem solving system's performance. (The use of the word "structure" will refer both to phenomena in IDP models and to corresponding phenomena in search spaces.)

A critical element of the IDP/*UPC* framework is that it provides a perspective of problem solving where meta-level processing, such as meta-level abstractions and approximations, and domain processing can be viewed and analyzed from a unified perspective. The basis for this perspective involves the distinction made between *local, independent control architectures* (or simply *local control*) and *non-local, dependent control architectures* (or *sophisticated control*). In the IDP/*UPC* framework, control is defined as the "evaluate/expand" cycle shown in Fig. 5. At each step of problem solving, the problem solver's control component chooses the highest rated operator and executes it. The execution of the operator

will usually cause other operators to be eligible for execution. For example, if a new state is created, the operators that can be applied to that state become eligible for execution. The control component's evaluation function rates the new operators and includes them in its deliberations in the next cycle.

As is described in Section 3, when evaluating a search operator, local control architectures only consider the characteristics of the operator and the state(s) it modifies and not more global information such as the characteristics of other, possibly interacting, search operators. In contrast, sophisticated control architectures evaluate problem solving actions by selectively applying a process that examines a search path's relationship with other, possibly interacting, search paths [27]. (Relationships will typically be based on the distribution of domain events and will be statistical in nature. For example, a relationship might indicate that two search paths lead to the same final result 50% of the time.) This process must be applied selectively in order to prevent a combinatorial increase in cost that would result from examining the relationships between every possible set of interacting search paths.

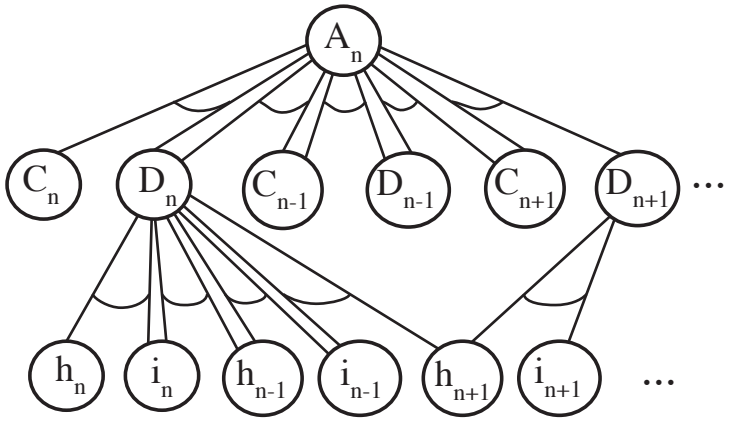
The analytical capabilities of the IDP/*UPC* framework are based on viewing the process of examining a search path's relationships as a *distinct search operation* and not as part of the control architecture. Interrelationships between search paths are represented as abstract or approximate states that are explicitly created by search operators and not by a monolithic control process. Furthermore, the process of examining the relationships between search paths is associated with an IDP domain theory representation consisting of an appropriate set of grammar rules and functions. Such problem solving actions have previously been referred to as *control problem solving*, *meta-knowledge operations*, *meta-operations*, *approximate problem solving*, *abstract problem solving*, etc. In general, these problem solving actions will be referred to as *meta-operators*. They will also be referred to as *abstract* or *approximate operators* because they are primarily derived from abstractions of other operators. Thus, abstractions and approximations used in problem solving to explicitly examine relationships between search paths are represented as extensions of a basic IDP model and both primitive operators and meta-operators correspond to production rules of the associated IDP grammar representation. Section 4 defines some of the general IDP structures associated with the abstract problem solving actions that are studied.

An example of the modifications that can be used to extend an interpretation grammar to represent meta-level problem solving actions is shown in figures 6 and 7. Figure 6 represents a typical interpretation grammar. The subscripts indicate that, for example, an A_n can be derived from a C_n and a D_n, a C_{n-1} and a D_n, a C_{n+1} and a D_n, a C_n and a D_{n-1}, etc. Given these rules, there is a great deal of ambiguity in this grammar. This grammar is based loosely on the vehicle tracking domain of the Distributed Vehicle Monitoring Testbed (DVMT) [6]. Figure 7 shows the same grammar modified to include a class of meta-level operators referred to as *goal operators* presented in [27]. This grammar is analyzed at length in Section 7.

An extended IDP domain theory that includes rules representing meta-operators for reasoning about subproblem interactions (i.e., a sophisticated control architecture) is mapped to a corresponding *UPC* representation where the subproblem relationships, or search space structures, implied by the existence of meta-operators are explicitly represented. This is illustrated in Fig. 8. In general, the abstractions and approximations associated with relationships among subproblems, and between subproblems and final solutions, are represented by abstract states in *projection search spaces* (or *projection spaces*) that are derived from a *base search space*. The base search space, or base space, is the search space defined by the generational model of an IDP. The base space contains no abstract or approximate states. In *UPC* models, meta-level processing and base-level processing are integrated into a unified representation where search paths connect states in the base search space with abstract states in projection spaces and abstract states in projection spaces with final states in the base search space. Consequently, sophisticated

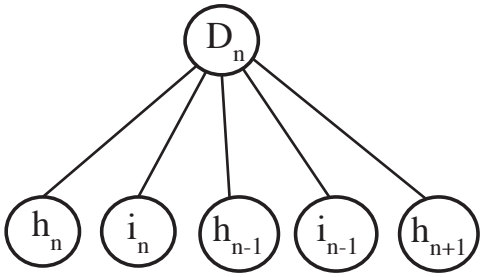
1. $A_n \rightarrow C_{n\pm1} D_{n\pm1}$
2. $B_n \rightarrow D_{n\pm1} E_{n\pm1} W_{n\pm1}$
3.a. $C_n \rightarrow f_{n\pm1} g_{n\pm1}$
3.b. $C_n \rightarrow f_{n\pm1}$
3.c. $C_n \rightarrow g_{n\pm1}$
4.a. $D_n \rightarrow h_{n\pm1} i_{n\pm1}$
4.b. $D_n \rightarrow h_{n\pm1}$
4.c. $D_n \rightarrow i_{n\pm1}$
5.a. $E_n \rightarrow k_{n\pm1} j_{n\pm1}$
5.b. $E_n \rightarrow k_{n\pm1}$
5.c. $E_n \rightarrow j_{n\pm1}$
6.a. $W_n \rightarrow x_{n\pm1} y_{n\pm1} z_{n\pm1}$
6.b. $W_n \rightarrow x_{n\pm1} y_{n\pm1}$
6.c. $W_n \rightarrow x_{n\pm1} z_{n\pm1}$
6.d. $W_n \rightarrow y_{n\pm1} z_{n\pm1}$
6.e. $W_n \rightarrow x_{n\pm1}$
6.f. $W_n \rightarrow y_{n\pm1}$
6.g. $W_n \rightarrow z_{n\pm1}$

a. extended interpretation grammar, G_n'



(signal data)

b. example structures from grammar



c. additional intermediate structures from grammar

Figure 6: Interpretation Grammar

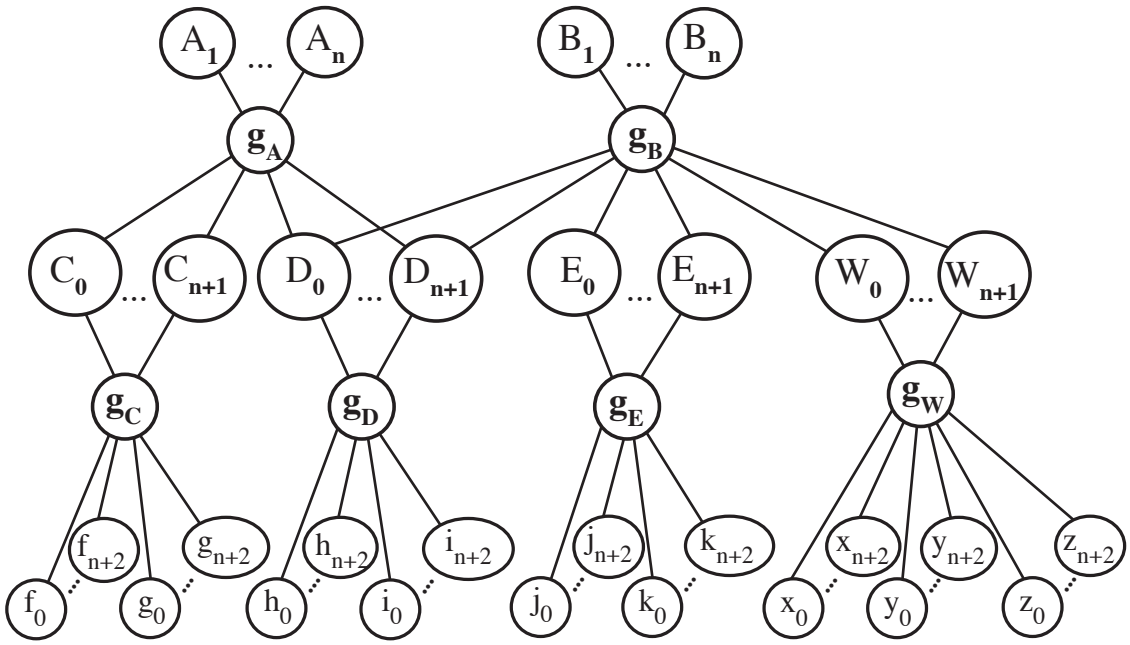


Figure 7: Representing Goal Processing in an Interpretation Grammar

control mechanisms are mapped into a state space representation where they can be viewed from the same perspective as traditional problem solving actions. Figure 8 depicts the integrated search space. This integrated perspective, which is discussed more fully in Section 11, suggests a new model of search where domain problem solving can be viewed as taking place incrementally and simultaneously in a continuum of abstraction spaces. (Abstract search spaces are referred to as *projection spaces*.) Sections 9 and 11 describe the search space structures corresponding to meta-operators.

In the IDP/*UPC* framework, by choosing the next problem solving action to perform, the control component determines the projection space in which problem solving will occur, and the operator which will carry out the action. For example, by choosing meta-operators, the control component projects one search space (possibly the base space) to another, more abstract space where certain subproblem interactions are explicitly represented. Alternatively, the control component can choose an operator that extends one or more partial solutions within a given projection space (i.e., carry out problem solving in a projection space), or it can "map back" (or *refine*) an abstract projection space to a less abstract space, possibly the base space. Mapping an abstract state to the base space propagates the implications of any subproblem interactions back to the utility, probability, and cost values of the operators that can be applied to states in the base space. This perspective is illustrated in Fig. 9. To represent the relative worth of executing a meta-operator in a manner that can be used by a control component's evaluation function, a metric has been developed which is referred to as *potential*, for quantifying these relationships.

The concept of potential is a critical element of the IDP/*UPC* analysis framework and it is used to address the question of how to evaluate the contribution made by meta-level control actions that use abstractions and approximations in terms that are consistent with the evaluation of problem solving actions that directly extend search paths in the base space. The concept of potential is applicable to all operators, but it is especially relevant to meta-operators. This is because, in general, meta-operators are not associated with effects that can be quantified in the same way as the effects of base space operators.

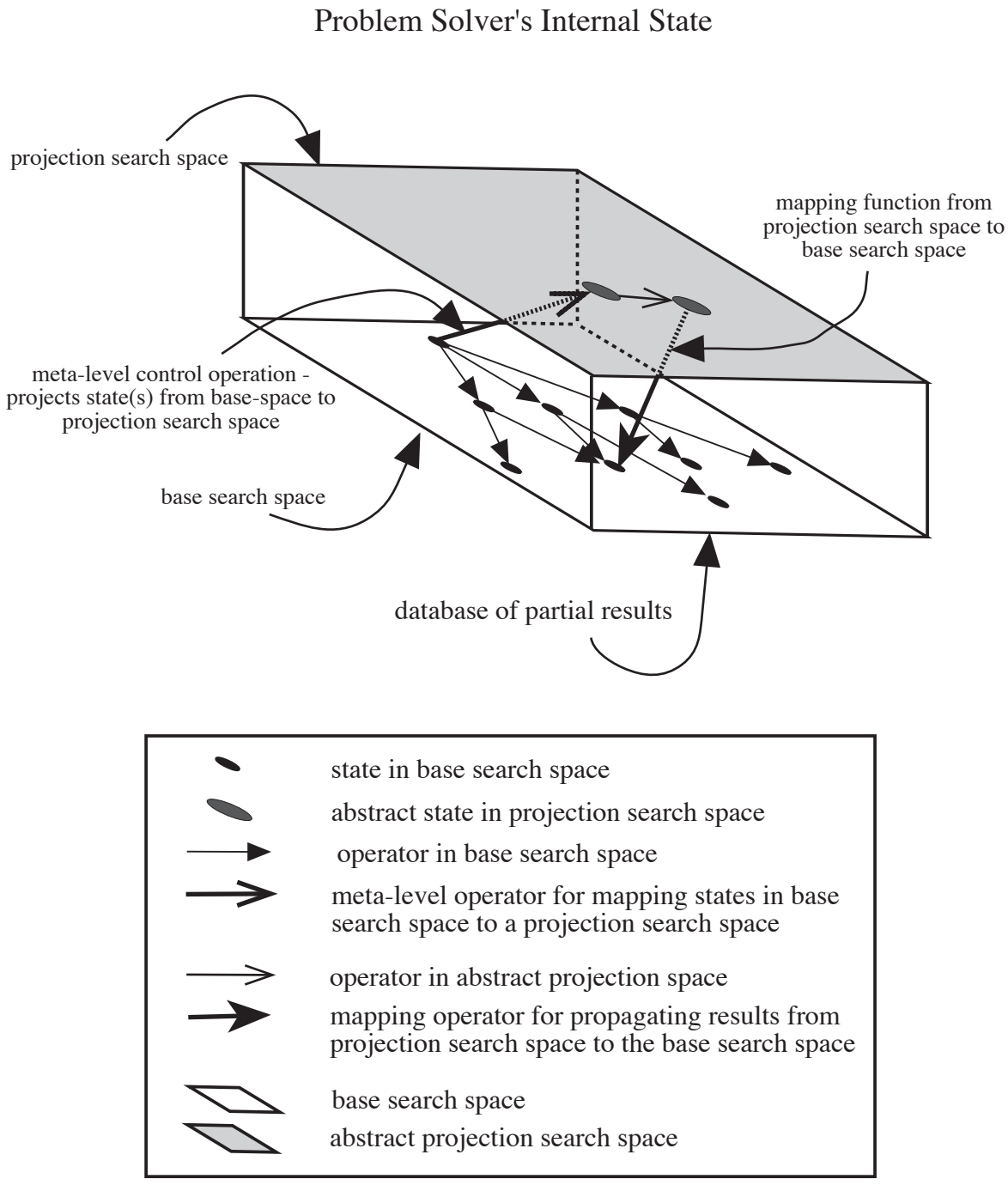


Figure 8: Explicitly Representing Meta-Level Control Actions as Components of a Problem Solver's Internal State

The effects of meta-operators are related more to long-term reductions in problem solving cost or increases in solution quality. In contrast, the effects of base space operators are more closely associated with immediate effects resulting in the extension of base space search paths and they can be more easily quantified. Thus, potential will be used to develop mechanisms that support the the understanding of the interrelationships that exist between the current set of states (i.e., the search paths that have been extended so far) and the states that can be derived from them. This includes using potential as a measure of the *expected long-term effects* an operator will have on problem solving in the base space. From a statistical perspective, potential takes into account the changes in the *UPC* representation of base space states that occur as a result of the added information provided by a meta-operator. Thus, although a meta-operator may have no immediate effect on any base space search paths, which might appear to make it an undesirable choice of action, it may have a very significant long-term effect that reduces the expected cost of problem solving dramatically, making it a very good choice of action.

For example, there may be several operators available to extend paths from a state, s_n, to other states in the base space. In addition, there may be an operator available that will extend a path from s_n to an abstract state in a projection space. All of the base space operators may appear to be attractive in the sense that, from a local perspective, there appears to be a high-probability that the paths generated by the base space operators will eventually lead to final states with high credibilities. However, if the meta-operator is executed, it may generate an abstract state that indicates only one specific final state, F, is reachable from s_n. Thus, all of the operators that do not extend paths from s_n that might eventually reach F can be pruned. Another operator, which is called a mapping operator or mapping function, can then be executed to transfer this information back to the base space by modifying the *UPC* values of states in the base space to reflect that only the operators that can extend paths that might eventually reach F should be considered.

In the IDP/*UPC* analysis framework, projection spaces and their associated operators (including the projecting and refining operators) are viewed as *UPC* instantiations of the corresponding IDP domain theory model. The projecting, refining, and problem solving operators are represented as part of an IDP model of a domain theory. The component structure suggested by meta-operators, or meta-rules, is integrated into an IDP model as a set of grammar production rules and associated utility, cost, and distribution functions. Formulating problem solving in this way unifies two forms of problem solving, domain problem solving and meta-level control, that have recently been viewed as distinct classes. In addition, formulating meta-level control in this way allows a problem solver to determine abstraction levels dynamically or to alternate the level of abstraction at which problem solving occurs in order to opportunistically exploit the results of intermediate problem solving. The IDP/*UPC* framework is particularly effective for analyzing problem solving systems, such as the extended Hearsay-II [12] blackboard model introduced by Lesser and Corkill, that integrate both *top-down* and *bottom-up* processing in a hierarchy of abstraction spaces [5, 6]. To our knowledge, no other analysis framework provides a perspective where different approaches to control can be analyzed as part of a unified domain theory.

One of the primary analytical uses of the IDP/*UPC* framework will be to compare and contrast the costs and benefits of alternative meta-operators in different domains. This will be achieved by defining domain independent *objective strategies* which are simple algorithmic statements such as "find the best solution," or "find the least cost solution," that will be used by a problem solver to make decisions regarding which action to take next. Inherent in the objective strategies used in the IDP/*UPC* framework is a perspective from which problem solving is viewed as an attempt by a problem solver to connect all the states in the base space by extending all potential search paths until they reach dead-ends

Exp	Generation				Interpretation				Sig	% C
	G	Dist	U	$E(C)$	G	Dist	U	Avg. C		
1	1	even	0.5	201	1	even	0.5	203	N	100
2	2	even	0.5	189	2	even	0.5	187	N	80
3	3	even	0.5	180	3	even	0.5	181	N	80
4	1	skew	0.5	368	1	even	0.5	369	N	100
...	...	...	...	...	...	...	...	...	...	...

Abbreviations

Exp: Experiment

G: The problem solving grammar used;
1: G'
2: G' and bounding functions with cost 10,
3: G' and bounding functions with cost 1,

Dist: Distribution of Domain Events;
even: domain events evenly distributed
skew: distribution skewed to more credible events,

U: expected problem instance credibility;
0.5: problem instances have expected credibility 0.5
0.25: problem instances have expected credibility 0.25
0.75: problem instances have expected credibility 0.75

$E(C)$: Expected Cost of problem solving for given grammar

Avg. C: actual average cost for 1000 random problem instances

Sig: Whether or not the difference between expected cost and the actual average cost was statistically significant
Y: yes, there is a statistically significant difference
N: no, there is not a statistically significant difference

% Correct: percentage of correct answers found

Table 1: Example of IDP/*UPC* Experiments Comparing Alternative Meta-Level Control Architectures

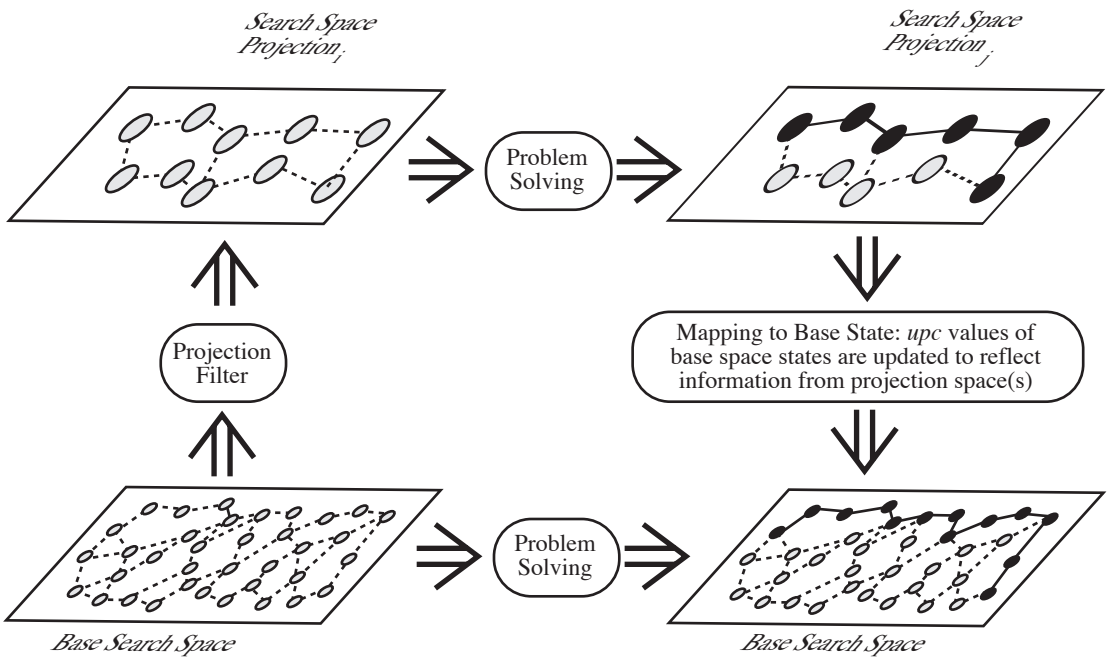


Figure 9: The Extended IDP/*UPC* Control Perspective

or final states. (Objective strategies are discussed further in Section 2 and formally defined in Section 8 and Appendix A.) In the IDP/*UPC* framework, knowledge-based actions that might otherwise be thought of as part of a control architecture are stripped out (leaving only the objective strategy) and represented in the same form as base-level problem solving actions. This enables direct comparisons to be made of the costs and benefits of alternative meta-operators in different domains. Thus, given a meta-operator, its expected performance characteristics can be identified for different domains, it can be compared and contrasted with other meta-operators, and classes of related meta-operators and domains can be identified. Example experiments are described in Sections 10 and 12. A summary of these experiments is shown in Table 1. In the different experiments shown in the table, elements of the grammar representing meta-level control and characteristics of the domain are altered and the resulting problem solving performance is compared. In experiments 2 and 3, specific meta-control operators that are referred to as bounding functions are added to the grammar resulting in a decrease in the expected (and observed) cost of problem solving. In experiment 4, the mechanisms used to generate domain events were altered to generate more noise and missing data than originally expected. This resulted in an increase in the actual cost of problem solving. As shown in the table, using the IDP/*UPC* framework, certain performance characteristics, such as the cost and quality of problem solving, can be predicted and and the predicted values and actual values can be compared. This can be done to verify aspects of the framework or to investigate unknown phenomena. The basic control mechanism used to generate the experimental results is described in Section 8.

The IDP/*UPC* framework provides a basis for establishing design theories for classes of AI search problems. The framework accomplishes this by clearly representing how assumptions about the characteristics of domain events and the structure of domain knowledge affect control architectures applied to the resulting search space. Furthermore, the framework analytically and empirically addresses two critical design issues, the *synthesis* problem and the *selection* problem. The synthesis problem, which

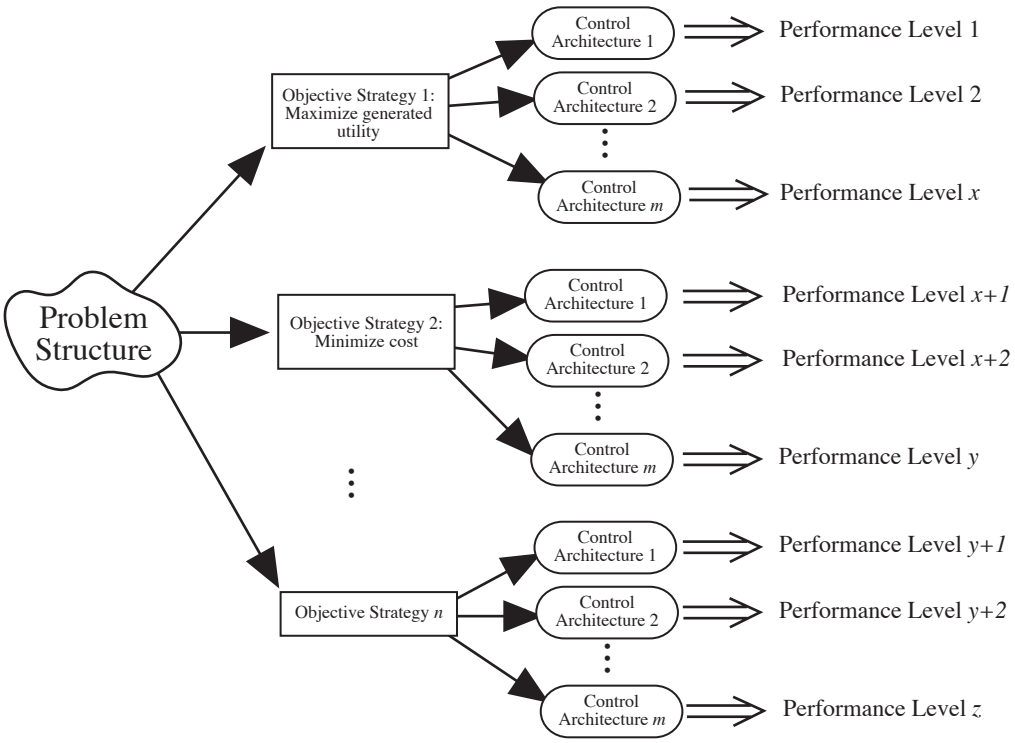


Figure 10: Illustration of the *Selection Problem* – A Given Problem Structure Implies Different Levels of Performance for Different Control Architectures

involves the generation of approximations and abstractions for a given interpretation domain theory, corresponds to adding meta-operators (and associated functions) to the grammar component of a domain's IDP model. Section 7 presents examples of how the IDP formalism can be used to address the synthesis problem for a specific form of dynamic, hierarchical problem solving: goal processing.

The selection problem, which involves choosing approximations and abstractions for use in problem solving, is represented in Fig. 10. As shown in this figure, given a problem structure and an objective strategy such as "find the highest rated solution," or "find any solution as quickly as possible," the core thesis of this work is that different control architectures, when applied to the same problem structure, result in different levels of problem solving performance[3]. The IDP/*UPC* framework will address the selection problem by helping to identify the causal relationships between problem structures, control architectures, and performance levels. This information can then be used to dynamically choose the appropriate control architecture based on the observed problem structure.

Some of the philosophical and paradigmatic approaches that lead to the development of the IDP/*UPC* framework are discussed in Section 2. Relevant background material is summarized in Section 3. Interpretation problems and some of their important structural characteristics are presented in Sections 4, 5, and 8 along with a general description of the IDP formalism. Quantitative analysis methods based on the IDP are presented in Section 6. Section 7 presents an example showing how the IDP model can be used to define problem structures in the Distributed Vehicle Monitoring Testbed (DVMT) [6]. The *UPC* formalism is introduced in Section 9. Section 10 presents experimental verification of the IDP/*UPC* framework. Section 11 introduces as extension of the *UPC* formalism

[3]Problem solving performance can also be thought of *system behavior*.

that will support the modeling and analysis of meta-operators and Section 12 defines and discusses the concept of potential that is necessary for analyzing the performance implications of sophisticated control architectures that use abstractions and approximations. This section also presents the results of our initial experiments with Potential.

2 Philosophical and Paradigmatic Implications

The IDP/*UPC* analysis framework is based on a philosophy emphasizing the importance of a problem domain's *structure*. At its simplest, this philosophy holds that the "AI universe" is structured and bound by laws and principles in much the same way that the "chemical universe" is structured and bound by laws expressed in the periodic table of elements or that the "physics universe" is structured and bound by the theory of relativity or by quantum theory.

As with the laws and principles that we associate with the physical sciences, e.g., the law of gravity, the theory of relativity, etc., the laws and principles discovered by AI scientists are meant to be applied to the models of reality that we construct to explain the natural world. In addition to forming the basis for continuing scientific research, these "models of reality" that scientists construct are subsequently used to develop design theories for building artifacts that will operate in the natural world, artifacts such as speech understanding systems, image understanding systems, etc. In both these endeavors, the key element is the model of reality that is constructed to explain and predict events in the natural world.

Thus, the IDP/*UPC* philosophy asserts that the occurrence of phenomena in any AI domain, not just the interpretation domains that are studied here, can be described formally and that this formal description is structured and constitutes the "causes" of the phenomena. Furthermore, the control architecture of a problem solver can exploit the structure of a domain's formal description in order to improve the performance of a problem solving system.

The primary contribution of the IDP/*UPC* analysis framework is that it provides a formalism for expressing the structure of the natural world in a way that can be used both for scientific analysis and for constructing design theories. In this paper, we will demonstrate both of these capabilities. In particular, we will show that control and problem solving actions can be viewed from a unified perspective in terms of a problem domain's structure and we will show how, for a given problem structure, theories can be constructed regulating the design of "meta" or "control" operators. Eventually, we plan to use the IDP/*UPC* analysis framework to develop a very broad perspective of problem solving that unifies diverse problem solving approaches including AI search techniques and techniques normally associated with operations research such as linear programming.

In the IDP/*UPC* framework, analysis of a problem solving system requires the explicit consideration of four elements: a problem's structure and a problem solver's *objective strategy*, *control architecture*, and *performance level (or behavior)*. In this paper, we will think of a problem's structure as a definite pattern of organization of the properties governing the creation of problem instances of a specific domain. The IDP formalism, introduced in Section 1, expresses structures in the form of phrase structured grammars with context-free production rules[4] and functions associated with rules of the grammar. Details of the IDP formalism are given in subsequent sections.

Furthermore, in the IDP/*UPC* analysis framework, every problem solver will be associated with

[4]The grammar representation used in the IDP/*UPC* framework is an extended form of the traditional context-free grammar representation. The extended form is used to explicitly represent information needed for analysis.

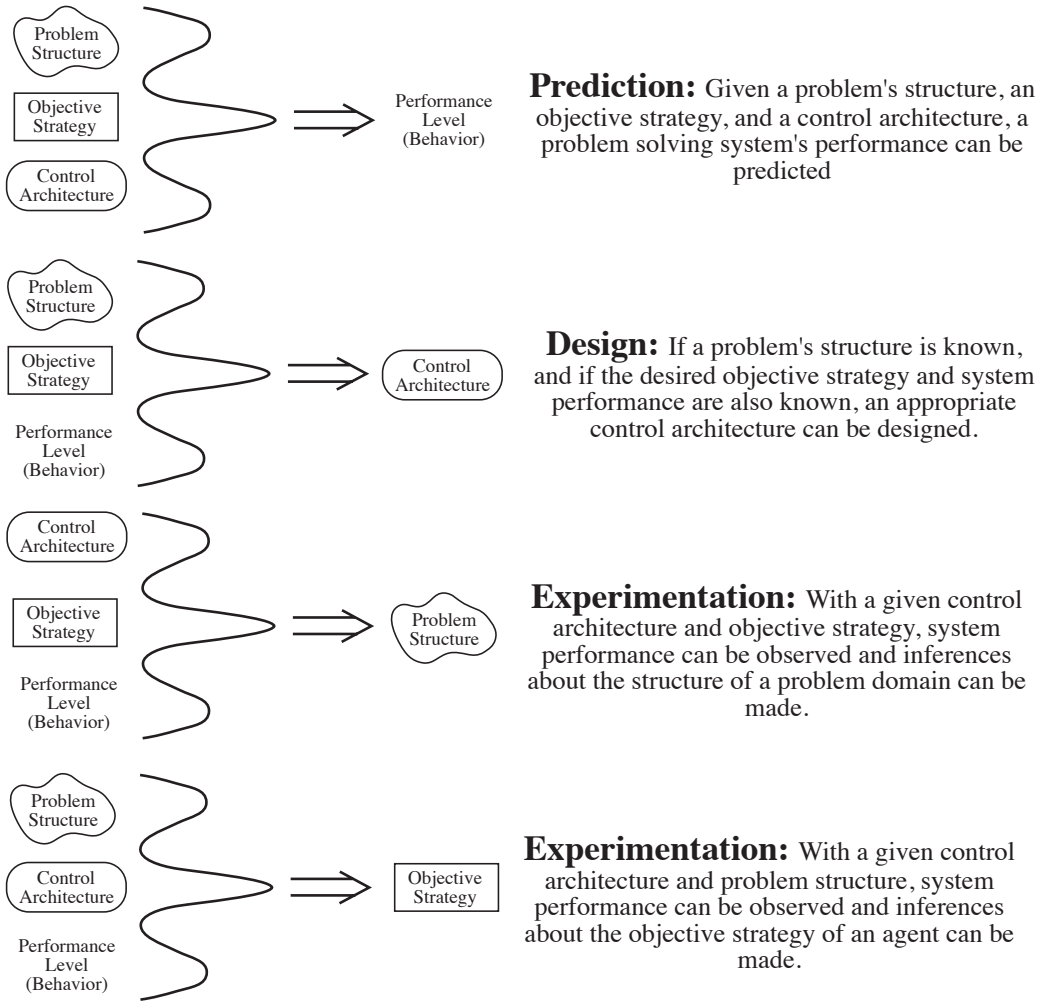


Figure 11: Summary of Analysis Perspectives Supported by the IDP *UPC* Framework.

an objective strategy that defines the goals a problem solver is trying to achieve. For example, simple objective strategies include, "find the least cost solution," or "find a solution as quickly as possible," or "find the best solution." The objective strategy of a problem solver can be thought of as being analogous to the *objective function* of problem solving strategies such as the simplex algorithm [33].

In a typical analysis situation, the object of consideration will be the control architecture – the algorithm(s) used by a problem solver to choose its next problem solving action. Control architectures will be expressed in terms of the problem structure. A given control architecture will be represented as rules (and associated functions) of the grammar used to define the problem domain.

In the IDP/*UPC* analysis framework, a problem solver's performance will primarily be measured in terms of the expected resources required to solve a problem and the expected "quality" of solution, where quality will normally be associated with a percentage of correct answers found in a series of test cases.

Though the primary emphasis of the analysis framework rests on the importance of a problem's structure, the significance of the other three elements should not be overlooked. In fact, the existence of these four elements implies a variety of analysis paradigms. These are summarized in Fig. 11, which illustrates the relationships between a problem's structure and a problem solver's objective strategy, control architecture, and performance level (or behavior). As shown in the figure, and as will be discussed in this paper, there are four basic analysis paradigms, *prediction*, *design*, *experimentation (problem structure)*, and *experimentation (agent objective strategy)*.

In the first paradigm, prediction, analysis focuses on predicting a problem solver's performance based on a given objective strategy, control architecture, and problem structure. In this paper, we will demonstrate the validity of the IDP/*UPC* framework by showing that a problem solver's performance can be accurately predicted based on a formal analysis of the objective strategy, control architecture, and problem structure.

The ultimate objective of this work is to develop methodologies for constructing design theories. This objective is addressed by the second analysis paradigm, which will be referred to as the design paradigm. The design paradigm is not discussed in this paper. Also, it should be noted that the synthesis and selection problems defined in Section 1 are addressed by the design analysis paradigm.

The two experimental analysis paradigms can be used to infer either a problem structure or an objective strategy given the other analysis elements. These forms of analysis will also be discussed further in subsequent publications. For now, it is interesting to note that in the experimentation paradigms, the control architecture actually becomes an experimental tool rather than the object of analysis. In an experimental analysis paradigm, the control algorithm can be chosen or modified in order to determine the structure of a problem domain or to determine the objective strategy being used by a problem solving agent.

For example, a researcher may wish to investigate the structure of a particular image interpretation domain. The researcher may carry out a series of experiments in which she makes assumptions about the structure of the domain, embeds these structures into an IDP representation, and then designs a control architecture to exploit the assumed structure. By implementing both a simulation of a problem solver operating in the domain with the assumed structure as well as a real problem solver operating in the actual domain, the researcher will be able to conduct comparative studies to determine if the assumptions about the domain's structure of the image interpretation domain are reasonable.

3 Background

Heuristic search is an important Artificial Intelligence (AI) problem solving paradigm that, over the years, has been used as the foundation for a great variety of AI systems. The search paradigm is based on the concept of a space of *states* (or *nodes*) – data structures characterizing relevant features of the problem domain. (These data structures will be referred to as *characteristic variables* or *CVs* throughout this paper.) Within this space, *search* is the process of "moving" through these states via domain specific *operators* that map states to states until a desired *final state* is found. Search invariably begins in a domain specific *start state* and as operators are applied "paths" are traversed between states. Search is often thought of as a process of *expanding* a state by applying one or more operators to it in order to generate new states. Each state, s_i, represents a *path* from the start state to s_i and expanding s_i is equivalent to extending the corresponding paths. A path that extends from the start state to a final state is referred to as a *solution* and paths that extend from the start state to intermediate states are sometimes referred to as *partial solution paths* or *partial solutions*. The general strategy for controlling the application of problem solving operators is based on the use of *evaluation functions*. As a state is expanded and new states created, an evaluation function gives each operator associated with the new states a rating that represents its relative worth or merit. The ratings are then used to determine an ordering for expanding paths.

Figure 12 is a representation of the internal state of a search-based, interpretation problem solver. As shown, the internal state of a problem solver is defined primarily in terms of the partial solution paths that it has expanded. The other components are the operators available for execution and the potential final states implied by the available operators. When a search path is created that represents a partial solution to a problem, at least one potential final solution is also created that, if generated, will include the partial solution. The importance of the implied potential final states is related to the analytical tools that will be defined and to the relationship between the analytical framework and previous work in search based problem solving. Specifically, the concept of an implied final state is common to most search algorithms. For example, A* search is defined in terms of the final states implied in its characteristic equation $f^*(n) = g^*(n) + h^*(n)$. In this equation, $f^*(n)$ is an evaluation function for state n, $g^*(n)$ estimates the minimum cost path from the start state to n, and $h^*(n)$ represents an underestimate of the distance from n to a goal state (or final state).

If a partial solution is created and no corresponding potential final solution can be created, the path to the partial solution will be referred to as a "dead end" and all operators that can be applied to the partial solution can be eliminated from consideration. A "dead end" state is one for which one of the following conditions is true; if there are no operators that can be applied to extend paths from the state; or if the problem solver determines that there are no paths from the state that will reach a final state; or if the problem solver determines that there is no way that the final states that can be reached from the state will have the highest utility (credibility).

When the internal state of a problem solver is such that all implied states have been generated and explicitly represented and there are no more operations that can be applied, the problem solver has reached *termination.* This is the same as saying that the base space has been connected - all paths from intermediate states have either been generated or terminated because the problem solver has determined that they cannot possible lead to the correct interpretation. It is also important to note that a problem solver, in trying to connect its internal state and reach termination, is also defining a search space. The proposed analysis techniques will be associated with the search paths of a problem solver's internal state.

As a typical problem solver explores a search space by expanding states and extending partial

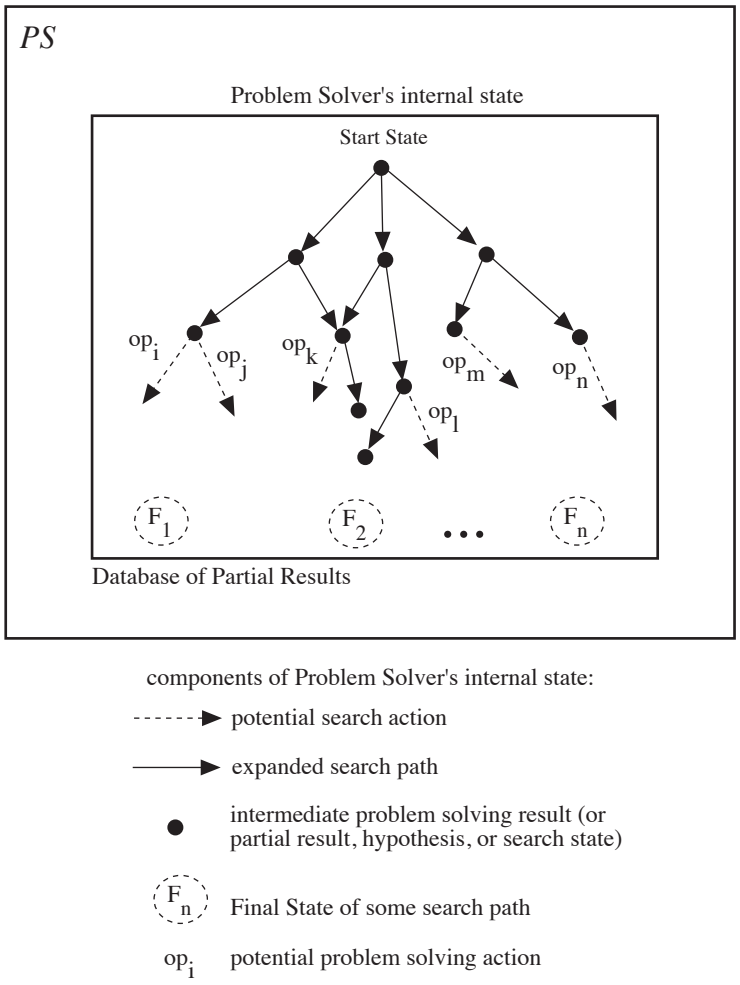


Figure 12: Representation of the State of the Problem Solver

solutions, the size of the search space, when measured in terms of the number of open solution paths plus the number of dead-end states plus the number of final states generated, grows at an exponential rate. Shown graphically, search spaces take on a tree-like structure where the start state is the root of the tree and the branches descending from each state represent the result of applying one or more operators to that state. In a typical problem, the branching factor ranges from only four or five to hundreds or thousands. Furthermore, a typical problem will require at least dozens of operator applications, resulting in dozens of levels to the "search tree." In general, the total size of a search space is calculated (approximately) by b^d where d is the depth of the tree and b is the branching factor of each state. (This calculation is approximate because the tree depth or branching factor could vary within a search tree.)

As a result of the enormous size of a typical AI search space, it is impractical or impossible to build problem solvers that are based on exhaustive search techniques. Instead, AI problem solvers employ strategies that selectively expand relatively small portions of the search space and then use these partial results to make inferences about the consequences of expanding much larger portions of the search space. These *implicit enumeration* strategies are often referred to as *control* or *meta-reasoning* strategies. (In this work, they will be referred to as meta-level, or abstract, operators.) The implementation of a control or meta-reasoning strategy will be referred to as a *control architecture* [4, 19, 18].

The analysis framework is intended to be used to analyze *sophisticated control* architectures (sophisticated control will be fully defined in Section 3.2) in *complex domains* (complex domains are defined in the following subsection). Analysis will focus on architectures that use *abstract*, or *approximate*, reasoning mechanisms, which explicitly represent subproblem interactions, to implement efficient implicit enumeration strategies for interpretation problems.

The following subsections will define and contrast complex domains and restricted domains. They will also define and contrast the associated sophisticated control architectures and local control architectures. Figure 13 summarizes these topics and illustrates their relationships to each other relative to *monotone* and *non-monotone* domains, which are also defined and discussed in the following subsections.

3.1 Complex and Restricted Problem Domains

Early work on the search paradigm was restricted to constrained domains such as game playing [1, 35], and theorem proving [32]. The heuristic knowledge used in the search process was relatively limited. This can be expressed formally by saying that, for a restricted problem domain such as game playing or logic, $cost(operator_i) = O(1)$, i.e., the cost of applying an operator is a constant value. Thus, in a restricted domain, the cost of evaluating and expanding a single state is $O(1)$ and the cost associated with problem solving is a function of the number of times this constant cost must be incurred. Since there is little or nothing that can be done to reduce the incremental cost of expanding a single state, the most appropriate problem solving strategy is to reduce the number of states that are expanded by pruning paths wherever possible.

For the most part, the pruning strategy formed the basis for algorithms such as AO*, B*, SSS*, Alpha-Beta, and their generalizations [1, 25, 34, 38]. The strategy incorporated in these algorithms is based on the use of problem structures manifested in a problem solver's evaluation functions – ratings and knowledge of the search space's structure are used to determine which paths are most likely to lead to final states and which paths are dead ends. Paths that are more likely to lead to final states are expanded first, and the results are used to prune alternative paths and dead ends. In research projects involving restricted domains, the emphasis was on developing algorithms that enabled the problem

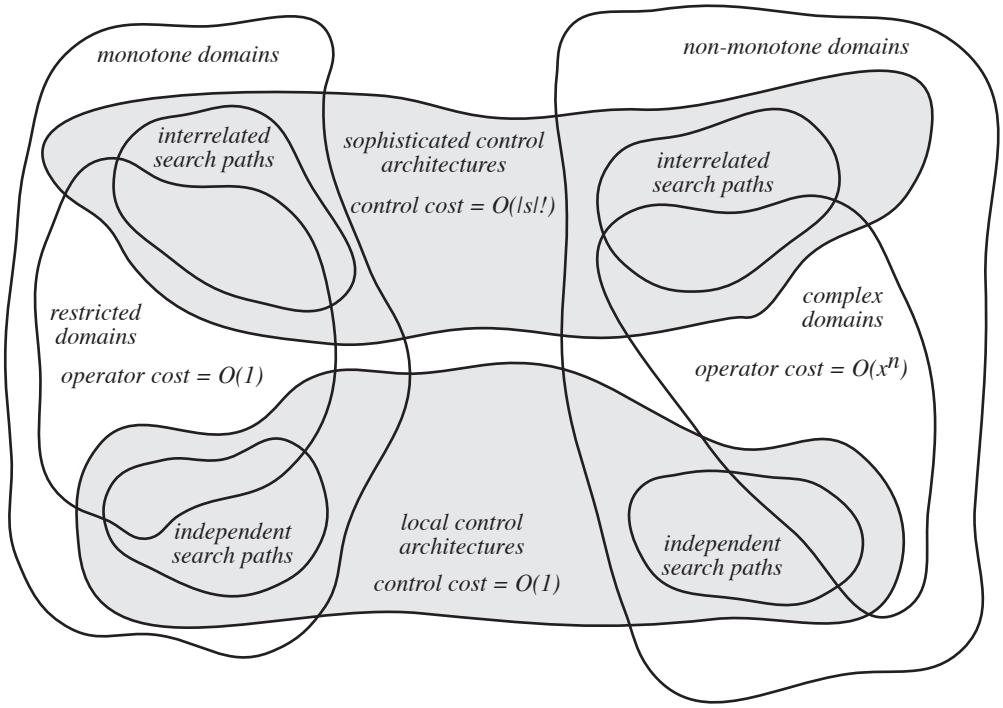


Figure 13: Classification of Problem Domains

solver to prune paths based on intermediate problem solving results [1, 25, 34, 38].

Previously, Pearl [34] has shown that statistical properties of a problem solving technique, such as expected cost, can be determined from an analysis of the structure of a graphical representation of the search paths explored by the problem solver. In particular, Pearl examines the effects of pruning operators and heuristics for ordering the application of operators in various game playing domains and in other restricted domains. In contrast to the proposed analysis techniques, the analysis techniques described by Pearl in [34] do not take into consideration dynamic subproblem interactions or the long-term effects of an action.

Extended studies and analyses of the work on search techniques for restricted domains have resulted in the identification of a taxonomy of search procedures by Kanal and Kumar [25]. Each of the procedural categories specified by Kanal and Kumar is defined in terms of the fundamental structure of the search spaces to which the procedures included in the category can be successfully applied. Kanal and Kumar's taxonomy links heuristic search techniques with problem solving techniques that are more closely associated with operations research (OR). This work has shown that basic AI search techniques can be classified as belonging to a subcategory of either *branch-and-bound* procedures or *dynamic programming* procedures [25].

More recently, AI problem solving has ventured into *complex domains* such as speech recognition, natural language processing, vision processing, pattern recognition, etc., that differ from the earlier, more restricted domains in two important ways. First, the search processes used in more complex domains do not exhibit the simplifying characteristics that would enable them to be categorized according to Kanal and Kumar's guidelines. This is in contrast to the characteristics of the restricted domains that imply the existence of certain types of search space structures that can be exploited during problem solving. These structures do not exist in the more complex domains and most of the search

and pruning techniques that were developed in earlier work, such as AO*, B*, SSS*, Alpha-Beta, and their generalizations, are not applicable [25].

There are a variety of reasons for this. The most significant is that complex problem formulations do not exhibit *monotone* properties and structures. In a complex problem domain, certain characteristics of a search path are not guaranteed to increase (or decrease) monotonically, in relation to alternative paths, as the path is extended. For example, in a complex interpretation problem domain, as a path is extended, its credibility is not guaranteed to increase or decrease in a way that is monotone in relation to alternative paths. At any given point, the credibility of a path could go from "high-certainty" to "low-certainty," or vice–versa, while the credibility of alternative paths, which are extended with the same low-level data, could fluctuate in the opposite way. Kanal and Kumar's taxonomy only classifies problems that are, minimally, monotone. As discussed by Kanal and Kumar [25], in order for AO*, alpha-beta, B*, SSS* and their generalizations to be applicable to a given domain, the problem formulation for that domain must exhibit monotonic properties.

The second important distinction between restricted and complex domains is that the cost of applying an operator in a complex domain can be arbitrarily expensive, even exponential. This implies that the most efficient course of problem solving is not necessarily associated with path pruning. Instead, it might involve reducing the incremental cost of each search operator application, or replacing expensive operator applications with one or more inexpensive operators, etc. Consequently, the general strategies used to reduce problem solving costs in restricted domains must be modified or replaced when working with more complex domains.

3.2 Sophisticated and Local Control

Intuitively, local control architectures focus on ordering search activities and pruning paths based on the evaluation function ratings of intermediate problem solving results. Local control architectures are most relevant to problem formulations with well understood search space structures that limit the complexity of the knowledge and the amount of resources required to prune paths. Specifically, local control architectures are most useful in domains where search paths are independent of each other. In these domains, the merits of alternative paths can be computed by functions that are based solely on the local characteristics of the individual paths and the merits do not have to be recomputed when new states are added. (Path independence is formally defined in Section 5.) The only interaction between paths occurs in the form of comparing their respective evaluation ratings. Because the evaluation functions are based solely on local information, their cost does not vary with the size of the search space. In fact, the evaluation functions can be thought of as constant cost functions. i.e., $\forall n, scost(f_{n,s}) = O(1)$, where $f_{n,s}$ is an evaluation function for the path represented by state n in search space s that defines the internal state of a problem solver at an intermediate state of problem solving. s corresponds to a problem solver's database of partial results shown in Fig. 12. As a consequence, scheduling and pruning algorithms based on these functions can be applied with little or no thought given to the costs involved. For example, given a search space with a very accurate and discriminating evaluation function, it may be possible to prune many search paths based on comparisons of evaluation ratings. In this domain, the control component of the search system could be a relatively simple mechanism that evaluates *every* state and expands those that are determined to be the "best" states and/or prunes states when it is determined that expanding them would be fruitless.

In contrast, sophisticated control architectures are most useful in situations where search paths are interrelated in such a way that extending one path somehow affects the results of extending other

paths. The formal definition of a *relationship* between partial solution paths will be given in a later section, but for now it is sufficient to think of a relationship as a set of *constraints* that allow operators to function more efficiently. Constraints can also increase the likelihood that operators extend paths that will eventually lead to the correct solution and improve the ability of a problem solver to estimate the overall utility of a potential operator. For example, extending path A could reduce the cost of extending path B, or, perhaps, provide information that enables a better measure of the utility generated by extending path B.

Another way of thinking about this is to view the problem domain as consisting of a set of interdependent subproblems where solutions to subproblems are aggregated into an overall solution to the problem. From this perspective paths (or states) in the search space are considered to be *competing* when they lead to different solutions for the same subproblems, *independent* when they are solving subproblems that do not interact, and *cooperating* when they lead to solutions of subproblems where the solutions must be consistent with the solutions to a more comprehensive subproblem that includes the interacting subproblems as components. In the case of cooperating paths, potential solutions to a subproblem impose constraints on sibling subproblems. These ideas are very important in the IDP/*UPC* framework and a more extensive and formal description is discussed in Section 5 and in previous work of mine in [27].

In choosing a path for extension in an interrelated search space, a sophisticated control algorithm seeks to optimize the amount of constraint that is generated by the extension both locally, for the path being extended, and more globally, for other paths related to the path being extended. The objective here is not necessarily to prune paths but to efficiently order problem solving activities so as to limit the overall cost of problem solving. Since paths are related to each other, the creation or extension of a specific path will (possibly) constrain future extensions of related paths. Thus, the goal of sophisticated control can be thought of as the optimization of constraint generation to minimize the cost of search by limiting the number of path extensions, as is done in conventional search, and by limiting the cost of individual path extensions and the cost of controlling the search process.

It is important to emphasize that constraining the expansion of a state is significantly different from pruning a path. For example, the constraint may be in the form of an ordering constraint, such as "do not expand state A until state B has been expanded." Such a constraint may be beneficial in a situation where the results of expanding state B significantly reduce the cost of expanding state A. In this situation, it may be the case that no paths are pruned – the constraints generated by the expansion of state B simply reduce the resources needed by the operator that expands state A. Constraint can be manifested in the form of a delay. The problem solver may determine that it can delay certain problem solving activities in the hopes that subsequent actions will make the delayed activities meaningless and eligible for pruning. Though some of the search algorithms applied in restricted domains do delay certain problem solving actions, this delay is made with hope that other problem solving actions will allow the problem solver to prune the delayed action. The delay is not expected to reduce the cost of the action. In more complex domains, where an operator can have significant cost, reducing the cost of operator executions can be critical.

Optimizing control for constraint generation requires that analysis associated with the evaluation of a partially expanded path be based on existing alternative paths. This analysis will not be a constant cost function – the cost will vary based on the number and characteristics of existing paths. Furthermore, analyzing the relationships between paths could be a very costly computation and the incremental cost of analyzing additional path relationships as problem solving progresses could also be very high. Since the cost of evaluating a path is based partly on the number of other paths it might be related to, and

since the total number of partially expanded paths grows exponentially, the cost of evaluating a single path in a complex domain could grow at a combinatorial rate. (i.e., $cost(f_{n,s}) = O(\mid s \mid !)$, where $f_{n,s}$ is an evaluation function for the path represented by state n in search space s, and $\mid s \mid$ represents the cardinality of s, i.e, the number of states expanded[5] so far and represented in s where s defines the internal state of a problem solver at an intermediate state of problem solving.)

Thus, the critical difference between sophisticated and local control is embodied in their respective costs. In a simple domain where solution paths are independent, the issue of control can be addressed by some control architecture that evaluates every potential path and uses the resulting information to efficiently expand and prune paths. It is usually assumed that the control component of such a problem solving system has cost $= O(1)$, the cost does not recur (i.e., paths are not re-evaluated in light of any new paths that are generated), and the problem solver need not be concerned with considerations associated with this cost. It is important to recognize that, even though evaluating a single path may have cost $= O(1)$, the number of paths that must be evaluated will grow exponentially. However, since the paths are independent, the addition of a new path does not affect the ratings of other, previously evaluated paths. Consequently, the incremental cost of evaluating the new paths that result from an expansion will still be $O(1)$. This form of search control, i.e., control architectures that are determined a priori, that are not modified based on intermediate problem solving results, that have cost $O(1)$, and that are not based on relationships between paths, will be referred to as *local, independent state evaluation control* or simply *local control.*

In more complex domains, the issue of control will require the use of a more costly evaluation mechanism and the problem solver must take this cost into consideration. This is a result of the fact that the cost of evaluating a single path is a function that grows at a rate that is potentially exponential in terms of the size of the search space. Furthermore, since paths in a complex domain are not independent, the expansion of a single path implies that *all* previously evaluated paths may need to be reevaluated, each at a potentially exponential cost. Because of the expense that may be incurred by these evaluations, it may not be practical to use a control mechanism that attempts to maximize the amount of constraint produced or seeks to find the "best" state for expansion. For example, if the cost of determining the "best" state for expansion has a cost that exceeds the cost of extending *all* the available paths, it is clearly more desirable to simply extend all the paths. Thus, it is no longer feasible to simply allow the control component to conduct exhaustive processing in order to determine which state to expand. Such a decision could be more costly than simply using exhaustive processing to solve the original search problem. This form of search control, i.e., control architectures that are determined dynamically based on intermediate problem solving results, that have a worst-case cost $O(\mid s \mid !)$, and that are based on relationships between partially expanded search paths, will be referred to as *non-local, dependent state evaluation control* or simply *non-local control.* Non-local control will also be referred to as *sophisticated control.*

As a consequence of sophisticated control, the problem solving process takes on a recursive quality where the control issue becomes a search problem in itself [3]. The control component becomes a knowledge based mechanism that reasons and searches for the best operator. The control mechanism must reason about which states to evaluate, when to evaluate them, and which evaluation architectures to use. For this reason, control problem solving will be represented in the proposed formalism as an

[5] In evaluating a potential action, it may be necessary to understand its relationship to partial solution paths, and also to paths that have terminated in dead ends and paths that have been extended to final states. Paths leading to dead ends and final states may represent important relationships which indicate that the current path can be pruned, or that it is very important, etc.

incremental search process based on the use of primitive operators.

3.3 Representing Complex Domains

The analysis framework relys on an explicit representation of the abstract and approximate search spaces that are used by a problem solvers meta-operators. Other researchers have reported analyses of the structures of abstract and approximate search spaces. For example, Knoblock discusses the maximum potential reduction in problem solving costs that can be achieved with the use of abstract and approximate search spaces [24]. However, his analysis does not explain how these reductions might be achieved and his techniques do not address the issues presented in Section 1.

In addition, the use of formal grammars and the associated graph structures as a basis for analyzing interpretation problems and for constructing problem solving systems is extensive. One of the more extensive uses of formal grammars is presented by Fu in [14]. Fu uses a representation he formalizes as a stochastic grammar that is virtually identical to component structure of the proposed framework. However, Fu's work differs from the proposed work in several important ways. Most importantly, Fu's emphasis is on parsing. He does this by representing the semantics of a visual scene with a grammar. Thus, he encodes relationships between primitives and nonterminals with additions to the grammar. For example, he makes relationships such as "above," "next to," "inside of," etc., elements of the grammar. Then he uses traditional parsing techniques to derive the semantics of the scene.

In the analysis famework, the processes used to solve a problem are represented as a context free grammar. No claims are made about representing the semantics of a domain explicitly with the grammar. In fact, semantic properties are represented with function, Γ, that are arbitrarily complex functions. The approach used in the proposed framework is intended to formalize the subproblem relationships in such a way that statistical analysis can be done to determine general properties of the domain. These properties can be used be a problem solver's control component to effectively order activities.

4 Interpretation Problems and the IDP Formalism

The IDP formalism describes a class of problems, *Interpretation Problems*, in terms of a search process where, given an input string X, a problem solver attempts to find the most *credible* (or "best") explanation for X. Thus, tasks where a stream of input data is analyzed and an explanation is postulated as to what domain events occurred to generate the signal data are thought of as interpretation problems – the problem solver is attempting to interpret the signal data and determine what caused it. Interpretation is a form of *constructive problem solving* based on *abductive inferencing* [22]. Interpretation is similar to a closely related form of problem solving, *classification*, and to a more distant form of problem solving, *parsing*.

More formally, let an IDP be defined as follows:

Definition 4.1 Interpretation Decision Problem (IDP) - Given X, an arbitrarily complex input signal, determine the *best* element, e, of the discrete set of all valid interpretations, I, such that $\forall i \in I, f(e) \geq f(i)$, for evaluation function f, where a valid interpretation is one that explains all the elements, $x_j \in X$.

In Section 8 we will introduce a variant of the Interpretation Decision Problem that can result in forms of problem solving that generate solutions that do not necessarily have the highest credibility

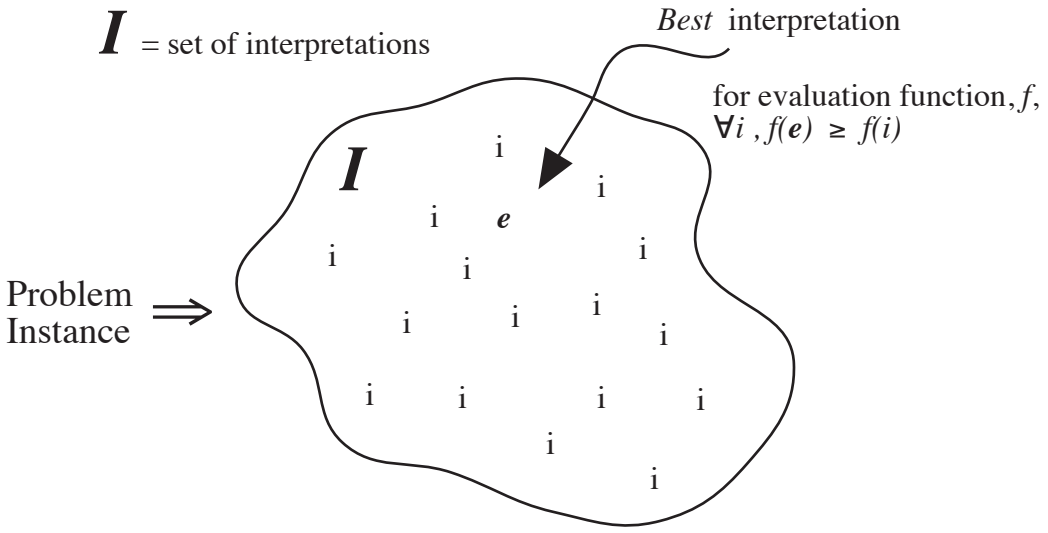


Figure 14: Representation of an Interpretation Decision Problem

rating. In this variant, the cost of problem solving can be significantly lower than the corresponding cost of problem solving in an identical IDP problem instance.

It is important to emphasize that, in interpretation problems, the set I is constructed dynamically. This is in contrast to classification problems where I is preenumerated and the problem solving task involves only the identification of the best element of I. Figure 14 is a representation of an Interpretation Decision Problem.

In most interpretation problems of interest, I is a potentially enormous, even infinite, set. In many of these domains, I can be specified in a naturally structured way and it is this structure that is exploited by control architectures to reduce the number of elements of I that must be generated or to otherwise increase the efficiency with which I is specified. Of particular interest are situations where I is finite and can be defined as the language generated by a grammar, G, and where the evaluation function, f, used during problem solving is recursively defined for strings i in I. This approach is similar to that used in the *Composite Decision Process (CDP)* model of Kanal and Kumar [25].

In these situations, interpretations take the form of derivation trees of X and the constructive search operators used in interpretation problems are viewed as production rules of G. G, therefore, defines how interpretations are decomposed. For example, the production rule $p \rightarrow n_1 n_2 n_3$ might correspond to the interpretation "p is composed of an n_1, an n_2, and an n_3." Given an n_1, an n_2, or an n_3, a problem solver may invoke the search operator o_p to try and generate a p.

Semantics associated with each production rule determine the actual domain interpretation. The semantic functions will typically make use of grammar element attributes that are not used by syntactic functions. These attributes will be represented using the *feature list convention* introduced by Gazdar, et al.[15]. This is an important point. In interpretation problems, each syntactic rule of the grammar is associated with a corresponding semantic process. (These semantic processes are discussed further in Section 4.2.) Thus, during interpretation, each application of a search operator consists of a semantic as well as a syntactic operation. Furthermore, the functionality of the semantic processes is unrestricted and could include virtually any form of processing including production rules, neural networks, etc. For example, a semantic process in a natural language interpretation task might combine a noun phrase and

a verb phrase into a sentence using a very complex process that simultaneously verifies consistency and combines the meanings of the two component phrases. Alternatively, phonemes might be combined in the same domain into words based on neural network algorithms.

4.1 Convergent Search Spaces

In the IDP formalism, the *primitive operators* available to solve a specific domain problem are represented as production rules of the domain's characteristic grammar. The mapping of interpretation decomposition to a formal grammar results in a type of search space that will be referred to as a *convergent search space*. The significance of convergent search spaces is that the relationships between search paths can be determined and analyzed based on a formal representation of a given domain. This analysis can be used to determine closed form expressions for the expected costs of problem solving. This is in contrast to domains where the relationships between search paths cannot be determined or formally analyzed.

Common to interpretation problems, convergent search spaces consist of search states that correspond to the symbols of the formal grammar's terminal and nonterminal alphabets. This is an important distinction because it limits the scope of each search state in a convergent space to an incomplete view of the search state relative to a more comprehensive, or global, view of problem solving in which relationships between individual search states is represented. Thus, each search state contains only local information associated with the specific partial solution that the search state represents. A search state may not contain any information about other, possibly interacting search states. This is in contrast to conventional search spaces, such as the typical representation of a chess domain or an 8-puzzle domain, where each search state contains complete information about a partial solution's relationship with other paths.

To clarify this point, let a search state be defined as a set of characteristic variables. A final state is then defined by a specific set of values for some characteristic variables, or by a function of characteristic variables. In a convergent space, a specific search state may not contain all the charateristic variables necessary to determine if a final state has been reached, or to determine precisely how much progress has been made toward reaching a final state. For example, in a speach understanding domain, a search state, s, may represent a partial interpretation corresponding to an interpretation of the first 2 seconds of data. There may be another 20 seconds for which s offers no explanation. However, the problem solver cannot determine if any partial interpretations for the other 20 seconds of data exist by examining s. In contrast, in a conventional search-based chess playing program, each search state is "self-contained." The problem solver can determine whether or not each of the search states is a final state, and all of the subproblem interactions are encompassed within a search state. In such domains, two paths are never merged or combined into a single, more comprehensive partial solution.

For a given set of input data, there may be multiple instantiations of a specific terminal or nonterminal symbol resulting from multiple, non-equivalent derivation paths. This will be discussed further in subsequent sections when we introduce the concept of *noise* and the associated ambiguity. Operators correspond to the inverse application of a production rule of the formal grammar. In addition, one of the key distinguishing characteristics of convergent search spaces is that an operator is applied to multiple states, not just one, but there is no explicit representation of the multiple states as a single state. Figure 15 is an example of an interpretation grammar, G', where each of the productions corresponds to a search operator[6]. Figure 16 represents portions of a convergent search space that

[6]In applying the IDP/*UPC* framework to more complex, real-world domains, we will use an augmented version of a

corresponds to grammar G' of Fig. 15. In Fig. 16, states correspond to the elements of the grammar's alphabet, arcs between states correspond to operator applications, and the shaded areas represent those states that are only represented implicitly in a convergent search space. When an operator is applied to a state, for example, rule (or search operator) 3 is applied to state "f" to generate state "C," there is an implicit merging of states "f" and "g." The result is as if op_3 were applied to a state representing both "f" and "g." Represented graphically, it appears that search paths from "f" and "g" meet or converge at state "C."

Implicit states in a convergent search space, those shown as shaded states in Fig.16, are good intuitive examples of the information captured in abstract states. There is, however, a distinction between an *implicit state* and an *abstract state*. Implicit states are collections of states to which an operator appears to be applied. i.e., an implicit state is one that contains exactly the elements of the RHS of some rule of the grammar. For example, the state "f,g" or the state "x,y,z." Abstract states are not restricted to including only sets of base space states that appear in the RHS of some grammar rule. For example, the state "f,h" is an abstract state since f and h do not appear as the RHS of any rule of the grammar. Both implicit and abstract states contain information that can be exploited during problem solving.

Consideration of implicit and abstract states is also useful for understanding potential. For example, consider if there was a meta-operator that generated a state representing an abstraction of the information contained in an implicit state. In certain situations, this could be very useful information. Such a situation might be one where the problem solver was trying to extend a state h. Based on a local perspective of the problem solving situation that only included information related to state h, the problem solver would not be certain whether to choose an operator leading to an interpretation of an A or one that would lead to an interpretation of a B. However, if the problem solver could generate an abstract state that would represent whether or not an f was present, and if the problem solver could map this information back to state h, it would know exactly what to do and it could ignore alternative actions. (If an f is present, take a path leading to the generation of an A, if an f is not present, take a path leading to the generation of a B.) The degree to which this strategy reduces the cost of problem solving can be thought of as the potential of the abstract state.

Convergent search spaces are associated with a set of properties often referred to as *opportunistic*. Opportunistic processing was first introduced in work on the Hearsay-II project [12]. The object of opportunistic processing is to enable a system to be guided by the most recently discovered results and not by a requirement to satisfy specific subgoals. In opportunistic processing systems, many redundant paths can lead to the same result and partial, intermediate results can be merged into more comprehensive results. For example, two or more subgoals could be part of the same higher-level goal and the completion of each of the tasks associated with the subgoals could independently trigger actions that would lead, redundantly, to the invocation of tasks to generate identical solutions for the higher-level goal. This can happen when a system does not fully understand the implications of a potential action. The system may not be able to determine that two or more potential actions will generate the same result without actually invoking them. The flexibility of opportunistic processing enables a system to effectively work around areas of a problem where the required data is of poor quality

grammar that makes use of the *feature list convention* discussed by Gazdar, et al.,[15]. This will increase the expressiveness of the grammar and enable it to model real world events more accurately, but it will not invalidate the analysis techniques that will be discussed later in this paper. For the sake of clarity and simplicity, the feature list convention will not be represented in the example grammars used in this paper unless it is explicitly stated. Our use of the feature list convention to generate problem instances is discussed further in Section 6.2.

Interpretation Grammar *G*'

0. S → A | B
1. A → CD
2. B → DEW
3. C → fg
4. E → jk
5. D → hi
6. W → xyz
7. f → (signal data)
8. j → (signal data)
9. g → (signal data)
10. k → (signal data)
11. h → (signal data)
12. x → (signal data)
13. i → (signal data)
14. y → (signal data)
15. z → (signal data)

Figure 15: Interpretation Search Operators Shown as a Set of Production Rules

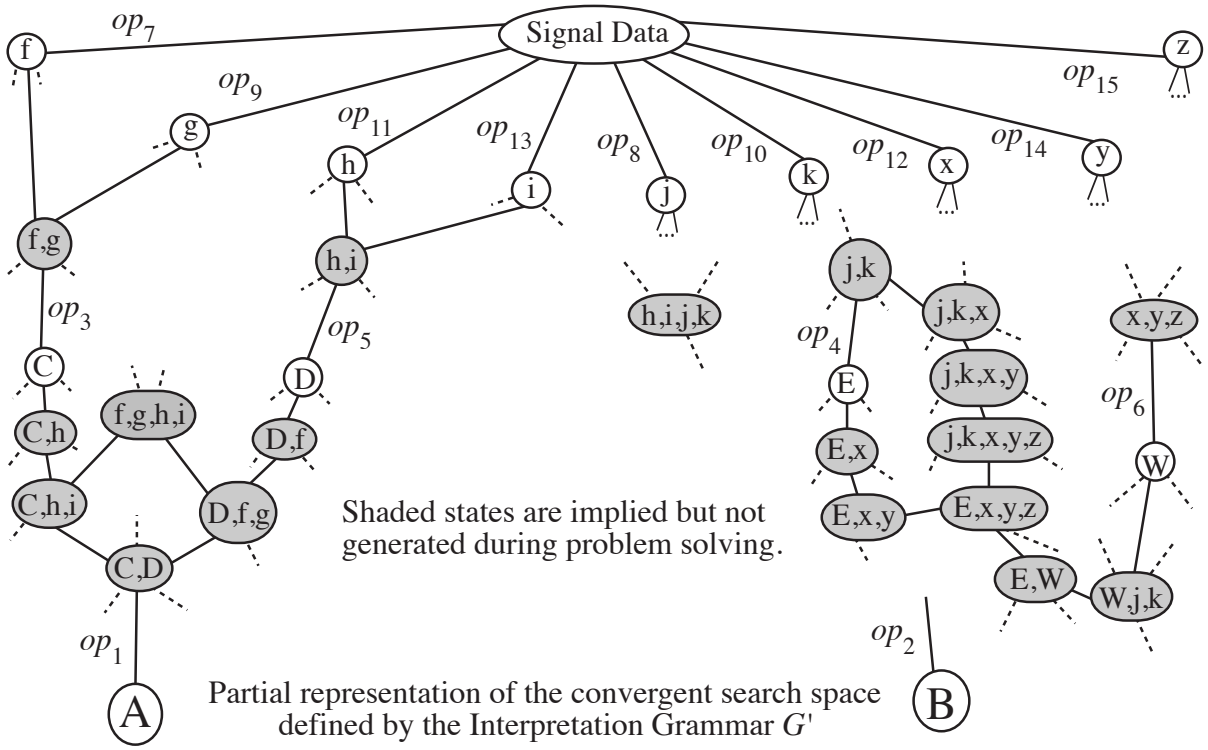


Figure 16: Convergent Search Space Defined by Interpretation Grammar

or missing or where little constraint is generated to guide subsequent processing. After other aspects of a potential solution have been determined, they can be used to constrain search in the bypassed areas. In many ways, opportunistic processing can be thought of as an extension of bidirectional search. Opportunistic search enables problem solving to proceed bottom-up from low-level data, top-down from general expectations and abstractions, or either top-down or bottom-up from intermediate results. The advantages of opportunistic processing have been demonstrated in a variety of projects including [12, 6, 10]

The interpretation search spaces presented in this paper are all convergent search spaces. The general structures that define these spaces are defined in Section 4.2. More domain specific structures are defined in Section 5.

4.2 Defining Problem Structures

Viewing a problem solver's search operators as a formal grammar is the basis for analyzing the *component*, *cost* and *utility* [7] structures of the domain theory of an interpretation problem. Component, cost, and utility, or credibility, structures are illustrated for interpretation grammar G' in Fig. 17. In general, each rule of the grammar is considered to be an arbitrarily complex problem solving operator with an associated credibility and cost function. For nonterminals with multiple right hand sides (RHSs), each RHS corresponds to a unique problem solving operator. In following sections, it will be shown that abstractions and approximations used in meta-level control actions can also be viewed from this same perspective, i.e., as operators specified by production rules of a domain's characteristic grammar.

The component structure of a problem is modeled directly by the rules of the grammar. Thus, the component subproblems of A are defined by the RHS of a rule of the grammar, $p : A \rightarrow$(component subproblems). The full structure is specified recursively. Represented graphically, the component structure of an interpretation problem appears as a derivation tree, as shown in Fig. 17.

The credibility structure is derived from the credibility functions corresponding to each of the production rules in the grammar. In Fig. 17, these functions are shown attached to the derivation tree states resulting from the inverse application of the production rule. For production rule (or state) *p*, the credibility function is represented f_p. Each credibility function is shown as a recursive function of a state's descendants. Thus, f_p is a function of *p*'s descendants. In interpretation tasks, credibility functions typically include a consideration of the credibilities of the component elements and a consideration of semantics. For example, the credibility of "A" is a function of the credibilities of "C" and "D" and the results of a *semantic function*, Γ, that measures the degree to which "C" and "D" are semantically consistent. In a natural language interpretation system, a semantic function might determine the degree to which the combination of a noun phrase and a verb phrase is meaningful. If the result is nonsensical, the semantic function, Γ, will return a relatively low value. If the result is meaningful and consistent, Γ will return a relatively high value. Similarly, in an acoustic vehicle tracking system, semantic functions might check for consistent harmonics, signal energies, etc. In this paper, we will represent credibility functions that take into consideration both the credibility and the semantic consistency of a state's descendants as, for nonterminal A with descendants C and D, $f_{\mathrm{A}}(f_{\mathrm{C}}, f_{\mathrm{D}}, \Gamma_p(\mathrm{C}, \mathrm{D}))$, where the subscript of Γ, p, is the number of the corresponding production rule from the grammar.

Similarly, the cost structure is defined in terms of the amount of a resource, such as time, required

[7] In the interpretation problems discussed in this paper, utility and *credibility* are synonymous.

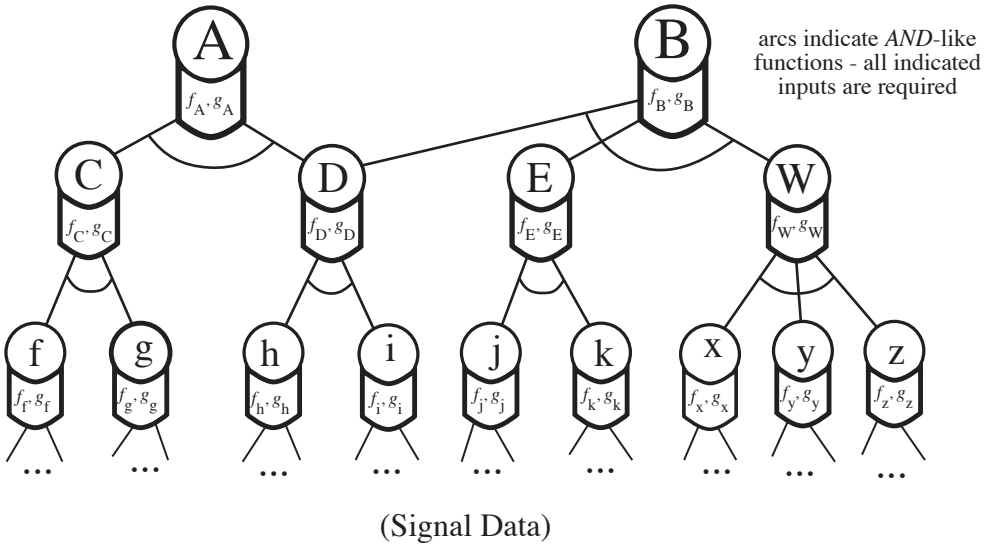


Figure 17: Derivation of Utility and Cost Structure From Interpretation Grammar

to inversely apply a production rule. For state *n*, the cost function is represented g_n. It is important to note that g_n defines the cost of generating the entire tree or subtree associated with state n, not just the cost of applying operator n.

This definition does not take into consideration the characteristics of the state of the problem solver at the time an operator is executed. For example, a component of an operator's cost may be a function of the amount of data currently stored in a database. Considerations such as these will not be dealt with in this paper. However, there are a variety of techniques for dealing with them. One such technique would be to simulate an operator being applied in a series of different contexts and to compile statistical information about the cost of applying the operator and to use this information to determine the expected cost and variance of applying an operator. (In fact, this is very similar to what we do. However, we do not actually conduct statistical studies of operator cost. We merely specify an expected cost and variance.) An alternative technique would be to incorporate a representation of significant aspects of the problem solver's internal state within each search state. This would enable our analysis techniques to differentiate search paths based on the order in which operators are applied and would enable analysis of problem solving costs with respect to the internal state of the problem solver.

In interpretation tasks, cost is typically a function of the cost of the component elements and the cost of the semantic function. For example, the cost of "A" is a function of the cost of "C" and "D" and the cost of the semantic function $\Gamma_p(\mathrm{C}, \mathrm{D})$. We will represent cost functions as, for nonterminal A with descendants C and D, $g_{\mathrm{A}}(g_{\mathrm{C}}, g_{\mathrm{D}}, cost(\Gamma_p(\mathrm{C}, \mathrm{D})))$, where the subscript of Γ, p, is the number of the corresponding production rule from the grammar and $cost(\Gamma_p(i, j, ...))$ is the cost of applying the semantic function Γ_P. Where it does not cause confusion, $cost(\Gamma_p(i, j, ...))$ will be also represented as $C(\Gamma_p(i, j, ...))$.

A typical credibility function might be "average," which determines the credibility of a state by normalizing the sum of the credibilities of the state's descendants and the result of the semantic function. Alternatives include taking the minimum value or reducing the average descendant credibility by a function of the output of the semantic function. A typical cost function might be "sum," which sums the cost of generating a state's descendants and the cost of the semantic function.

Interpretation Grammar G'

grammar rule	distribution	credibility	cost
0.1 S → A	$\psi(0.1) = 0.2$	$f_{0.1}(f_A)$	$g_{0.1}(g_A)$
0.2 S → B	$\psi(0.2) = 0.2$	$f_{0.2}(f_B)$	$g_{0.2}(g_B)$
0.3 S → M	$\psi(0.3) = 0.2$	$f_{0.3}(f_M)$	$g_{0.3}(g_M)$
0.4 S → N	$\psi(0.4) = 0.2$	$f_{0.4}(f_N)$	$g_{0.4}(g_N)$
0.5 S → O	$\psi(0.5) = 0.2$	$f_{0.5}(f_O)$	$g_{0.5}(g_O)$
1. A → CD	$\psi(1) = 1$	$f_1(f_C,f_D, \Gamma_1(C,D))$	$g_1(g_C,g_D,C(\Gamma_1(C,D)))$
2. B → DEW	$\psi(2) = 1$	$f_2(f_D,f_E,f_W, \Gamma_2(D,E,W))$	$g_2(g_D,g_E,g_W,C(\Gamma_2(D,E,W)))$
3.0 C → fg	$\psi(3.0) = 0.5$	$f_{3.0}(f_f,f_g, \Gamma_{3.0}(f,g))$	$g_{3.0}(g_f, g_g,C(\Gamma_{3.0}(f,g)))$
3.1. C → fgq	$\psi(3.1) = 0.5$	$f_{3.1}(f_f,f_g,f_q, \Gamma_{3.1}(f,g,q))$	$g_{3.1}(g_f,g_g,g_q,C(\Gamma_{3.1}(f,g,q)))$
4. E → jk	$\psi(4) = 1$	$f_4(f_j,f_k, \Gamma_4(j,k))$	$g_4(g_j,g_k,C(\Gamma_4(j,k)))$
5.0 D → hi	$\psi(5.0) = 0.5$	$f_{5.0}(f_h,f_i, \Gamma_{5.0}(h,i))$	$g_{5.0}(g_h,g_i,C(\Gamma_{5.0}(h,i)))$
5.1. D → rhi	$\psi(5.1) = 0.5$	$f_{5.1}(f_r,f_h,f_i, \Gamma_{5.1}(r,h,i))$	$g_{5.1}(g_r,g_h,g_i,C(\Gamma_{5.1}(r,h,i)))$
6.0 W → xyz	$\psi(6.0) = 0.5$	$f_{6.0}(f_x,f_y,f_z, \Gamma_{6.0}(x,y,z))$	$g_{6.0}(g_x,g_y,g_z,C(\Gamma_{6.0}(x,y,z)))$
6.1. W → xy	$\psi(6.1) = 0.5$	$f_{6.1}(f_x,f_y, \Gamma_{6.1}(x,y))$	$g_{6.1}(g_x,g_y,C(\Gamma_{6.1}(x,y)))$
7. f → (s)	$\psi(7) = 1$	$f_7(f_{(s)}, \Gamma_7(\ (s)\ ))$	$g_7(g_{(s)},C(\Gamma_7(\ (s)\ )))$
8. j → (s)	$\psi(8) = 1$	$f_8(f_{(s)}, \Gamma_8(\ (s)\ ))$	$g_8(g_{(s)},C(\Gamma_8(\ (s)\ )))$
9. g → (s)	$\psi(9) = 1$	$f_9(f_{(s)}, \Gamma_9(\ (s)\ ))$	$g_9(g_{(s)},C(\Gamma_9(\ (s)\ )))$
10. k → (s)	$\psi(10) = 1$	$f_{10}(f_{(s)}, \Gamma_{10}(\ (s)\ ))$	$g_{10}(g_{(s)},C(\Gamma_{10}(\ (s)\ )))$
11. h → (s)	$\psi(11) = 1$	$f_{11}(f_{(s)}, \Gamma_{11}(\ (s)\ ))$	$g_{11}(g_{(s)},C(\Gamma_{11}(\ (s)\ )))$
12. x → (s)	$\psi(12) = 1$	$f_{12}(f_{(s)}, \Gamma_{12}(\ (s)\ ))$	$g_{12}(g_{(s)},C(\Gamma_{12}(\ (s)\ )))$
13. i → (s)	$\psi(13) = 1$	$f_{13}(f_{(s)}, \Gamma_{13}(\ (s)\ ))$	$g_{13}(g_{(s)},C(\Gamma_{13}(\ (s)\ )))$
14. y → (s)	$\psi(14) = 1$	$f_{14}(f_{(s)}, \Gamma_{14}(\ (s)\ ))$	$g_{14}(g_{(s)},C(\Gamma_{14}(\ (s)\ )))$
15. z → (s)	$\psi(15) = 1$	$f_{15}(f_{(s)}, \Gamma_{15}(\ (s)\ ))$	$g_{15}(g_{(s)},C(\Gamma_{15}(\ (s)\ )))$
16. M → Y	$\psi(16) = 1$	$f_{16}(f_Y)$	$g_{16}(g_Y)$
17.0 Y → qr	$\psi(17.0) = 0.5$	$f_{17.0}(f_q,f_r, \Gamma_{17.0}(q,r))$	$g_{17.0}(g_q,g_r,C(\Gamma_{17.0}(q,r)))$
17.1 Y → qhri	$\psi(17.1) = 0.5$	$f_{17.1}(f_q,f_h,f_r,f_i, \Gamma_{17.1}(q,h,r,i))$	$g_{17.1}(g_q,g_h,g_r,g_i,C(\Gamma_{17.1}(q,h,r,i)))$
18. N → Z	$\psi(18) = 1$	$f_{18}(f_Z)$	$g_{18}(g_Z)$
19. Z → xy	$\psi(19) = 1$	$f_{19}(f_x,f_y, \Gamma_{19}(x,y))$	$g_{19}(g_x,g_y,C(\Gamma_{19}(x,y)))$
20. O → X	$\psi(20) = 1$	$f_{20}(f_X)$	$g_{20}(g_X)$
21.0. X → fgh	$\psi(21.0) = 0.5$	$f_{21.0}(f_f,f_g,f_h, \Gamma_{21.0}(f,g,h))$	$g_{21.0}(g_f,g_g,g_h,C(\Gamma_{21.0}(f,g,h)))$
21.1. X → fg	$\psi(21.1) = 0.5$	$f_{21.1}(f_f,f_g, \Gamma_{21.1}(f,g))$	$g_{21.1}(g_f,g_g,C(\Gamma_{21.1}(f,g)))$

(s) = signal data $\Gamma_n(i,j,...)$ = semantic evaluation function for rule n $C(\Gamma_n(i,j,...))$ = cost of executing $\Gamma_n(i,j,...)$

Figure 18: Example of Interpretation Grammar with Fully Specified Distribution, Credibility, and Cost Functions

For a given instance of an IDP, I, the problem structure of the domain will now be defined in terms of G_I, the characteristic grammar for I's domain, and the functions f_p and g_p. Formally,

Definition 4.1 *An IDP Grammar is a grammar,* $G_I =< V, N, SNT, S, P >$, where V is the set of terminal symbols, N is the set of nonterminal symbols, SNT is the set of *solution-nonterminal* symbols that correspond to final states, S is the start symbol for the grammar, and P is the set of context-free production rules.

This grammar specification is very similar to traditional grammar specifications with the notable exception of the set of solution nonterminals. This specification is necessary for analytical reasons that will be explained in subsequent sections. f_p and g_p are defined for elements $p \in P$. In addition, another function, ψ, will also be used to define problem structures by augmenting G_I's expressive power. Specifically, for each rule $p \in P$, there is a corresponding $\psi(p)$. $\psi(p)$ will be used to represent the distribution of "right hand sides (RHSs)" associated with p. Each of these alternative RHSs will be thought of as a distinct problem solving operator that is associated with a distinct credibility and cost function. This is illustrated in Fig. 18. ψ is introduced here to support the definition of problem structures in subsequent sections. ψ will be used to support the modeling of real-world phenomena such as uncertainty caused by noise, missing data, distortion and masking (see Section 5). ψ will be represented by saying that, for production p which decomposes to RHS_1, there are one or more corresponding p' that decompose to alternative RHSs, $\text{RHS}_2 \ldots \text{RHS}_m$. The definition of ψ is shown in Fig. 19. In Fig. 19, production rule p has a number of possible RHSs. These different RHSs can be thought of as the right hand sides of variations of p numbered $p.1$ through $p.\text{m}$ and, for a given p, $\sum_n \psi(p.n) = 1$. Figure 18 shows an example of a simple interpretation grammar with fully specified distribution, credibility, and cost functions. For some of the rules, e.g., 0.1, 0.2, 20, etc., there are no semantic functions. This is reflected in the credibility and cost functions that only take into consideration the credibility and cost of descendant states.

In general, there are no restrictions on the production rules of G_I. In practice, however, it will be necessary to limit any recursive rules to a specific number of iterations in order to perform numerical analyses. This will not limit the applicability of the analysis tools we derive in any way. This is because real-world interpretation systems must function with similar restrictions in that each problem solving instance must be of finite length. For systems that are intended to interpret streams of continuous data, the data is divided into "time slices" of finite length. In these systems, our analysis techniques would be applied only to the individual time slices. The techniques described in this paper will have to be extended to address issues associted with problem solving systems that process data from multiple time slices simultaneously.

It is important to note that multiple final states may correspond to each element of SNT. For example, in a natural language domain, SNT may contain a single element, "sentence." In a typical problem solving instance, there will be multiple (often numerous) different sentences generated. Similarly, in a vehicle tracking domain, a typical element of SNT might be "vehicle track of type 1" and any given problem solving instance may generate multiple, different interpretations of type 1 vehicle tracks. The alternative instantiations of an SNT are differentiated by the characteristics of the specific instances. For example, the alternative type 1 vehicle tracks may pass through different locations.

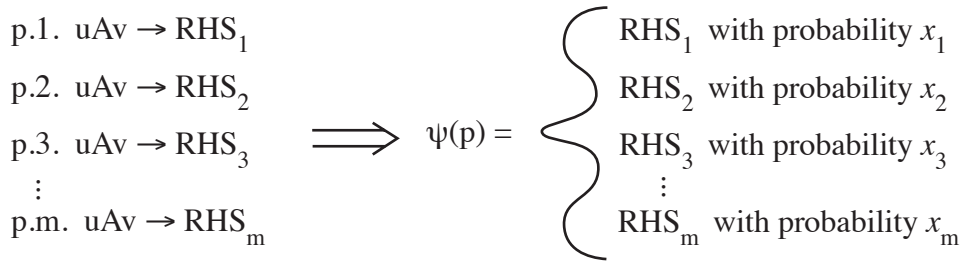


Figure 19: Example of the Distribution Function ψ

4.3 Structural Interaction

In the preceding subsection, we defined the component, credibility, and cost structures of a domain as independent entities. In fact, in many domains, these structures interact. Such interactions can lead to very complex problem solving behaviors and to corresponding analysis techniques that are also very complex. In the examples in this paper, we will present and discuss analysis techniques that are appropriate for various structural interactions. However, we will often use approximations of these techniques for computability reasons. In the remainder of this subsection, we will define and give examples of structural interactions.

In Section 9 we will formalize the computation of certain values, which we will refer to as *UPC* values, based on the component, credibility, and cost stuctures of an IDP grammar. In general, there will be a set of values associated with each of the basic IDP structures, a set of probability values associated with the component structure, a set of credibility (or utility) values associated with the credibility structure, and a set of cost values associated with the cost structure. In certain domains, a given set of *UPC* values may be associated with more than a single IDP structure. We refer to this phenomenon as *structural interaction*. Thus, structural interaction occurs when the *UPC* values for an IDP domain are computed based on the interaction of two or more basic structures. For example, when the probability values are computed based on the interaction of the component and credibility structures.

To understand structural interactions, consider the IDP domain structure shown in Fig. 20. This hypergraph is very similar to the one shown in Fig. 17, with the notable exception of the 'prime' states. In this example, the prime states correspond to dynamic pruning operators. (Dynamic pruning operators are discussed at great length in subsequent sections.) A dynamic pruning operator of the form $s' \rightarrow s$ has a corresponding interpretion operator which functions as follows: "given an s, generate an s' if the credibility rating of s is above a certain threshold, t." This interpretation constitutes a structural interaction because the probability of generating an s' is a function not only of the component structure of a domain, but the credibility structure as well. Specifically, the probability of generating an s' given an s is a combination of the conditional probability function, P(s' | s), which is determined from the component structure, and the distribution of the credibility functions of the grammar. Thus, the component and credibility structures interact.

For example, consider a domain where the pruning threshold is 0 (i.e., no pruning takes place), and the IDP grammar rules are those shown in Fig. 20. In this domain, the probability of generating an x' given an x is 1. This is determined from the conditional probability P(x'| x) which is computed from

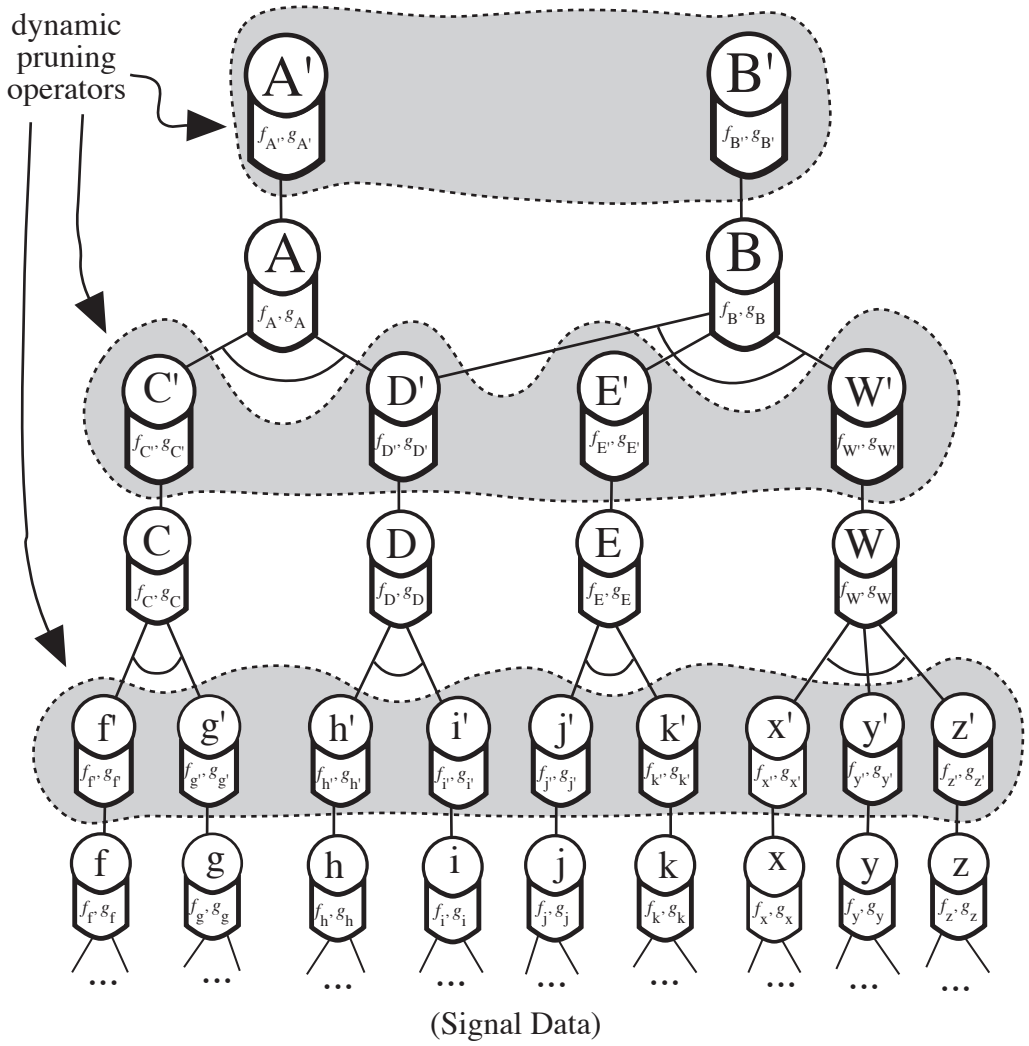


Figure 20: Example of the Structural Interaction

the component structure of the grammar. Now consider the same situation with a pruning threshold of t. In this new situation, the probability of generating an x' given an x is 1 (the conditional probability of generating an x' given an x) multiplied by the probability that the credibility of x is greater than, or equal to, t, or 1 - P(credibility(x) < t).

It is important to note that in domains with structural interactions, the computation of *UPC* values must be done dynamically. This is because the *UPC* values for a state are a function of the characteristics of a state, such as credibility, that are unknown until the state is actually created. Therefore, the *UPC* values cannot be computed a priori. However, it should be noted that certain computations can be done a priori, as will be discussed in later sections. These computations can be stored and retrieved at run-time to reduce the cost of computing *UPC* values dynamically.

4.4 Interpretation Problem Solving and Formal Problem Solving Paradigms

As discussed by Kanal and Kumar in [25], formal problem solving paradigms that are based on search techniques can be divided into two general categories, either *top-down* or *bottom-up*. The IDP/*UPC* framework is consistent with this and supports the analysis of control architectures from either a top-down or a bottom-up perspective. This is important because we hope to develop a unified perspective of problem solving by extending Kanal and Kumar's taxonomy to include the sophisticated control architectures that can be described with the IDP/*UPC* framework.

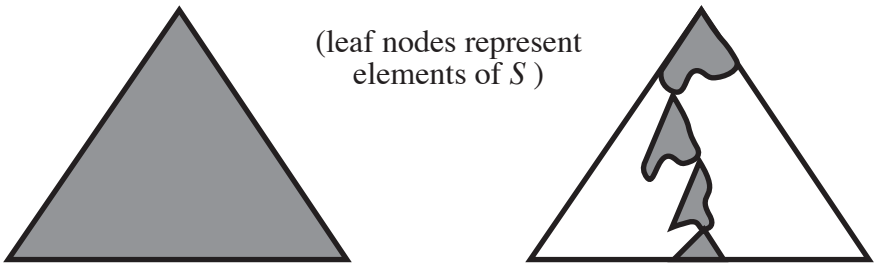
Using bottom-up control architectures, rules of the grammar are inversely applied and more comprehensive derivation trees are generated and represented as new states. Thus, larger problems are solved starting with their smaller components. This approach is used in many search procedures and is the basis for dynamic programming [33, 34, 25]. The examples in Section 4.1 are representative of bottom-up problem solving in interpretation problems.

Using top-down control architectures, some representation of the total set of interpretations is repeatedly partitioned and pruned. After each partition, all members of the partition are deleted for which it can be shown that, even after elimination, the most credible interpretation is an element of one of the remaining partitions. This technique has been used extensively in Operations Research, where it is referred to as Branch-and-Bound [33, 34, 25]. Top-down and bottom-up architectures have been effectively combined in a number of systems such as blackboard systems [12, 6]. In these systems, the top-down strategies, such as goal processing techniques, can be considered meta-control architectures because they use abstract or approximate states.

Using either or both of these approaches, problem solving continues until *every possible* derivation tree is either generated or eliminated from consideration based on the structure of the problem. Once all derivation trees have been determined, the problem solver identifies which has the highest rating and returns that derivation tree as the interpretation.

This definition is important because it implies that the problem solver cannot simply compute one derivation tree and stop. In order to find the "best" interpretation, *every* possible search path must be explored in some way. Thus, for any given state, at termination, every operator that can be applied to it has been accounted for in some way. Such a state will be referred to as a *connected state* and a space consisting solely of connected states will be referred to as a *connected space*. An *open state* will be a state that is not connected. The set I of interpretations from Definition 4.1 corresponds to a connected space. Implicit in this definition is the requirement that a connected space "account for" or "explain" all the input data. This requirement is derived from the requirement that each individual solution that is considered as an overall solution to a specific problem must explain all the detected input data. The distinction of "detected data" is important because the analysis techniques only work for the problem solving actions represented as search processes. They do not necessarily explain all the data input to the non-search processes such as low-level data filtering operators.

Clearly, depending on a domain's characteristic grammar and associated functions, interpretation problem solving can be a formidable task. The total number of solutions generated for an arbitrary set I can be enormous. In general, the role of sophisticated control is to limit the size of I by implicitly enumerating as much of the search space as possible. This objective is illustrated in Fig. 21, from Berliner [1]. Figure 21.a represents the set I generated by grammar G without the use of sophisticated control techniques. In this example, problem solving is essentially exhaustive – every possible derivation tree is generated and compared. Figure 21.b shows the effects of sophisticated control. Here, large portions of the search space are eliminated from consideration based on the problem solver's understanding of



a. Without the use of sophisticated control, search spaces must be enumerated exhaustively.

b. Using sophisticated control, many areas of a search space can be eliminated from consideration implicitly, thereby reducing the cost of problem solving.

Figure 21: Implicit Enumeration – the Role of Control

the problem's structure.

Finally, it is important to emphasize that G_I, f_p, g_p, and ψ are used to specify the set I from which the best interpretation will be identified.

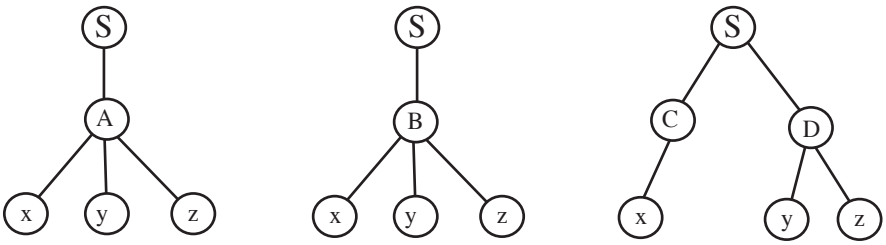
5 Defining IDP Structures

Section 4 introduced the IDP formalism for modeling domain theory problem structures. The significance of these structures is that they are defined in terms of the operators available to solve the problem. Conversely, problem solving operators are defined in terms of the structure of a problem's domain theory.

This section will demonstrate how the IDP formalism can define a number of important domain theory problem structures. As noted above, these structures also define the operators and control actions available to solve a problem. Subsequent sections will define another formalism, the *UPC* formalism, that will be used in conjunction with the IDP formalism as a general model of control and problem solving activities. Using the *UPC* formalism, the cost and utility of control and problem solving actions will be compared directly. The sections describing the *UPC* model will draw heavily on the structures defined in this section.

The power of the IDP formalism lies in its ability to model the structure of a problem domain. The key to exploiting the IDP formalism is based on the degree to which the structure of a problem formulation maps into and is represented by the IDP's grammar, credibility functions, cost functions, and RHS distribution functions.

For this paper, analysis will focus on five primary aspects of problem structure: uncertainty, operator organization, redundancy, interacting subproblems, and bounding functions. These aspects of the structure of interpretation problems are introduced in this section. Two of these structural aspects, uncertainty and redundancy, are inherent components of a domain. They are derived naturally by representing a problem domain as an IDP model. The other aspects can be thought of as transformations of the base IDP model. These transformations are used to model control actions in terms of a unified perspective with problem solving actions. Typically, these transformations are used to exploit structures present in the base IDP model.



a. An ambiguous grammar. There are three parse trees for the string xyz.

Figure 22: An Example of Ambiguity

Undoubtedly, many other structures are manifested in a given problem domain. However, it is far beyond the scope of this work to attempt to classify, or even identify all possible aspects of problem space structure.

5.1 Inherent Uncertainty

One of the more difficult aspects of an interpretation task is dealing effectively with *inherent uncertainty*. In terms of the IDP formalism, inherent uncertainty can be thought of as ambiguity in the interpretation grammar. Ambiguity increases the complexity of an interpretation problem by making a large number of partial solutions and full, potential solutions seem plausible. This implies that additional operator applications must be used to generate and differentiate the ambiguous interpretations, as will be discussed in the next section. Thus, inherent uncertainty structures are not used to model control actions, rather, they are structured phenomena that control actions are intended to exploit to improve the efficiency of problem solving. Using the IDP model, inherent uncertainty will be classified into the taxonomy presented in the following subsections. The definitions of the classes of inherent uncertainty will be associated with the rules of the grammar and with a formal definition of ambiguity.

For a rule of the grammar, p, variations of the rule will be used to help define certain forms of uncertainty. This will be represented by associating multiple RHSs to p. The distribution of the domain events that determine which of the RHSs is appropriate will be modeled by $\psi(p)$. This is an important point that will be emphasized in Section 9. Finally, each of the RHSs will be thought of as a unique primitive operator that the problem solver can use to build a partial or full interpretation.

The formal definition of ambiguity that will be used to define the taxonomy of inherent uncertainty structures is presented in the next subsection.

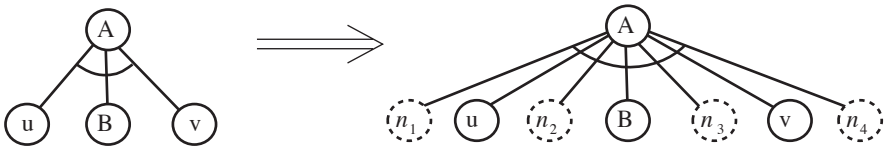
5.1.1 Ambiguity

Ambiguity is a property of a grammar that leads to multiple interpretations for a given input. In an ambiguous IDP, the only way to differentiate a correct interpretation from an incorrect interpretation is to compare their credibility values. More formally:

Definition 5.1 *Ambiguity* - An instance of an IDP is *ambiguous* iff two (or more) interpretation trees exist for some input, X. (A more precise definition of ambiguity will be given in Definition 9.14.)

An example of an ambiguous grammar is shown in Fig. 22 where three interpretation trees exist for the terminal string xyz.

IDPs can also be *semi-ambiguous*.



b. The production rule p: A -> uBv and the corresponding `noisey' rule p': A -> n_1un_2Bn_3vn_4.

Figure 23: An Example of a Noisy Grammar Rule

Definition 5.2 *Semi-Ambiguity* - An instance of an IDP is semi-ambiguous iff the IDP is not ambiguous and two (or more) interpretation subtrees exist for some portion, Y, of an input string, X.

In a semi-ambiguous IDP, it is not possible to differentiate alternative interpretation subtrees, $(y_1, \ldots y_n)$, that are components of the best interpretation ("correct" subtrees) from subtrees that are not components of the best interpretation ("incorrect" subtrees) simply by comparing their credibility levels and choosing the subtree with the higher rating. This is because IDP domains are non-monotone. (Non-monotonicity is formally defined in Subsection 5.5.) The only way to determine which of the subtrees, $(y_1, \ldots y_n)$, is the correct subtree is to continue the interpretation, extending each of the competing alternatives. One of two things will happen, either the problem solver will be unable to extend a partial interpretation or the problem solver will find a solution. If the problem solver fails to extend all partial interpretations derived from a given subtree, then that subtree is incorrect. When the problem solver identifies the best interpretation, all the subtrees associated with that interpretation are considered correct partial interpretations.[8]

Intuitively, the cost of ambiguous IDPs is the sum of the costs of generating each of the derivation trees corresponding to ambiguous interpretations and the subtrees, $(y_1, \ldots y_n)$, corresponding to partial interpretations, plus the cost of differentiating the correct interpretation (and, by implication, the correct derivation subtrees). In general, as the number of ambiguous interpretations and partial interpretations increases in size, the cost of problem solving will increase.

5.1.2 Noise

Intuitively, noise can be thought of as domain data that is generated in a spurious fashion and that is not necessary or sufficient to infer the associated higher-level partial or full interpretation. Noise does not necessarily correspond exclusively to unknown phenomena. For example, in a vehicle tracking domain, noise may correspond to relatively normal domain events, such as sounds associated with weather phenomena or animals. Formally,

Definition 5.3 *Noise* - A grammar, G, is subject to noise iff
$\exists$ a rule p $\in$ P, where p: A $\rightarrow$ uBv $\Rightarrow$ p$'$: A $\rightarrow$ n_1un_2Bn_3vn_4, for u,v $\in$ (V $\cup$ N)*, A $\in$ (SNT $\cup$ N), B $\in$ (V $\cup$ N), $n_i \in$ (V $\cup$ N)*, and $f_p(f_u, f_B, f_v, \Gamma_p(u, B, v)) > f_{p'}(f_{n_1}, f_u, f_{n_2}, f_B, f_{n_3}, f_v, f_{n_4}, \Gamma_{p'}(f_{n_1}, u, f_{n_2}, B, f_{n_3}, v, f_{n_4}))$ for maximum ratings of the f_{n_i}. The distribution of p and p' is modeled by ψ.

[8] It is important to note that virtually all interpretation problems are at least semi-ambiguous. An unambiguous interpretation problem might be more properly categorized as a parsing problem or a classification problem.

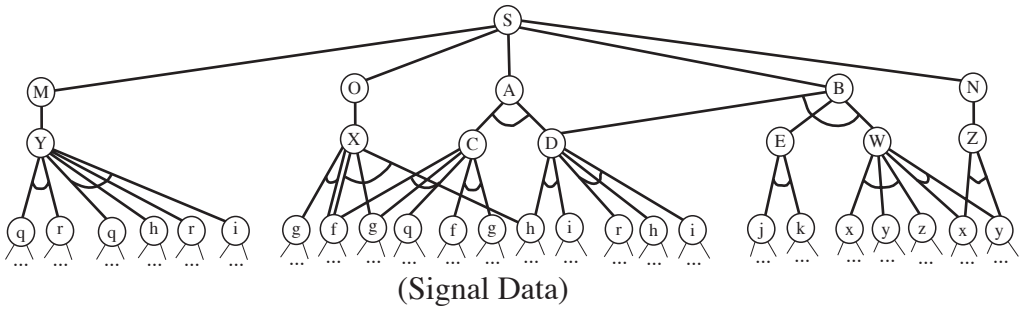


Figure 24: Interpretation Grammar G' with Added Noise and Missing Data Rules

In other words, for some rule p in G, there are at least two possible RHSs. The frequency with which noise appears in a domain is modeled by ψ. Figure 23 is a representation of the definition of noise. Each of the states labelled n_i could lead to arbitrary "noisy" subtrees[9]. In many domains, each production rule may have numerous alternative noisy productions. Again, the determination of which of these productions is used is made by the function ψ. It is important to emphasize that, by definition, a noisy rule has a credibility that is lower than the corresponding non-noisy rule. Given a set of RHSs that are candidates for being returned by ψ, the non-noisy RHS is the one with the highest credibility. (In situations where a noisy rule has higher credibility than a non-noisy rule, we would consider the noisy rule to be the "correct" rule and the non-noisy rule to be a missing data rule. Missing data rules are defined below.)

Noise is problematic for an interpretation problem because it can increase the amount of ambiguity with which the problem solver must contend. This increases the cost of problem solving. An increase in ambiguity implies that the problem solver must apply additional operators, which increases cost, to build and differentiate alternative interpretations. Consider a grammar in which no two rules have identical right hand sides (RHSs). (Note that this IDP may or may not be ambiguous.) Now assume that noise is introduced in such a way that at least one of the "noisy" rules has an RHS that is identical to the RHS of an existing, non-noisy rule. An example of such a situation is shown by Figs. 24 and 25. By definition, the IDP represented by these figures is at least semi-ambiguous.

Given a problem instance in which the noisy rule is the correct interpretation for the signal data, the problem solver will have to differentiate the interpretation in which the noisy rule is used from interpretations corresponding to "non-noisy" interpretations of the data. This can be very difficult because the only way to differentiate a correct interpretation tree from an incorrect one is by comparing their credibility ratings and, as was discussed previously, this technique does not work, in general, for partial trees. From the local perspective of competing alternative partial interpretation trees, relative credibility ratings may correlate to the probability of a given subtree being correct. However, this correlation may be significantly less than one. Noisy production rules often have low credibility ratings and correct subtrees that contain noisy production rules frequently have lower credibility ratings than incorrect alternative subtrees. Consequently, the problem solver may not be able to differentiate competing alternative interpretations without additional, perhaps expensive, processing. It may be the

[9]Please note that in the figures in this paper, and in the IDP/*UPC* Framework, there is no meaning associated with the order in which terminal or nonterminal symbols are represented. Thus, the sequence of terminal symbols "r h i" is identical the the sequence "h r i."

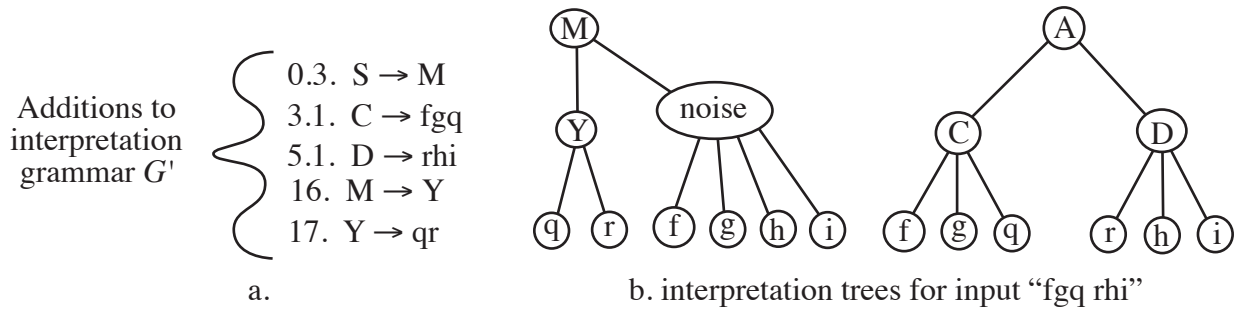


Figure 25: An Example of Correlated Noise - The noise in rules 3.1 and 5.1, q and r, is correlated to an interpretation of M.

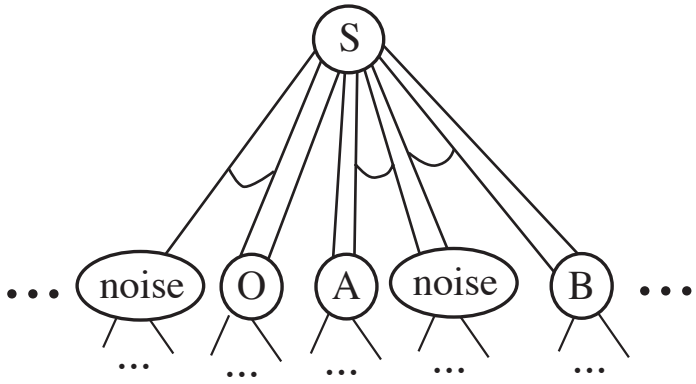


Figure 26: Implicit Rules for Interpreting Noise

case that, in order to differentiate competing alternatives in a semi-ambiguous grammar, the problem solver is forced to generate nearly complete solutions incorporating each of the competing alternatives. The need to apply additional operators to build and differentiate alternative subtrees could result in very expensive problem solving.

In Fig. 24, noise has been added to grammar G' from Fig. 15. In this grammar, SNT contains A, B, M, N, and O. Some of the noise is specified in the new rules 3.1 and 5.1 shown in Fig. 25. Also, a new interpretation, M (rule 16), has been added to the grammar. (Note that each of these rules corresponds to the addition of a new primitive operator.) As a result of adding these production rules, in certain noisy situations, signal data can be interpreted as either an "M" or an "A." Figure 25, which shows alternative derivation trees for "M" and "A" given the input "fgq rhi," depicts such a situation.

From a local perspective, the partial interpretation corresponding to state Y may be rated higher than either of the partial interpretations corresponding to states C or D. However, the full interpretation represented by state M will have a lower rating than the full interpretation represented by state A. Intuitively, this is a result of the fact that the A interpretation explains all the observed phenomena and the M interpretation only explains the data related to q and r. To account for data that is not explained by an interpretation, we assume the existence of rules such as those in Fig. 26 that account for extraneous noise data. In a typical situation, an interpretation that includes noise as shown in this figure has a much lower credibility than an interpretation that explains all of the data as being something other than random noise.

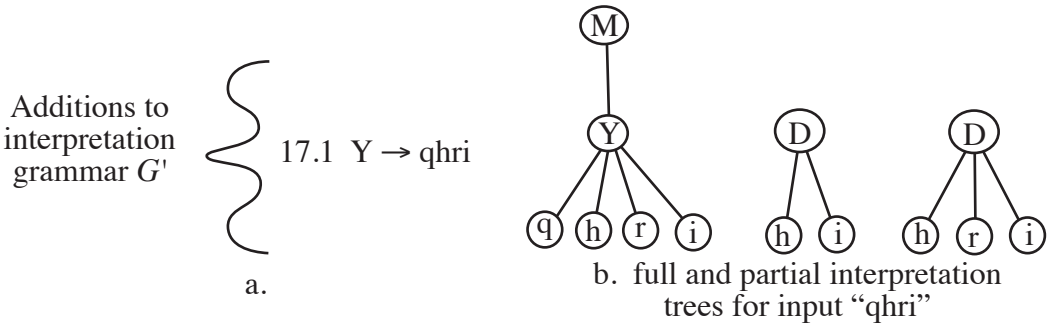


Figure 27: An Example of Uncorrelated Noise

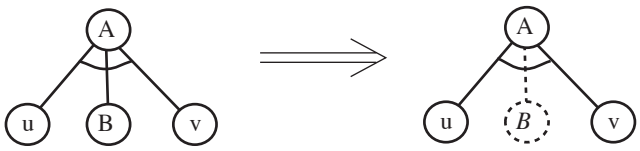
This is an example of how noise can add ambiguity to a grammar. Without the noise, when the problem solver recognizes a q or an r, it can immediately return an interpretation of M, since no other interpretation contains a q or an r. However, with the addition of noise, the problem solver must differentiate interpretations of M and A. This will be much more expensive because the problem solver will have to apply all the operators implied by each interpretation's component structure in order to generate accurate ratings. This is necessary because the problem solver will have to execute all the semantic functions to generate the credibility ratings needed to differentiate the alternative interpretations.

Figure 27 illustrates a slightly different situation that is also based on the modifications to G' shown in Fig. 24. In this example, noise is added to grammar G' in the form of rule 17.1, and this noise leads to the generation of the input "qhri." During interpretation, two partial interpretations corresponding to D are generated. However, because there is no additional data, interpretations involving the use of D fail to generate an A interpretation and the D states become connected without being used in a full interpretation. This is an example of how noise can make a grammar semi-ambiguous. Though the increase in cost may not be as severe as an ambiguous case, semi-ambiguity still increases interpretation costs because it forces the problem solver to determine which alternative partial interpretations cannot be extended to full interpretations.

These examples, and the observation that noise may lead to rules with identical RHSs and ambiguity, lead to definitions for two specific kinds of noise.

Definition 5.4 *Correlated Noise* - Noise that results in ambiguous interpretations. Correlated noise can significantly increase the amount of work or other resources required to solve an IDP if it requires the problem solver to generate multiple interpretations in order to differentiate the competing partial interpretations. Correlated noise is especially problematic when the correlation is high and the introduction of noise leads to numerous incorrect, but credible, interpretations. An example of correlated noise is shown in Fig. 25.

Definition 5.5 *Uncorrelated Noise* - Noise that does not result in ambiguous interpretations. Uncorrelated noise may lead to semi-ambiguous IDPs, but, in general, is easier to differentiate than correlated noise. With uncorrelated noise, the problem solver is not required to generate complete interpretation trees to differentiate the correct noisy partial interpretation from incorrect competing alternatives. Instead, the competing alternatives will be eliminated when the problem solver fails to extend them at some point of the interpretation process. The incorrect interpretations will not correspond to any possible interpretation and the problem solver will not be able to inversely apply any production rules



c. The production rule p: A -> uBv and the corresponding
`missing data' rule p': A -> uv.

Figure 28: An Example of a Missing Data Grammar Rule

to extend the interpretation. An example of uncorrelated noise is shown in Fig. 27. Also, as will be discussed, additional problem structures defined by *bounding functions* (see Subsection 5.5) can be used to identify incorrect partial interpretations without the derivation of a full interpretation.

The correlation of noise to potentially correct interpretations is much more complex than the dichotomy suggested by the above definitions. In fact, the correlation of noisy interpretations to correct interpretations can be thought of as a continuum in terms of cost. At one extreme is noise that is uncorrelated to correct data. This noise can be disambiguated with little cost. At the other extreme is noise that leads to numerous full interpretations and significant cost. Between these extremes are cases where the noise is uncorrelated to actual data, but where the structural constraints of the problem that enable incorrect, uncorrelated partial interpretations to be pruned are costly to apply. For example, the problem solver may expend a great deal of effort on a highly rated partial interpretation before encountering grammar constraints that prevent its further extension. Thus, the concept of the correlation of noise to potentially correct, alternative interpretations will be thought of as a continuum where low-correlation implies that incorrect interpretations resulting from noise can be disambiguated with little cost and high-correlation implies that incorrect interpretations resulting from noise will be very expensive to disambiguate.

5.1.3 Missing Data

Intuitively, missing data is the result of sensing and other domain phenomena that prevent certain data normally associated with a domain event from being detected. For example, in a vehicle monitoring domain, the microphones used to pick up signals from vehicles might be flawed so that some frequencies are missed in situations where the vehicle is moving slowly. The characteristic signals are present, but are below some threshold and are not 'heard' by the sensors. Missing data also results from inappropriately processed low-level data. For example, a filtering algorithm may be used with a poor selection of tuning parameters resulting in the loss of significant data. Formally,

Definition 5.6 *Missing Data* - A grammar, G, is subject to missing data iff
$\exists$ p $\in$ P, where p: A $\rightarrow$ uBv $\Rightarrow$ p$'$: A $\rightarrow$ uv, for u,v $\in$ (V $\cup$ N)*, A $\in$ (SNT $\cup$ N), B $\in$ (V $\cup$ N),and $f_p(f_u, f_B, f_v, \Gamma_p(u, B, v)) > f_{p'}(f_u, f_v, \Gamma_{p'}(u, v))$. In addition, the distribution of p and p' is modeled by ψ.

As with noise, missing data implies the existence of multiple RHSs for some production rule(s). The distribution of the domain events that determine which of the RHSs is appropriate is modeled by ψ. Figure 28 illustrates the definition of missing data. In many domains, each production may correspond to numerous alternative missing data productions. Again, by definition, a missing data

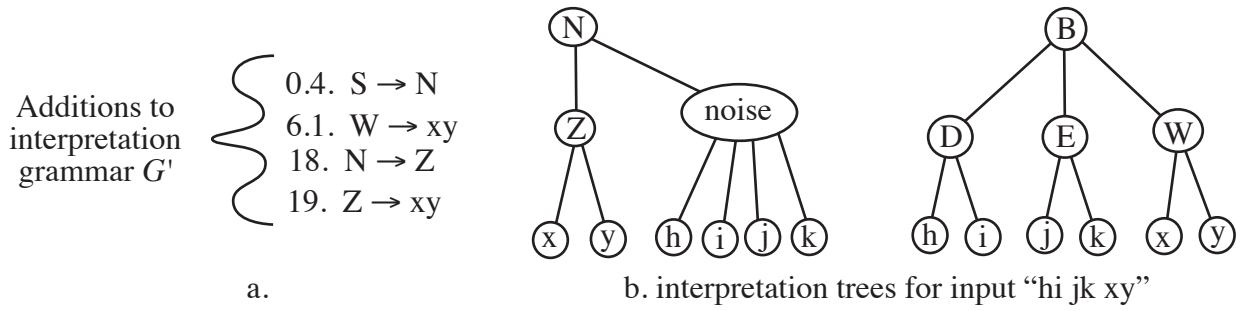


Figure 29: An Example of Correlated Missing Data - The data missing in rule 6.1, w, is correlated to an interpretation of N.

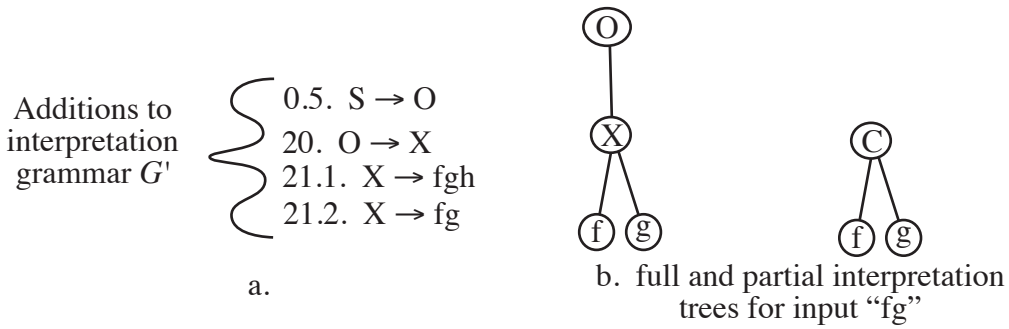


Figure 30: An Example of Uncorrelated Missing Data

rule has a credibility that is lower than the corresponding "complete-data" rule. Missing data can increase the ambiguity of an IDP and, in most cases, this will increase the cost of differentiating correct interpretations from incorrect interpretations.

Figure 24 also shows G' with added rules representing missing data rules. Figure 29 shows an example of a situation where missing data leads to multiple interpretations of signal data. In this example, missing data is added to grammar G' in the form of rule 6.1. The signal input to the problem solver is "hi jk xy" and, as a result of missing data rule 6.1, the data can be interpreted as a B. However, the same data can also be interpreted as an N. The determination of which is correct cannot be made without additional processing. The partial interpretation corresponding to state Z may have a higher rating than the partial interpretations corresponding to states D, E, or W. The differentiation must be made between states N and B. Here, B will have a higher rating since it explains all the signal data and N only explains the data corresponding to x and y. This is an example of missing data causing ambiguity.

Figure 30 depicts a situation where missing data again leads to multiple interpretations. This situation is also based on the grammar shown in Fig. 24. In this example, missing data is added to grammar G' in the form of rule 21.2. In this situation, an interpretation of O will result from the input "fg." In addition, partial interpretation C will be formed. Partial interpretation C will not be extended since the data needed to generate an A is not present. Therefore, C will be connected without resulting in the generation of a full interpretation. This is an example of missing data causing semi-ambiguity.

Missing data can also be categorized into two broad classes.

Definition 5.7 *Correlated Missing Data* – Missing data that results in ambiguous interpretations. Similar to correlated noise, correlated missing data can significantly increase the amount of work or other resources required to solve an IDP if it requires the problem solver to generate multiple interpretations in order to differentiate the competing partial interpretations. Correlated missing data is especially problematic when the correlation is high and the introduction of missing data leads to numerous incorrect, but credible, interpretations. Figure 29 shows an example of correlated missing data.

Definition 5.8 *Uncorrelated Missing Data* – Missing data that does not result in ambiguous interpretations. Similar to uncorrelated noise, uncorrelated missing data may lead to semi-ambiguous IDPs, but, in general, is easier to differentiate than correlated missing data. With uncorrelated missing data, the problem solver is not required to generate complete interpretation trees to differentiate the correct missing data partial interpretation from incorrect competing alternatives. Instead, the competing alternatives will be eliminated when the problem solver fails to extend them at some point of the interpretation process, either as a result of syntactic constraints of the grammar or boundary function constraints (see Subsection 5.5) defined by the functions f_p and Γ_p . In other words, the incorrect partial interpretations will not correspond to any possible full interpretation and the problem solver will not be able to inversely apply any production rules to extend the interpretation. Figure 30 shows an example of uncorrelated missing data.

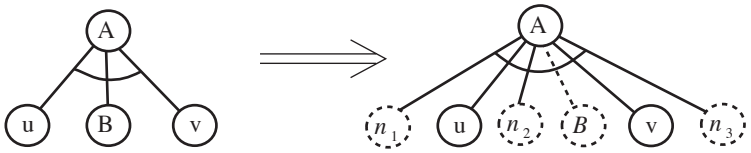
As with noise, the degree to which missing data is correlated to other domain events is a continuum from uncorrelated to correlated. Again, the continuum is expressed in terms of the additional cost that must be incurred before enough constraints are applied to eliminate an incorrect partial interpretation from further consideration. In other words, highly correlated missing data will increase interpretation cost significantly and missing data with low correlation will increase interpretation cost only slightly.

5.1.4 Distortion

Distortion is an effect caused by the combination of noise and missing data. Distortion is represented as a distinct phenomenon for representation clarity and convenience. The causes and effects of distortion are the same as the causes and effects of noise and missing data. In many domains, distortion is a common phenomenon. For example, in acoustic sensing domains, two low energy peaks that are close to each other in the frequency domain might combine into a single high-energy peak at a frequency between the two original signals.

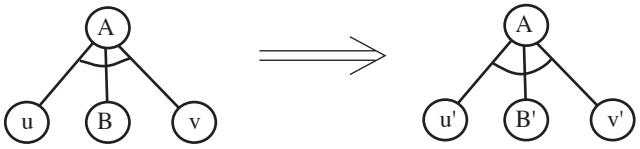
Definition 5.9 *Distortion* – A grammar, G, is subject to distortion iff
$\exists$ p $\in$ P, where p: A $\rightarrow$ uBv $\Rightarrow$ p$'$: A $\rightarrow$ n_1un_2Bn_3 | n_4Bn_5vn_6 | n_7Bn_8 | $n_9\lambda n_{10}$, for u,v $\in$ (V $\cup$ N)*, B $\in$ (V $\cup$ N), A $\in$ (SNT $\cup$ N), $n_i \in$ (V $\cup$ N)* and $f_p(f_u, f_B, f_v, \Gamma_p(u, B, v)) > f_{p'}$ for any combination of parameters. In addition, the distribution of p and p' is modeled by ψ.

Figure 31 is a representation of the definition of distortion. Distortion implies the existence of multiple RHSs for some production rule and, by definition, a distorted rule has a credibility that is lower than the corresponding non-distorted rule. Distortion causes problems that are identical to those caused by both noise and missing data. This includes increasing both the ambiguity of a grammar and the associated cost of problem solving. In addition, *correlated distortion* and *uncorrelated distortion* are phenomena that have definitions identical to correlated and uncorrelated noise and



d. The production rule p: A -> uBv and the corresponding `distortion' rule p': A -> $n_1 u n_2 v n_3$.

Figure 31: An Example of a Distortion Grammar Rule



e. The production rule p: A -> uBv and the corresponding `masking data' rule p': A -> u'B'v'.

Figure 32: An Example of a Masking Grammar Rule

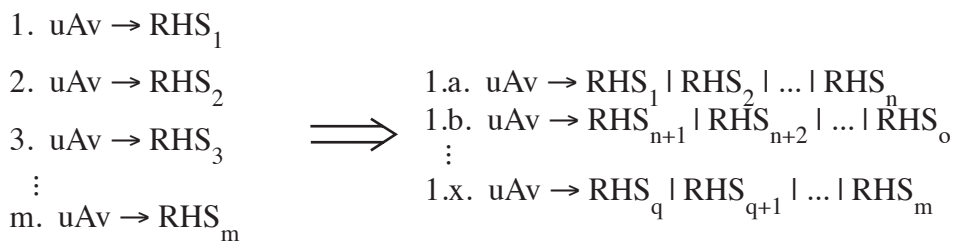
missing data. As with noise and missing data, the correlation of distortion to potentially correct, alternative interpretations defines a continuum in terms of the increased cost required to eliminate incorrect interpretations.

5.1.5 Masking

Masking is a special case of distortion where noise and missing data phenomena combine to generate an RHS that is very similar to the original RHS – for each element e on the RHS of the production rule, p, there is a corresponding element e' on the RHS of p$'$ that has a very similar semantic interpretation. Masking is thought of as a distinct phenomena for intuitive reasons. There are specific, real-world events that lead to masking phenomena in many interpretation domains and these events are significantly different from the events that result in noise, missing data, or distortion. For this reason, masking can be thought of as a unique phenomena rather than a combination of noise and missing data. Formally,

Definition 5.10 *Masking* – A grammar, G, is subject to masking iff
$\exists$ a rule p $\in$ P, where p: A $\rightarrow$ uBv $\Rightarrow$ p$'$: A $\rightarrow$u$'$B$'$v$'$, for u,v,u$'$,v$'$, $\in$ (V $\cup$ N)*, A $\in$ (SNT $\cup$ N), B,B$'$ $\in$ (V $\cup$ N), and $f_p(f_u, f_B, f_v, \Gamma_p(u, B, v)) > f_{p'}(f_{u'}, f_{B'}, f_{v'}, \Gamma_{p'}(u', B', v'))$. In addition, the distribution of p and p' is modeled by ψ.

Figure 32 is a representation of the definition of masking. By definition, a masking data rule has a credibility that is lower than the corresponding non-masking rule. The semantic interpretation of the masking rule is usually very similar to that of the masked rule. The biggest problem introduced by masking is that it causes correct interpretations to be given lower credibility ratings. This can have detrimental effects when the masked/masking rules are correlated to noise associated with other production rules. In these situations, the noisy rules may have higher credibility values and the problem solver may be forced to expend some amount of work differentiating the alternatives.



Example of an operator organization structure specification.
All RHSs included in 1.a through 1.x are elements of the structure.

Figure 33: Example of Operator Organization Representation

5.2 Operator Organization

In Section 4, it was specified that the primitive operators available to an interpretation problem solver are represented as rules of the characteristic grammar for the problem solver's domain. In the previous subsection, inherent uncertainty was defined in terms of additional grammar rules incorporating noise, missing data, masking effects, or some combination of these phenomena. As stated, these additional rules imply the existence of additional problem solving operators.

However, it is possible to organize or group sets of grammar rules into single operators. This is especially useful for efficiently combining large numbers of rules associated with inherent uncertainty and it is done in order to reduce the control costs of invoking operators. By invoking multiple operators as a single unit, it may be possible to reduce overhead costs. In general, two or more grammar rules may be combined into a single *macro-operator*[10] if they are all pairwise syntactically related. Two rules are syntactically related if they have identical left-hand-sides (LHSs). Rules that do not have identical semantics or that are not syntactically related can also be grouped into operators, but such groupings are not considered an operator organization structure.

Operator organization structures can be thought of as the most primitive units of activity that a system reasons about. Operators are usually organized into structures to promote efficiency. Consequently, operator organization can be thought of as a minor transformation to the grammar, often done to offset the effects of inherent uncertainty. Used in this manner, operator organization is very important.

Figure 33 shows the notation that will be used to represent operator organizations. Primitive rules will be included in a macro-operator by specifying them as RHS options of a rule (the vertical lines represent "or" notation), or by specifying them as RHS options in "subrules." In Fig. 33, the rules labelled 1.b through 1.x are all subrules of 1.a and the RHSs of all these rules are the primitive operators included in macro-operator "1." Essentially, all the primitive rules specified in a macro-operator are "applied" when the macro-operator is invoked.

5.3 Redundancy

Another aspect of interpretation problems that causes difficulty is a special form of ambiguity that is referred to as *redundancy*. Formally,

[10] *When confusion with primitive operators does not result, "macro-operators" will be referred to simply as "operators."*

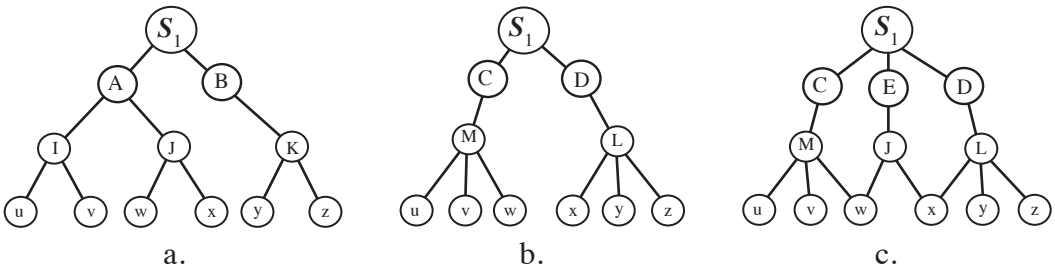


Figure 34: Example of Redundancy

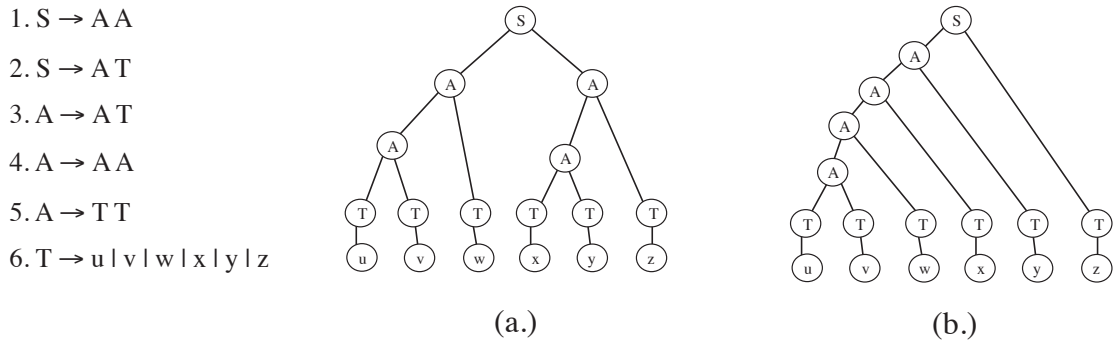


Figure 35: Redundant Interpretations for Input "uvwxyz"

Definition 5.11 *Redundancy* – An IDP is subject to redundancy iff, for some input, X, there exist two or more interpretation trees that have identical semantic interpretations for X.

The presence of redundancy means that, for at least some interpretations, there are multiple solution paths that lead to the exact same interpretation. For example, consider a system that tracks flying objects using radar and sonic sensors. For any given object, the system can rely on radar or acoustic data (or both) to develop an interpretation. Redundancy provides a problem solver with flexibility in choosing problem-solving activities but also allows results to be rederived using alternative paths, possibly without recognizing the redundancy until the last step. Various studies have shown that the cost associated with redundancy can be substantial [26]. Figure 34 shows an example of redundancy. (Note that this grammar is not the same as the grammar that is being used to illustrate other phenomena. The grammar used in the example of redundancy will not be used in any other examples.)

Redundancy is also shown in Fig. 35. This example depicts a stylized version of redundant processing that can occur in certain interpretation tasks such as the Distributed Vehicle Monitoring Testbed (DVMT) [6]. The problem solving strategy embodied in this example offers alternative paths for interpreting tracks of data, in this case the input "uvwxyz." In Fig. 35.b, a track interpretation is formed incrementally by extending a partial interpretation one time unit. Alternatively, in Fig. 35.a, a partial interpretation consisting of multiple track positions is extended by combining it with another partial interpretation that also consists of multiple track positions.

5.4 Interacting Subproblems

One of the most important properties that determines a problem's structure is the nature of its interacting subproblems. In the IDP/*UPC* framework, interacting subproblem structures will form the basis for designing, implementing, and analyzing meta-level operations. In particular, relationships between states will be explicitly represented with abstract or approximate states. Recent studies [26, 27, 9] have demonstrated that knowledge of a problem's structure can be derived by reasoning about interacting subproblems and that this knowledge can be used to more effectively control problem solving activities. For example, some interacting subproblem structures can be used to dynamically implement hierarchical problem solving strategies, or to implement efficient pruning techniques such as the inhibition of redundant processing [10, 26, 9].

In general, control architectures such as the hierarchical problem solving strategies used in [12] and [11, 10, 27] can be modeled by transforming an IDP problem representation to include abstractions and approximations used in control actions in the form of meta-operators. The meta-operators will be represented as special rules of the IDP grammar, along with their associated functions, that are derived from interacting subproblems in the base IDP model.

In domains with interacting subproblem structures, control architectures that are based on "local control" (from Section 3.2) strategies are often ineffective when compared to control architectures that are capable of exploiting the subproblem interactions. However, this is somewhat misleading. Unlike uncertainty and redundancy, interacting subproblems do not *increase* the complexity or difficulty of interpretation problems. Rather, their importance is derived from their ability to *reduce* the cost of problem solving dramatically. Consequently, the need to formally specify structures associated with interacting subproblems is motivated by the desire to exploit these structures and not by the need to avoid or compensate for their presence.

To understand these observations, consider two very similar grammars G_1, with interacting subproblems, and G_2, with no interacting subproblems. Assume that the complexity of the domains described by the grammars is equivalent from the perspective of a common local control architecture. In other words, the local control architecture does not attempt to exploit the interacting subproblem structures in G_1. In this situation, the interacting subproblems in G_1 do not increase the complexity of problem solving in the corresponding domain. However, a control architecture can be constructed to exploit the interacting subproblems and a problem solver using such a control architecture would likely outperform a problem solver using only a local control architecture. This would depend on the costs associated with recognizing interacting subproblems, the accuracy with which the effects of interrelationships are evaluated, the strength with which these interrelationships constrain further problem solving, and the cost of exploiting these relationships.

Though the majority of discussion related to interacting subproblems is contained in later sections and other publications [27], their definitions and representations are included here for consistency.

5.4.1 Defining Interacting Subproblems

Formal definitions of interacting subproblems will be based on the following definitions of *component set* and *result set*. The definitions given here will rely on interpretation grammar G' from Fig. 15, on extensions to G' presented in Fig.25, and on Figures 36, 37, and 38. In these figures, the states labelled A' and D' are indications of multiple instantiations of the states A and D. They do not represent new or unique states.

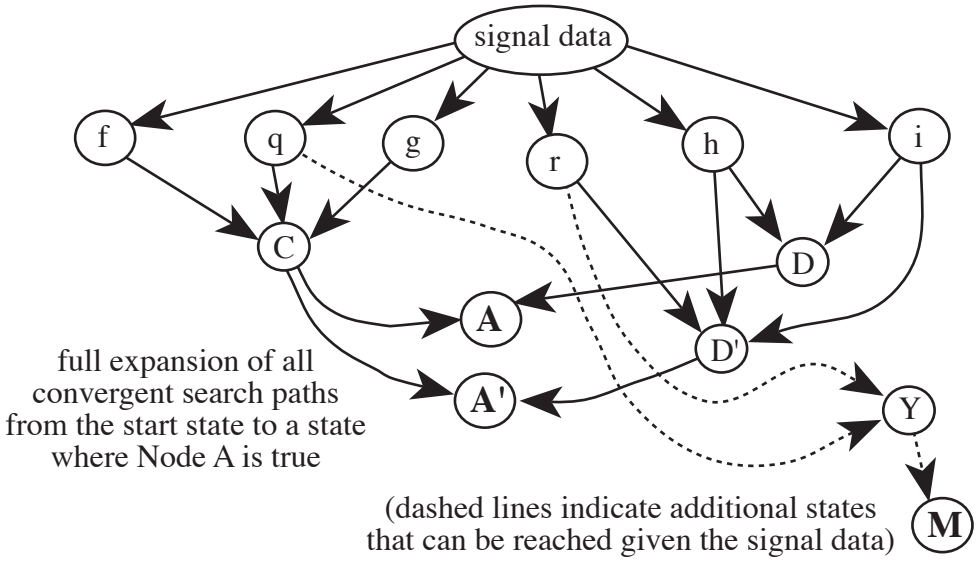


Figure 36: Example of a Fully Expanded Convergent Search Space

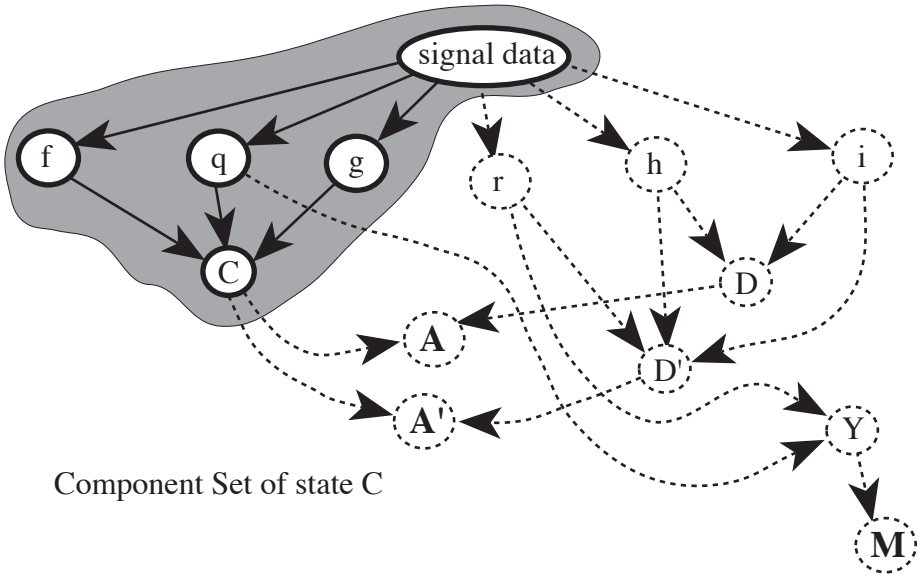


Figure 37: Component Set Example

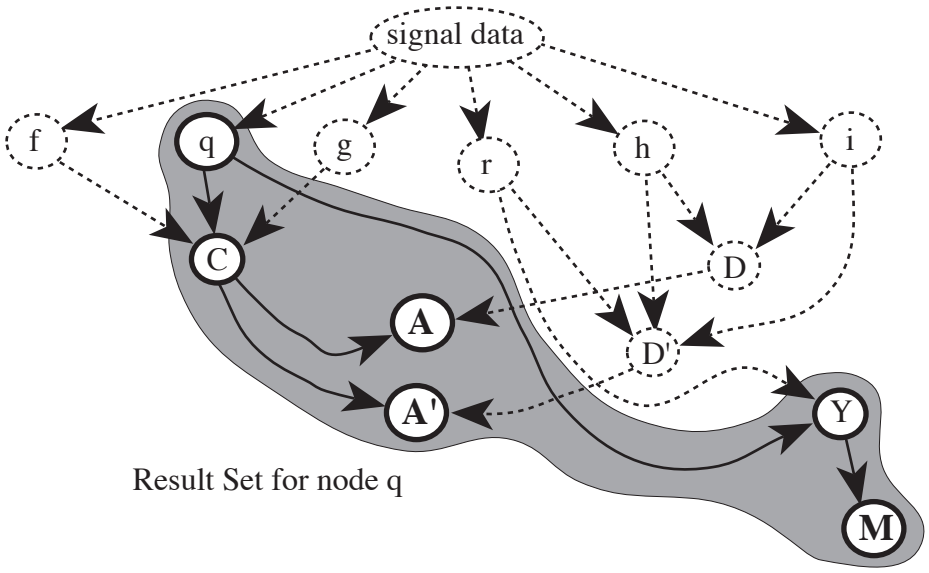


Figure 38: Result Set Example

Definition 5.12 *Component Set (CS) -* The component set of a state, s_n, includes all the states that lie on paths from the signal data to s_n. Figure 36 shows the complete convergent search space associated with an interpretation of A as well as paths to a state corresponding to an interpretation of M. Figure 37 shows the component set of an intermediate state labelled C. An intermediate search state such as C corresponds to a partial interpretation tree, or *interpretation subtree*, t. The component set of a state includes its corresponding interpretation tree or subtree, t.

Definition 5.13 *Result Set (RS) -* The result set of a state, s_n, in a search space includes all the states that can be reached from s_n. Figure 38 shows the result set of the intermediate state q.

Definition 5.14 *Subproblem -* In the IDP framework, a subproblem is any symbol from the alphabet of of an IDP grammar's that appears on the right-hand-side of a production rule.

Definition 5.15 *Subproblem Relationships and Interactions -* If a function, r, is defined in terms of the Component Sets and/or Result Sets of two or more subproblems, we say that r defines a *relationship* (or *interaction*) between the subproblems.

In [26] a number of relationships were defined and their use in controlling problem solving was discussed. We now update these definitions.[11]

Definition 5.16 *Subsumption:* Partial result t_1 subsumes a second partial result, t_2, if the component set of t_2 is a subset of the component set of t_1. Formally, t_1 subsumes t_2 if $\forall y, y \in CS(t_2) \Rightarrow y \in CS(t_1)$.

Definition 5.17 *Competition:* Two partial results, t_1 and t_2, are competing if the intersection of their component sets is non-empty and the intersection of their result sets is empty. Partial results t_1 and t_2 are competing iff $CS(t_1) \cap CS(t_2) \neq \emptyset$ and $RS(t_1) \cap RS(t_2) \neq \emptyset$.

[11]The term "partial result" will refer to a partial interpretation tree.

Definition 5.18 *Cooperation:* Two partial results are cooperating if the intersection of their component sets is empty and the intersection of their result sets is non-empty. In other words, they are cooperating if it is possible that they might be incorporated into a single interpretation tree at some point in the future. Partial results t_1 and t_2 are *completely cooperating* iff $RS(t_1) = RS(t_2)$ (i.e., $RS(t_1) \cap RS(t_2) = RS(t_1) \cup RS(t_2)$. Partial results t_1 and t_2 are *partially cooperating* iff $RS(t_1) \cap RS(t_2) \neq \emptyset$ and $(RS(t_1) \cap RS(t_2)) \neq (RS(t_1) \cup RS(t_2))$.

Definition 5.19 *Overlapping:* Two partial results are overlapping if the intersection of their component sets is non-empty and the intersection of their result sets is non-empty. In addition, their component sets must not be identical. In other words, they are overlapping if they share some component subtrees and if it is possible that they might be incorporated into a single interpretation tree at some point in the future. Partial results t_1 and t_2 are overlapping iff $CS(t_1) \neq CS(t_2)$, $CS(t_1) \cap CS(t_2) \neq \emptyset$ and $RS(t_1) \cap RS(t_2) \neq \emptyset$.

Definition 5.20 *Independence:* Two partial results are independent if the intersection of their component sets is empty and the intersection of their result sets is empty. This means that they are not competing and that it is not possible for them to be incorporated into a single interpretation tree other than an interpretation rooted at the signal data. Partial results t_1 and t_2 are independent iff $CS(t_1) \cap CS(t_2) = \emptyset \ \wedge RS(t_1) \cap RS(t_2) = \emptyset$.

5.4.2 Representing Interacting Subproblems With the IDP Model

In the IDP formalism, interacting subproblem structures will be explicitly represented by abstract states. The relationships captured by these abstract states will be defined in terms of the relationships from the previous section. These relationships will be specified by the addition of grammar rules that will be referred to as *meta-operators*, *meta-rules*, or *abstract operators*. Thus, meta-operators and meta-rules are problem solving actions that either create or modify abstract states. In addition, abstract states will be associated with mapping functions that extend or modify the base space in some way in order to use the results of meta-level processing. The following example illustrates how the IDP formalism can represent interacting subproblems and the associated meta-operators and mapping functions. In this and subsequent publications, we will use the IDP representation of meta-level processing to implement a variety of problem solving strategies, especially top-down processing (branch and bound) strategies.

Figure 40 shows an example of meta-operators for the grammar G' from Fig. 39. We repeat this figure because it naturally represents certain subproblem relationships. This example is intended to illustrate the representation techniques that are used in the IDP framework and it is not intended to demonstrate the utility of the techniques.

In this example, rules M5 and M6 are added indicating that the abstract, "meta-state" M can be derived from an h or an i. Rules M3 and M4 indicate that the abstract states A' and B' can be derived, respectively, from an f (with M) or an x (also with M) respectively. Rules M1 and M2 indicate, respectively, that an A can be derived from an A' and a B can be derived from a B'. Note that these rules are additions to the grammar that are not used for generational purposes, only for interpretation. They are not used to model the events that created the observed signal data and these rules would be ignored by a domain event simulator that is based on the use of an IDP grammar specification.

From the grammar in Fig. 39, it should be clear that h and i are completely cooperating. Consequently, there is no point in pursuing distinct solution paths from h and i - there is no interpretation that can be formed starting with i that will not be formed starting with h, and vice-versa. The abstract

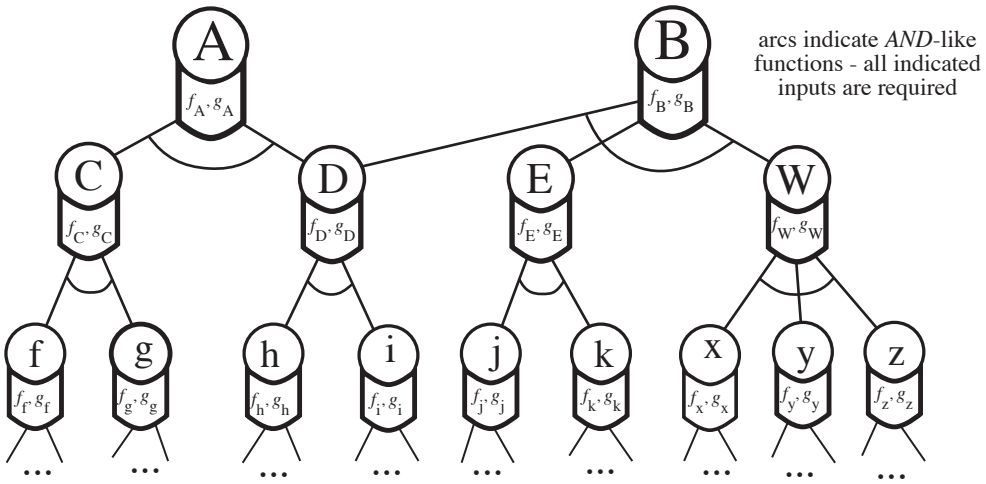


Figure 39: Graphical Representation of Example Interpretation Grammar

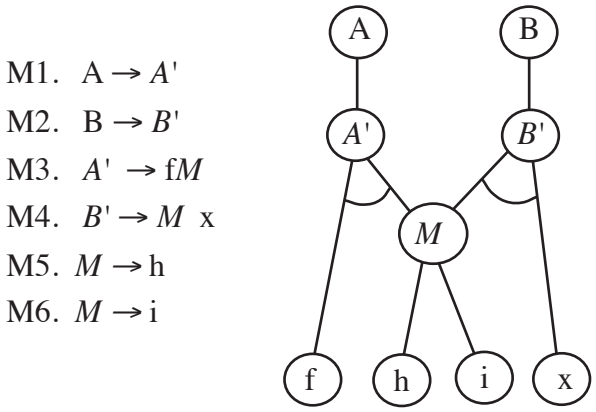


Figure 40: Meta-Operators Expressed as Rules of a Grammar

state M represents this relationship. Intuitively, M can be thought of as a combination of the paths from h and i.

It should also be clear from the grammar that f and x are independent and that f and x are both cooperating with M. The abstract states A' and B' represent these relationships. A' represents the intersection of the Result Sets of f and M and B' represents the intersection of the Result Sets of x and M.

In order to exploit the relationships represented by the abstract states A', B' and M, there must be a corresponding mapping function that, when applied to one of the abstract states, extends or modifies the base space. This role is filled by the mapping functions corresponding to rules M1 and M2.

Rules M5 and M6 can be thought of as meta-operators that project the base space to an abstraction space by generating an abstract state, M, given an h or an i. Rules M3 and M4 can be thought of as top-down operators that bound M. For example, M3 checks for the presence of an f, in which case it generates an A'. If there is no f, M3 fails and generates no new states. Similarly, M4 checks for the presence of an x. If there is such data, M4 will generate a B', otherwise it will fail and not generate any new states.

The rules M1 and M2 map the projection space back to the base space. Given an A', the problem solver knows that the correct interpretation is an A, and it adjusts the *UPC* values of low-level states appropriately. Unlike the other grammar rules, mapping operators do not create new states in the base space[12]. The production rule notation is used to specify the states that are updated, and the information used to update them. Specifically, the states that are updated are those in the component set of the abstract state. These states are updated with information about the final state that is on the left-hand-side of the grammar rule specifying the mapping operator. For example, meta-operator M1 will update the *UPC* values for states M, f, h, and i with information indicating that the probability of generating an A is 1. It is important to note that, even though the problem solver knows the interpretation has to be an A, it still has to execute all the semantic functions to develop the correct "meaning" of the signal data. For example, in a vehicle tracking domain, the problem solver may know that the vehicle it is tracking is of type 1, but it will still have to execute the semantic functions to develop a full interpretation of the vehicle's actions over a period of time. Similarly, given a B', the meta-operator M2 updates the *UPC* values for states M, h, i, and x with information indicating that the probability of generating a B is 1.

The advantages of adding the meta-operators shown in Fig. 40 can be analyzed with the tools that we will define later in this paper. The data needed for proper analysis of the value of these meta-operators has been omitted, but, intuitively, the meta-rules have obvious advantages in terms of reducing the superficial complexity of the problem. Use of these meta-operators will eliminate the application of many redundant processing steps, such as extending search paths from the lower-level states that will eventually turn out to be redundant, and it is reasonable to expect that this will reduce the overall cost of problem solving. Though this example was kept intentionally simple to illustrate the representation used in the IDP formalism, subsequent examples in this and future papers will be based on very similar analysis.

[12]This convention is used for the discussion and analysis presented in this paper and may be altered in subsequent publications.

5.5 Non-Monotonicity and Bounding Functions

Monotonicity is a property of problem structures that has been used to construct many important search control architectures [25]. Though there is no guarantee that a monotone problem can be solved efficiently, most existing efficient control architectures such as AO*, alpha-beta, B*, SSS* and their generalizations are only applicable in monotone domains. In addition, problem solving techniques such as dynamic programming, A* search, Martelli and Montanari's bottom-up search, and Dijkstra's shortest path algorithm are also restricted to monotonic domains.

For the most part, IDP structures of interest will be non-monotone. However, even in non-monotone structures, it is still possible to define *bounding functions* that can be used effectively in control architectures. The structures associated with these functions are defined in this section.

5.5.1 Non-Monotonicity

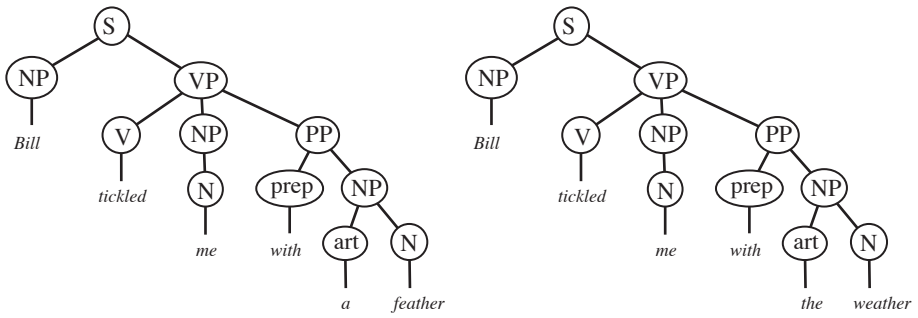
An interpretation problem's monotonicity is specified in terms of the function f_p associated with production rule $p : n \rightarrow n_1 \ldots n_k$. If, $\forall p$, when all of the k-ary credibility attributes associated with the production $p : n \rightarrow n_1 \ldots n_k$ are monotonically nondecreasing in each variable it is implied that the corresponding semantic functions and evaluation functions are also nondecreasing, then the interpretation problem is said to be monotonic. i.e., if $x_1 \leq y_1 \& \ldots \& x_k \leq y_k$ implies $(\Gamma_p(x_1 \ldots x_k) \leq \Gamma_p(y_1 \ldots y_k)$ and $f_p(f_{x_1} \ldots f_{x_k}, \Gamma_p(x_1 \ldots x_k)) \leq f_p(f_{y_1} \ldots f_{y_k}, \Gamma_p(y_1 \ldots y_k)))$, where x_i and y_i represent the i^{th} element of the rule p, and function Γ_p is the semantic credibility function associated with p, then the interpretation problem is monotone.

For many interpretation domains, such as the DVMT[6] and Hearsay-II[12], monotonicity does not hold. A simple example illustrates how this might occur. Figure 41 shows two interpretations of signal data in a speech understanding domain. In this example, the partial interpretation "tickled me with the weather" is rated higher than the partial interpretation "tickled me with a feather." In a monotone domain, this would mean that the full interpretation "Bill tickled me with the weather." would be the solution. This is because $f_{NP}(\text{Bill}) = f_{NP}(\text{Bill})$, which, by the preceding definition of a monotone IDP, would imply that $f_S(\text{"Bill tickled me with a feather."}) \leq f_S(\text{"Bill tickled me with the weather."})$. However, in this example, the best full interpretation of the signal data is "Bill tickled me with a feather." Consequently, this is not a monotone domain.

The non-monotonicity of a domain can be thought of as a function of the semantics of a domain. As individual components are added to a growing interpretation, they must be consistent with all of the other components that have been added so far. In many interpretation domains, such as speech recognition, components with high individual credibilities are often inconsistent with some or all of the other components of a partial interpretation [12]. For example, a particular word may be the best interpretation for a particular time-slice of data, but that word may not be consistent with the "meaning" (i.e., the semantics) of a partial interpretation the problem solver is trying to extend. As a consequence, the resulting extension may have an arbitrarily low-rated credibility depending on how ridiculous it is.

5.5.2 Bounding Functions

In certain monotone domains, problem solving is simplified by the fact that optimal subproblem solutions are guaranteed to be components of the optimal solution [25]. In non-monotone interpretation problems, this guarantee obviously does not hold. However, based on the problem structure defined



$$f_{\text{S}}(\text{“Bill tickled me with a feather.”}) > f_{\text{S}}(\text{“Bill tickled me with the weather.”})$$

$$f_{\text{VP}}(\text{“tickled me with a feather”}) < f_{\text{VP}}(\text{“tickled me with the weather”})$$

$$f_{\text{PP}}(\text{“with a feather”}) < f_{\text{PP}}(\text{“with the weather”})$$

Figure 41: An Example of a Non-Monotone Interpretation Domain

by a domain's characteristic grammar, G, and evaluation function f, certain *bounding functions* can be defined. For example, given a partial interpretation, i, a bounding function determines the likelihood of i being included in a solution with a rating above some *threshold*. If it is unlikely that the partial interpretation will lead to a solution rated above the threshold, all derivation paths including the partial interpretation are pruned. For example, given threshold t, for any partial interpretation, i, if $upperbound(i) < t$, then i can be pruned. $upperbound(i)$ can be defined as the maximum expected utility for any of the final states that can be reached along paths from i.

Intuitively, the threshold can be established in a number of ways. One way is to set it at a level where any solution rated below the threshold would be considered very questionable and the problem solver would not return any such ratings. Another way to set the threshold is to examine the distribution characteristics of interpretations for a domain and set the threshold at a level where there is a very high probability that at least one solution will have a rating above the threshold. In an ideal situation, only a single solution will have a rating above the threshold.

Like interacting subproblems, bounding functions and monotone properties are structures that can be exploited by a control architecture to reduce the overall cost of problem solving. And similar to interacting subproblems, the importance of identifying these structures is based on this potential to increase the efficiency of problem solving.

In addition to structures specified by bounding functions, non-monotone domains often exhibit *semi-monotone structures*. These are structures where the rating of a partial interpretation is correlated to the probability that the partial interpretation is a component of the best interpretation. Thus, partial interpretations with relatively low ratings have a low probability of being a component of correct solution and partial interpretations with relatively high ratings have a high probability of being a component of correct solutions. In semi-monotone domains, it is possible to prune paths based on the credibility of intermediate results in such a way that the probability of eliminating the correct interpretation is known. This strategy can be used in situations where it is possible to gain dramatic performance improvements by accepting a slight risk of eliminating the correct solution from consideration.

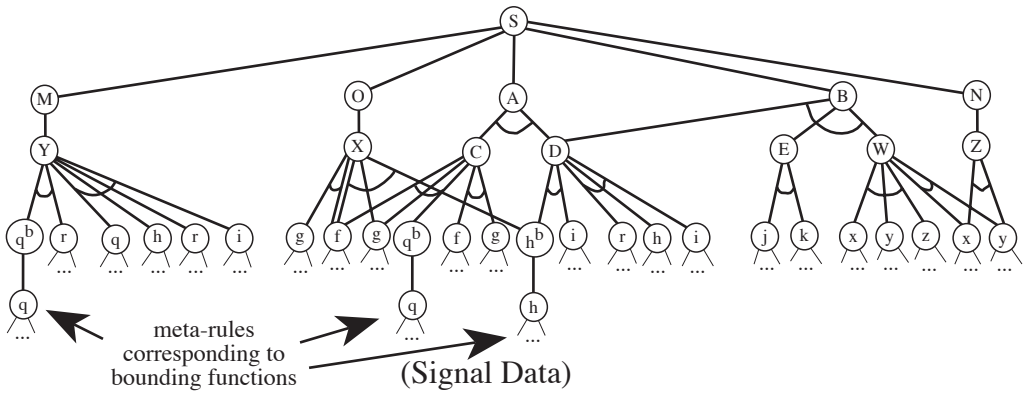


Figure 42: Example of Bounding Function Incorporated in a Grammar

5.5.3 Representing Bounding Functions With the IDP Model

In the IDP formalism, bounding functions are represented by extending the grammar. Given a state, n, for which a bounding function can be defined, every appearance of n on the RHS of a grammar rule is replaced with the nonterminal n^b and the rule $p : n^b \rightarrow n$ is added to the grammar. The knowledge incorporated in the operator corresponding to p is the bounding function on n. Given an n, the operator p executes the bounding function. If the output of the bounding function indicates that extending interpretations using state n is pointless, the operator fails to generate n^b and no further derivations using n are built. If the output of the bounding function is within a range indicating that further interpretations using n should be constructed, the operator will generate n^b and processing will continue. More specifically, in the IDP/*UPC* framework, a state is pruned by the credibility function, f_{n^b}, which returns a value of 0 for that state. As will be discussed in a later section which provide details of the IDP/*UPC* testbed, states with credibility of 0 are not expanded.

In this representation, the context the bounding function operates in, the "bounding context," is not represented explicitly in the grammar. Instead, the bounding context is specified in the definition of the evaluation function, f_{n^b}, associated with bounding rule p. This is due to a lack of a satisfactory representation scheme that would enable the bounding context to be represented explicitly in the grammar in a clear and unambiguous manner.

An example of incorporating bounding functions in a grammar is shown in Fig. 42. In this example, a bounding function has been defined for state q, and the implications have been incorporated in the extended version of grammar G' shown in Fig. 24. The inclusion of a bounding function is represented by the addition of the rule $q^b \rightarrow q$. If, when applied, the bounding function operator determines that there is no point in extending state q, it will fail to generate state q^b. This will prevent the generation of state Y and make the application of the operator corresponding to the rule $C \rightarrow fgq^b$ unnecessary. This bounding function operator might be useful in a situation where state q is generated as the result of noise and has a credibility rating so low that any interpretation incorporating it would have an extremely low rating.

Bounding functions incur a cost to execute and, as a consequence, there is a corresponding increase in the expected cost of problem solving when a grammar is extended to include bounding functions. This increased cost is offset by situations where a bounding function eliminates paths from further consideration and the corresponding costs are not incurred.

To better understand the role of bounding functions, consider the case of a natural language

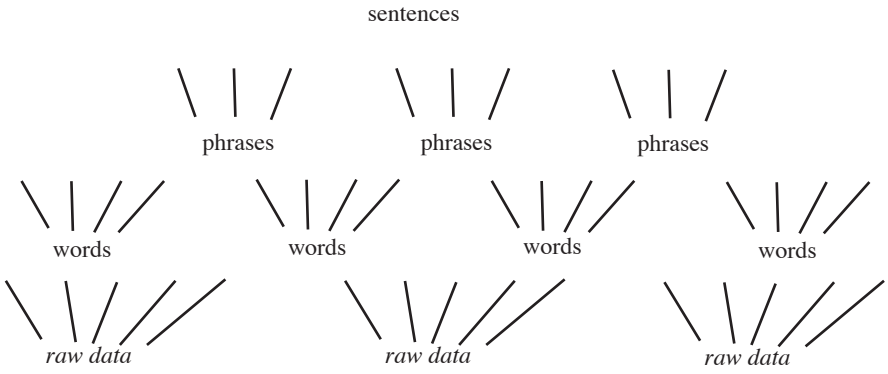


Figure 43: Example of a Natural Language Processing System

understanding system where low-level data is processed into word elements, word elements are processed into phrase elements, and phrase elements are processed into sentences. A representation of this system is shown in Fig. 43. Furthermore, assume that analysis has indicated that words with credibilities less than 0.33 (on a scale of 0 to 1.0) have probability = 0 of being included in a correct interpretation.

In this situation, it may be advantageous to add bounding functions to the grammar to check to see if a word element has a credibility greater than 0.33 before trying to extend the corresponding search state. If a state corresponding to a word has a credibility less than 0.33, then any effort expended trying to extend this state is wasted effort and pruning the state without expanding might reduce the overall cost of problem solving. Whether or not this is advantageous is based on a number of factors, but the general rule is that if the overall cost of executing the bounding functions is less than the cost associated with the pruned search paths, then including bounding functions is advantageous.

For example, consider a case where the expected cost of expanding a state corresponding to a word with credibility less than 0.33 is 10, where the probability of a word having a credibility less than 0.33 is .75, and where the expected quantity of words generated in a problem instance is 100. In this case, the expected cost associated with searching paths from "bad" states, i.e., states corresponding to words that have probability = 0 of being included in a correct interpretation, is $0.75 * 10 * 100 = 750$.

Now, assume that bounding functions are added that prune states with credibilities less than 0.33. If the expected cost of each bounding function is 10, the overall cost associated with the bounding functions will be cost multiplied by the expected number of words that the bounding functions are applied to, or $10 * 100 = 1000$. In other words, the cost of applying the bounding functions would be 1000, and, from the previous equation, the cost without the use of the bounding functions would be 750. Thus, the use of bounding functions would increase the expected cost of problem solving by 250 units.

Now consider if the expected cost of each bounding function is 5. In this case, the cost of applying the bounding functions is $5 * 100 = 500$. Thus the use of bounding functions reduces the expected cost by 250 units.

To illustrate another issue that must be considered in the analysis of bounding functions, consider the situation where the expected cost of applying each bounding function is 5, the expected number of words generated in each problem solving instance is 100, the expected cost of searching a bad path is 10, all as in the previous situation, but the probability of a word having credibility less than 0.33 is 0.25. In this case, the expected cost of applying the bounding functions will still be the expected cost of each bounding function application multiplied by the expected number of words, $5 * 100 = 500$, but

the expected cost savings from pruning bad paths will be the probability that a word has a credibility less than 0.25 multiplied by the number of words multiplied by the expected cost of a bad path, or $0.25 * 100 * 10 = 250$. In this situation, the net effect of using bounding functions is to increase the expected cost of problem solving by 250 units.

In the IDP/*UPC* framework, explicitly representing bounding functions as an integral part of the set of problem solving actions facilitates analysis such as that presented in the preceding paragraphs. In addition, the IDP/*UPC* framework can be used to analyze more sophisticated bounding function implementations where the thresholds are determined dynamically. This will be demonstrated further is subsequent sections.

6 Quantitative Analysis and Experimentation with IDP Models

Within the IDP/*UPC* analysis framework, the IDP formalism is used to generate simulated domain events in an experimental testbed and to define a problem solver's control architecture. Furthermore, it forms a basis for the general analysis framework and the specific analysis paradigms discussed in Section 2. We will refer to the IDP grammar used to generate domain events as the *domain grammar,* IDP_G. The IDP grammar that defines a problem solver will be referred to as the *interpretation grammar,* IDP_I.

The IDP formalism will thus support a variety of experimental activities. Specifically, given a domain, the performance of different control architectures can be tested by modifying the interpretation grammar. Alternatively, given a control architecture, its applicability to and effectiveness in new domains can be measured by altering the domain grammar. Thus, given a control architecture that is very successful in one specific domain, it is possible to identify other domains, or classes of domains, where it will perform equally well. In both these capacities, large numbers of experiments can be run and analyzed quantitatively. In addition, unknown problem structures can be analyzed experimentally using a control architecture as an experimental tool.

Figure 44 represents an abstract view of an experimental testbed based on the IDP formalism that we have developed. In this system, a *Domain Simulator* uses the domain grammar to generate signal data corresponding to domain events. A *Problem Solver* with a control component based on a potentially different, *perceived* IDP structural definition (the interpretation grammar) interprets the signal data. A detailed description of some of the more important aspects of the functioning of the domain simulator is given in Section 6.2.

In addition to its use as an experimental tool, the IDP formalism can also be used as an analytical tool for prediction and explanation. Specifically, we will use the IDP formalism to calculate $E(C)$, the expected cost of a single problem solving instance for a given domain. We will define the calculation of $E(C)$ here, and we will experimentally verify its accuracy in Section 10.

Given the experimental paradigm described above, it will be necessary to calculate $E(C)$ for two different scenarios. In one, the domain grammar and the interpretation grammar are identical. In the other, the domain grammar and the interpretation grammar are different. The calculation of $E(C)$ is very similar for both cases. Using the IDP specifications of the domain and interpretation grammars, we calculate the *expected frequency,* F_n, of each element (i.e., terminal, nonterminal, SNT) in the interpretation grammar. F_n represents the number of expected search state instantiations corresponding to grammar element n per problem solving instance. By multiplying F_n by the expected cost of connecting each of the states corresponding to n, the expected cost associated with grammar

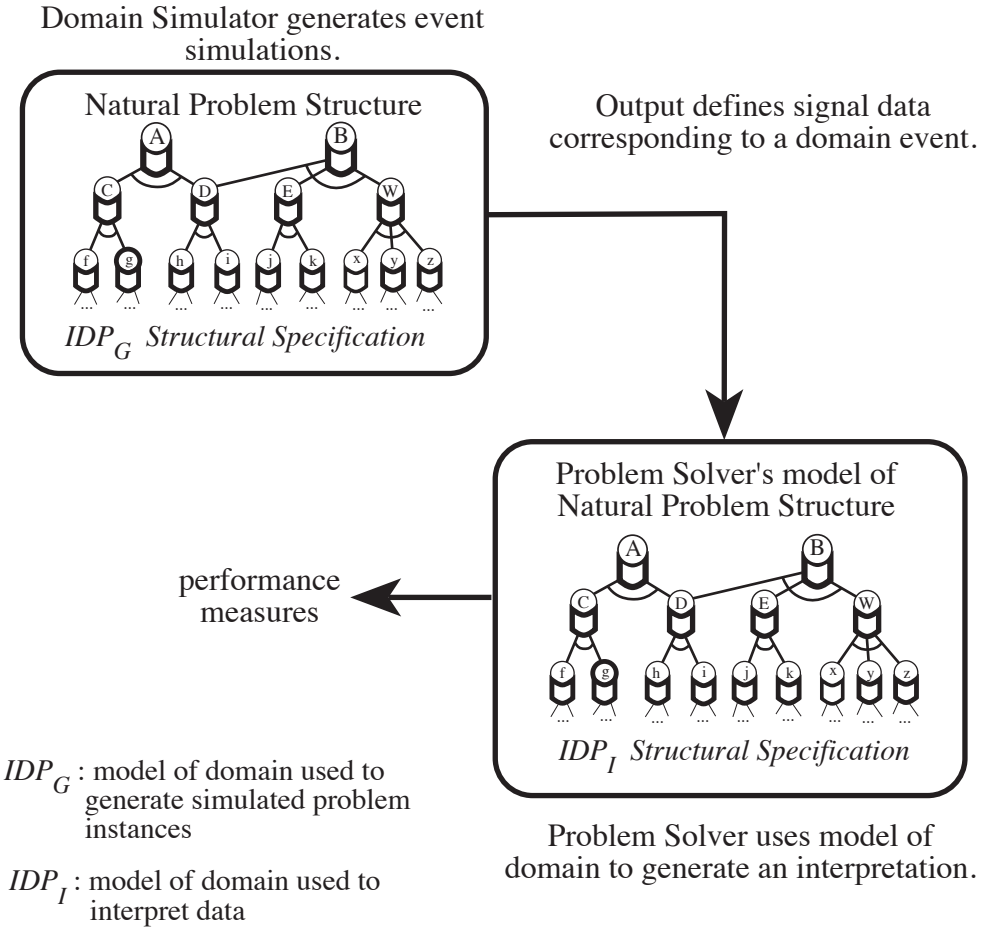


Figure 44: Overview of the IDP Model as the Foundation of an Experimental Testbed

element n is computed. Summing these values for every element of the grammar yields $E(C)$.

It is important to realize that each element of the grammar may have multiple search state instantiations. For example, in a given problem instance, element "A" of the grammar may correspond to many distinct search states. The derivation paths of the distinct states may be differentiated either by different search paths or different inputs, i.e., identical paths with different components. (Two search paths that represent the application of the same sequence of operators will produce different results if any of their inputs are different. In a search space, this is represented by states with slightly different characteristic variables. See Section 9 for more details.) If a specific element of the grammar has an expected frequency of five, it means that, on average, five instantiations of that element of the grammar will be made, each instantiation being a unique state in the corresponding search space. These instantiations (or states) will be distinguished by the individual characteristics of the search paths that lead to the creation of the state. Although redundant paths will lead to the same state instantiation representing the same interpretation or partial interpretation, other search paths, though they correspond to the same element of the IDP grammar, represent different interpretations or partial interpretations.

Section 9 will explain how IDP structures are represented as characteristics of a search state's specification, but it should be intuitive that, for most domains, distinct interpretation trees built from an IDP grammar correspond to distinct search space states. This is because, for the most part, the properties of an interpretation tree are derived from the properties of its subtrees such as "credibility," "location," "time," etc. Section 6.2 describes how these properties are generated.

For example, Fig. 45 shows a situation where signal data leads to the generation of states representing multiple instantiations of the same IDP element. In Fig. 45 the signal data leads to the generation of the six interpretations shown. There are multiple instantiations of the partial result C that are shown in Figs. 45.a and 45.b as C^1 and in Figs. 45.b and Figs. 45.d as C^2. The differences between C^1 and C^2 are seen clearly in the figure as differences in the subtrees that were used to generate the specific instantiations of C. (In an actual implementation, this would be implemented as differences in the characteristic variable "supporting data" associated with each of the instantiations of C.) Similarly, there are two distinct instantiations of the partial results Y and D. The different partial result instantiations are used to generate four distinct instantiations of the SNT A and two distinct instantiations of the SNT M. Finally, there are a total of six different interpretations of the data. This is shown as six different instantiations of S.

For element n of the grammar, where n is a nonterminal or SNT, the calculation of F_n is defined in terms of the elements on the RHS of n's production rules, i.e., the children of n, and any bounding functions incorporated in the grammar. If n is a terminal symbol, its frequency is based on the function ψ and is derived in a top-down manner from the start symbol and from the function ψ. Thus, given a domain grammar, IDP_G, we can determine F_n for $n \in V$, the terminal symbols of the grammar, and use these values to calculate F_n for the nonterminals and SNTs of IDP_G or IDP_I. Given the values F_n, we can then calculate $E(C)$.

It is important to point out that, using the IDP formalism in this way enables us to analytically determine the expected cost of problem solving in situations where IDP_G and IDP_I are the same *and where they are different.* This form of analysis is valid even for grammars with very different nonterminal, SNT, and production rule sets. The only components that two grammars need to have in common is the set of terminal symbols. This is verified experimentally in Section 10. This is a significant result because it will enable us to conduct sensitivity analysis experiments with control architectures by adding, subtracting, or altering rules corresponding to meta-operators and then determining the expected cost of problem solving in the resulting grammar. This will provide the prediction and

interpretation trees derived from signal data “fqg rhi”:

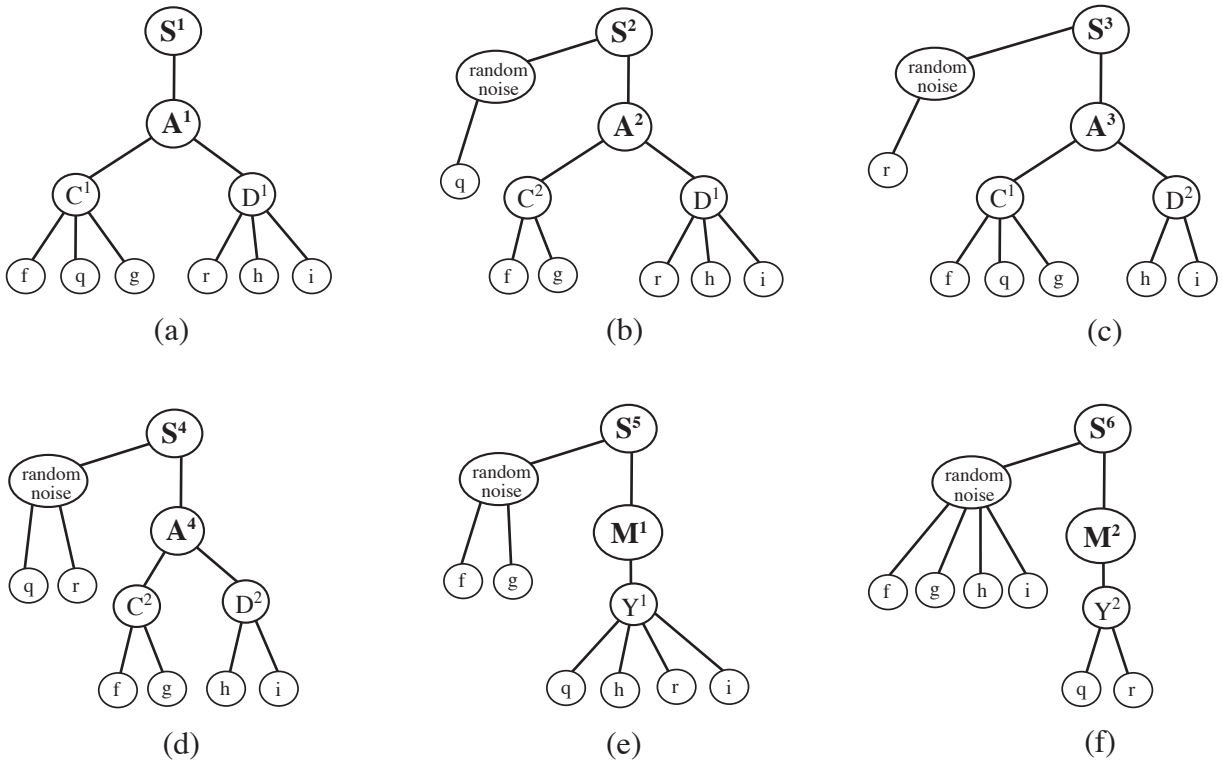


Figure 45: Example of Signal Data Leading to Multiple IDP State Instantiations

explanation capability necessary to develop design theories for interpretation domains.

The remainder of this paper will demonstrate these results. In particular, we will develop the *UPC* formalism that will allow us to characterize control in such a way that the problem structure defined by an IDP grammar can be exploited by a problem solver using an evaluation function based control mechanism. After this, we will show experimental results indicating that the value of $E(C)$ determined analytically from the IDP structure of a grammar is statistically consistent with the actual results of a problem solving system. The following subsection will formally define the calculations of $E(C)$ and expected frequency.

6.1 Calculating $E(C)$ and Expected Frequencies

To define the calculation of $E(C)$, we will start by defining some important concepts that will be used in the calculation.

Definition 6.1 *Sample Set, S_{sample}:* A set of specific problem instances generated using IDP_G. This set can be generated in one of two ways; exhaustively or randomly. When the Sample Set is generated exhaustively, a problem instance is created for every possible combination of rules in the grammar[13]. For example, if nonterminal element A of a grammar has three possible RHSs for one of its production rules, a problem instance will be generated for each RHS. If the Sample Set is generated randomly, the distribution function, ψ, for IDP_G is used to create the samples in a probabilistic way. For example, if nonterminal element A of a grammar has three possible RHSs for one of its production rules, a random number is generated and this is used to determine which of the three rules to use to generate a specific problem instance. For large or complex grammars, it is impractical to use the exhaustive method of generating the Sample Set. However, in some of the experiments described in this paper, this method was used in order to verify that the calculations were correct.

Definition 6.2 *Sample Set Weightings, w_i:* A set of weighting factors associated with the elements of the Sample Set. In the case where S_{sample} is generated randomly and contains n elements, the weightings are $\frac{1}{n}$ for each element of S_{sample}. In the case where S_{sample} is generated exhaustively, the weighting of an element is equal to the product of the ψ values of the grammar rules used to generate the element.

Definition 6.3 *Base Frequency for element n of the grammar, $F^B_{n,i}$:* The expected frequency of n for a given sample i, where i is an element of S_{sample}, is determined from the expected frequencies of n's children, where a "child" of a grammar element n is a terminal or nonterminal element of the grammar that appears on the RHS of one of n's production rules. The base frequency calculation is satisfactory for determining $E(C)$ in situations where there are no pruning operations. For a given production rule, p, for which n is the left-hand-side element, $F^B_{n,i}$ is calculated in a two step process. First, the expected frequency of n from each of the RHSs of p is calculated as the product of the expected frequencies of the elements of the RHSs, or $\prod_j F_{e_j}$, where each e_j is an element of the RHS. Second, the expected frequencies from each of the RHSs of p are combined by a function that is specific for n. Most combination functions are "addition." Thus, the base frequency of n from p is the sum of the base frequencies from each of the RHSs of p. In the case where n is a terminal symbol, its

[13]Note that this does *not* generate every possible string in the language defined by the grammar. Because interpretation grammars use the feature list convention, there can be many (possibly an infinite number) specific problem instances for each combination of grammar rules. Each specific problem instance is distinguished by the values instantiated for variables in its feature list.

frequency corresponds to the number of occurances in sample i. In the case where n is an abstract state in a projection space (these states are defined in Section 11), the combination function often used is "maximum," which takes the maximum base frequency from the RHSs. The combination function used for these abstract states reflects the fact that in certain instances, new abstract states are not created for each combination of input data. Rather, all data is "clustered" into a single state.

Consider the problem instance shown in Fig. 45 as a simple example. For the nonterminal element C, there is one production rule with two RHSs; C $\rightarrow$ fqg | fg. In this example, the frequencies of f, g, and q are all 1 and the base frequency of C associated with each RHS is $(1 * 1 * 1)$ and $(1 * 1)$ respectively. The base frequency of C derived from the rule is the sum of these frequencies, or $(1 + 1)$. As shown in the figure, there are two instantiations of C corresponding to these rules. Similarly, for nonterminal element A, there is a single rule with one RHS; A $\rightarrow$ CD. This results in a base frequency for A of $(2 * 2)$, since the frequency of D is two. As shown in the figure, there are four instantiations of A.

When pruning operations are available, the base frequency calculation must be modified appropriately. The following definitions are used to accomplish this by factoring in the probability that a state will be pruned based on its credibility. If other pruning operations are available, such as an operation that would prune a state based on its probability of being included in a final solution, they would be factored into the calculation of frequency in a similar way.

Definition 6.4 *Expected credibility of element n of the grammar,* $\mu_{Cred}(n) = \sum_i \mu_{f_{p.i}} * \psi(p.i)$, where $\mu_{f_{p.i}}$ is the expected credibility of the evaluation function $f_{p.i}$ and $\psi(p.i)$ is the distribution of the rules $p.i$. In the experiments we describe in this paper, we calculate $\mu_{f_{p.i}}$ by assuming a distribution for the credibility functions $\Gamma_{p.i}$ that is normal with mean equal to the average of the means of the inputs. This enables us to recursively calculate the expected credibility of each element of the grammar based on the expected credibilities of an element's children.

Definition 6.5 *Expected standard deviation from $\mu_{Cred}(n)$ for element n of the grammar,* $\sigma_{Cred}(n) = ftn(\sigma_{f_{p.i}})$, where ftn is a function of the variances of the credibility rules, $f_{p.i}$ and is based on the standard equation $Var(X+Y) = VarX+VarY+2*Cov(X,Y)$. In the experiments we describe in this paper, we assume variances for the credibility functions Γ_p that is equal to sum of the variances of the inputs. (The standard deviation is then the square root of the variance.) We assume that the credibilities and variances of siblings are independent, and this simplifies to $Var(X + Y) = VarX + VarY$. As with expected credibility, we can recursively calculate $\sigma_{Cred}(n)$ for each element of the grammar.

Definition 6.6 *Pruning Modifier for element n of the grammar,* $Pf_n = (1 - P(Credibility(n) \leq T))$, where $Credibility(n)$ is the credibility of the search state corresponding to n that is determined dynamically at run time, and T is the pruning threshold[14]. Given that we have assumed normally distributed credibilites, this value can be calculated from $\phi(\frac{T-\mu_{Cred}(n)}{\sigma_{Cred}(n)})$, where ϕ is available in standard probability textbooks and is equal to $\frac{1}{\sqrt{2\pi}} * e^{-x^2/2}$.

Definition 6.7 *Expected frequency for element n of the grammar,* $F_n = \sum_i w_i * (F_n^B * (1 - Pf_n))$, where i is an element of S_{sample} and $F_{n,i}^B$ the base frequency of n. The expected frequency of n is determined by determining the base frequency of n for element of the Sample Set, modifying this value to reflect pruning actions, multiplying by the sample weight to normalize the value, then summing all values.

[14] In later sections we introduce bounding functions with thresholds that are determined dynamically. In these situations we compute pruning modifiers based on T equal to the expected credibility of a "correct" result for a given problem instance.

Given the expected frequency, it is possible to calculate $E(C)$ by multiplying the expected cost associated with each search state by the frequency of the state and summing for all states.

Definition 6.8 *Expected cost associated with connecting search state* n, $E_{cost}(n) = \sum_p \bar{g}_p$, where p is a production rule, n is an element of some RHS of p, and $\bar{g}_p$ is the expected cost of applying the search operator corresponding to p. To determine the expected cost associated with connecting a state, the expected costs of all the operators that can be applied to the state are summed. An operator can be applied to a state if the grammar element associated with the state appears in any of the RHSs of the grammar production rule associated with the operator. Since a state is connected only when all paths that lead from it are either extended to final states, extended to dead end states, or pruned, this sum represents the cost of connecting a state.

Definition 6.9 *Expected cost of problem solving,* $E(C) = \sum_n F_n * E_{cost}(n)$. The expected cost of problem solving is determined by summing, for each element of the grammar, the expected frequencies of the grammar elements multiplied by the expected cost associated with connecting the corresponding search state.

The procedure used to calculate $E(C)$ can be summarized as:

1. Generate the Sample Set, either randomly or exhaustively.
2. Generate the sample weightings.
3. Using a bottom-up recursion, calculate, in order, the following for each element of the grammar:
   - base frequency of the element
   - expected credibility of the element
   - variance from expected credibility of the element
   - pruning modifier of the element
   - expected frequency of the element
4. Calculate the expected costs associated with connecting search states corresponding to grammar elements.
5. Calculate $E(C)$.

6.2 Generating Problem Instances

The analysis framework described in this paper is embedded in a simulator system that is used to experimentally verify the computational methods used. To fully understand the framework and the verification methods, it is necessary to understand how the simulator works. In particular, it is necessary to understand how the simulator generates problem instances and characteristics of the problem instances, such as credibility.

Figure 46 represents the basic approach used by the simulator to generate problem instances. This approach is based on the notion of the *feature list convention* developed by Gazdar, et al. [15]. As

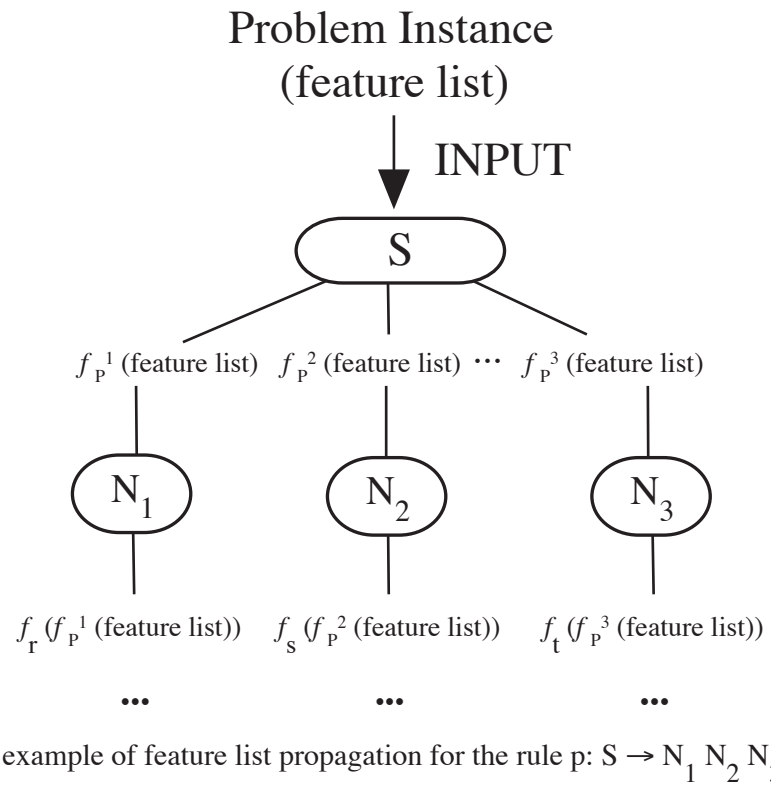


Figure 46: Generating Problem Instances with the Feature List Convention

shown in the figure, each symbol of the grammar has a feature list. In addition, each production rule is associated with a set of functions which take a feature list as an input.

In order to generate a problem instance, the simulator begins by creating a feature list for the start symbol, S. The most important aspect of this task is the determination of a credibility. Each grammar has an expected credibility and variance, and this information is used to probabilistically choose a specific credibility for the problem instance. In addition, other features can be determined at this time. For example, in a vehicle tracking domain, some of the features may include vehicle heading, velocity, type, etc.

Next, the generator probabilistically chooses a production rule, p, associated with the start symbol and applies p to the start symbol. i.e., the generator randomly chooses a production rule with the start symbol on the left-hand-side and replaces S with the right-hand-side of the rule. The choice of which rule to apply is made based on the weights of the ψ associated with the different rules. As the generator replaces S, it assigns each of the replacing symbols a feature list. Each of the feature lists is determined by applying p's feature list functions to S's original feature list.

The new symbols, each with their own, unique feature list, are added to either a queue, if the symbol is a nonterminal, or an output set, if the symbol is a terminal, and the whole process is repeated. The first element of the queue is removed and processed, and the resulting new symbols are added to the tail of the queue or to the output set. This continues until the queue is empty and the output set contains terminals with fully specified feature lists.

More formally, the generation process consists of the following steps:

1. Initialize GENERATION-QUEUE to the start symbol, S.
2. If the GENERATION-QUEUE is empty, terminate. Otherwise, set SYMBOL to the first element of the GENERATION-QUEUE and remove it from the queue.
3. If SYMBOL is a terminal, place it in the OUTPUT-SET and return to step 2. Otherwise, probabilistically choose a production rule, p, to use to expand SYMBOL.
4. For each element of the right-hand-side of p, set its feature list equal to $L_p(SYMBOL_{fl})$, where L_p is a function that creates a new feature list and $SYMBOL_{fl}$ is the feature list associated with SYMBOL.
5. Add each of the elements of the right-hand-side of p to the GENERATION-QUEUE.
6. Return to step 2.

Note that an item's position within the OUTPUT-SET is insignificant. All ordering constraints must be specified in the feature list. For example, the "time" at which an event occurs is represented as a characteristic of a symbol represented in its feature list.

As discussed previously, the basis of the IDP/*UPC* analysis framework is the assumption that the problem instances of a domain occur in patterned, principled ways. We associate events that might shape the characteristics of a problem instance with rules of a grammar that represent, for example, the occurance of an event that shifts a signal slightly or that introduces noise or missing data. Thus, the feature list functions associated with the rules of a grammar modify the feature lists appropriately to represent the events associated with the production rules.

For example, one of the characteristics represented in the feature lists of a vehicle tracking domain might be "energy." Some of the production rules of the grammar may represent events that affect

the perceived energy of a signal. These events may increase or decrease this level. Similarly, other characteristics might include "position," "frequency," etc., and they will all be included in the feature list representation and similarly influenced by which production rules are chosen.

In the test domains that we use, the credibility generation function for a nonterminal element on the RHS of a production rule is:

Equation 6.1 *credibility - $delta_p$.*

Where "credibility" is the credibility from the feature list of the element on the LHS of the production rule and $delta_p$ is an offset that is used to represent the decrease in credibility associated with noise and missing data. For non-noise/missing data rules, $delta_p$ is 0. For rules that represent the addition of noise or missing data, $delta_p$ is greater than 0. For the experiments in this paper, $delta_p$ is typically set to 0.2. This is somewhat unrealistic in the sense that "real world" credibility functions would return specific values that might vary considerably from rule to rule. However, this technique does achieve the desired effect of reducing the credibility associated noise and missing data.

The credibility generation function for a nonterminal element on the RHS of a production rule is:

Equation 6.2 *random variable with density N (credibility, variance).*

Where N is the standard normal density, and "credibility" and "variance" are the credibility and "variance" from the feature list of the element on the LHS of the production rule. "Credibility" corresponds to the expected value of the normal density and "variance" corresponds to the variance.

Intuitively, the credibility of the start symbol is passed down through the generation process to the terminal symbols that constitute the actual signal that is input to the problem solver. As the credibility is passed down from feature list to feature list, it is modified only to reflect effects associated with noise and missing data. In general, we associate noise and missing data with domain events that reduce the credibility of an interpretation, and the feature list generation functions associated with noise and missing data rules reflect this by reducing the credibility included in the new feature lists. Finally, the credibility assigned to terminal symbols is a random number generated with an expected value and variance equal to the credibility and variance passed to the feature list of the terminal symbol.

6.3 An Example of Using Feature Lists in a Grammar

Figure 47 shows a grammar that uses feature lists to generate problem instances for a domain similar to the domain in the DVMT [6]. This grammar has been used to generate problem instances such as the one shown in Fig. 48. The grammar generates a sequence of signal data, each with a time and location. Though the figure does not explicity represent time or the properties (frequency, energy level, etc.) of each piece of signal date, this information is generated. The data corresponds to the sounds generated by a vehicle passing through an area that is being monitored by an array of acoustic sensors. In the figure, it can be assumed that both tracks (the vehicle track and the ghost track) are moving right to left (or vice-versa) and that the time labels of the signal data reflect this. Note that this grammar could be extended easily to include a representation of frequency, energy level, vehicle type, etc.

The key to interpreting the feature lists and functions of feature lists is the following:

f: A feature list. In this example, "f" represents characteristics that are not otherwise represented such as energy, frequency, etc.

t: Time.

Sample Grammar for Complex Vehicle Tracking Domain

1. $*S_{[f]} \rightarrow Noise_{[f]}\ Tracks_{[f]}$
2. $Noise_{[f,t]} \rightarrow Noise_{[f,t+1]}\ N_{[f,t]}$

3. $N_{[f,t]} \rightarrow N_{[f,t]}\ n_{[f,t]}$ p = .10
   $\rightarrow n_{[f,t]}$ p = .25
   $\rightarrow \lambda$ p = .65

4. $Tracks_{[f]} \rightarrow Tracks_{[f]}\ Track_{[f]}$ p = .10
   $\rightarrow Track_{[f]}$ p = .90

5. $Track_{[f,t,x,y]} \rightarrow I\text{–}Track_{[f,t,x,y]}$ p = .50
   $\rightarrow P\text{-}Track_{[f,t+1,x+vel+acc,y+vel+acc]}\ P\text{-}T_{[f,t,x,y]}$ p = .25
   $\rightarrow G\text{-}Track_{[f,t+1,x+vel+acc,y+vel+acc]}\ G\text{-}T_{[f,t,x,y]}$ p = .25

6. $I\text{-}Track_{[f,t,x,y]} \rightarrow I\text{–}Track_{[f,t+1,x+vel+acc,y+vel+acc]}\ T_{[f,t,x,y]}$

7. $P\text{-}Track \rightarrow P\text{-}Track_{[f,t+1,x+vel+acc,y+vel+acc]}\ P\text{-}T_{[f,t,x,y]}$

8. $G\text{-}Track \rightarrow G\text{-}Track_{[f,t+1,x+vel+acc,y+vel+acc]}\ G\text{-}T_{[f,t,x,y]}$

9. $P\text{-}T_{[f,t,x,y]} \rightarrow T_{[f,t,x+offset,y+offset]}\ T_{[f,t,x,y]}$

10. $G\text{-}T_{[f,t,x,y]} \rightarrow G_{[f,t,x+offset,y+offset]}\ T_{[f,t,x,y]}$

11. $T_{[f,t,x,y]} \rightarrow V_{[f,t,x,y]}\ Correlated\text{-}Noise(?)_{[f,t,x,y]}$

12. $V_{[f,t,x,y]} \rightarrow GC1_{[f,t,x,y]}\ GC3_{[f,t,x,y]}\ GC7_{[f,t,x,y]}$ p = .40
    $\rightarrow GC1_{[f,t,x,y]}\ GC3_{[f,t,x,y]}$ p = .30
    $\rightarrow GC1_{[f,t,x,y]}\ GC7_{[f,t,x,y]}$ p = .25
    $\rightarrow \lambda$ p = .05

13. $G_{[f,t,x,y]} \rightarrow G\text{-}GC1_{[f,t,x,y]}\ G\text{-}GC3_{[f,t,x,y]}\ G\text{-}GC7_{[f,t,x,y]}$ p = .70
    $\rightarrow G\text{-}GC1_{[f,t,x,y]}\ G\text{-}GC3_{[f,t,x,y]}$ p = .20
    $\rightarrow G\text{-}GC1_{[f,t,x,y]}\ G\text{-}GC7_{[f,t,x,y]}$ p = .05
    $\rightarrow \lambda$ p = .05

* The subscript notation in brackets indicates *feature list* information where the symbol f indicates all feature list list information not represented explicitly.

Figure 47: An Example of Feature Lists in a Grammar

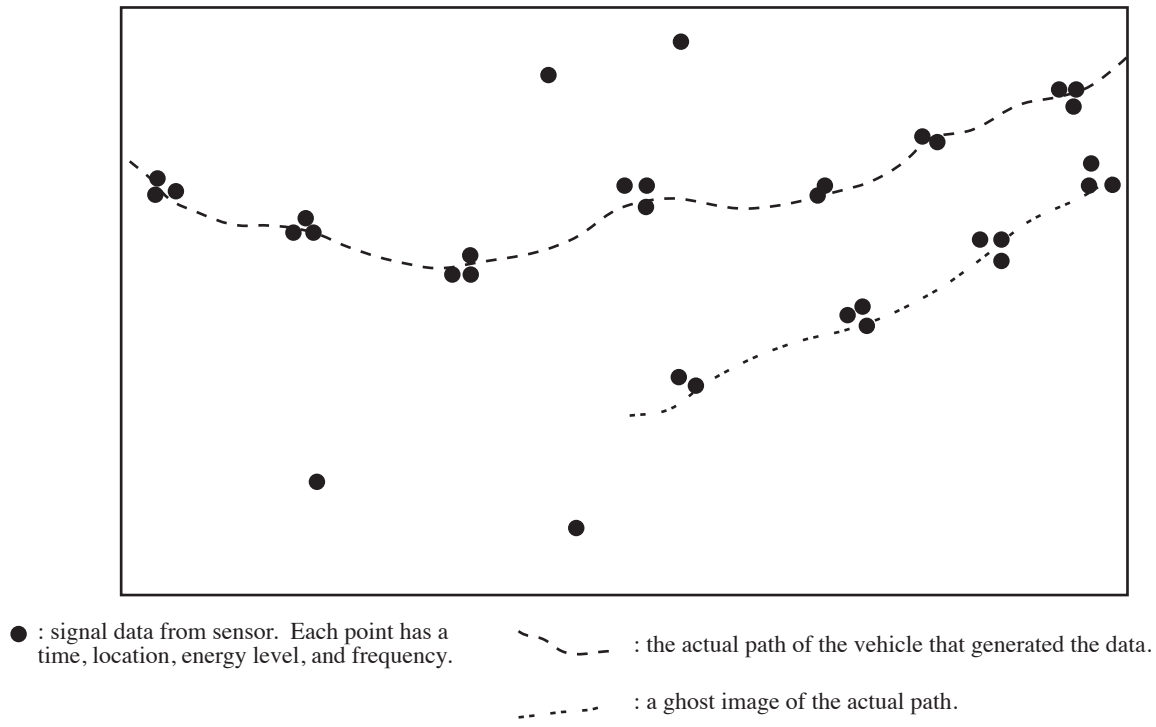


Figure 48: Problem Instance Generated with Grammar and Feature Lists

x: The x-coordinate of a location.

y: The y-coordinate of a location.

vel: Velocity. This is used to generate the x and y coordinates of the next location in the track.

acc: Acceleration. This is used to adjust velocity over time. Velocity and acceleration are also used to constrain the progression of a track. For example, each particular kind of vehicle track has a maximum velocity and acceleration that cannot be exceeded.

offset: The offset from a "true" position. This is used to generate pattern tracks of multiple vehicles and ghost tracks.

As shown in the grammar, rules 1, 2, and 3 generate random noise. Rules 1, 4, and 5 generate vehicle tracks. A vehicle track is extended to a new time-location by rules 5, 6, 7, and 8. Rule 5 is used to initiate tracks. Rule 6 generates single, "Independent Tracks." Rule 7 specifies "Pattern Tracks." Rule 8 specifies "Ghost Tracks." For a given time-location in a ghost or pattern track, rules 10 and 11 generate the signal data at an appropriate offset. Rules 12 and 13 are used to generate the actual signal data corresponding to the sounds made by a vehicle or noise.

7 Analyzing Control Architectures with IDP Models

This section will demonstrate how IDP models can be used to analyze sophisticated control mechanisms. In particular, we will focus on the use of abstractions and approximations[15] in sophisticated control architectures. The examples presented will help motivate the development of the *UPC* formalism that will be introduced in Section 9.

The analytical power of an IDP domain model is based on the degree to which the abstractions used in control can be viewed from the same perspective as problem solving actions. Thus, the key to analyzing abstractions is to view them in terms of the domain problem structures they define. In the IDP formalism, a domain's problem solving actions are represented in terms of the domain's characteristic grammar and the associated functions. These operators generate fully specified partial interpretations and ignore all considerations of the efficiency of their actions. This is in contrast to meta-operators based on abstractions or approximations that generate partial interpretations that are underconstrained in some way. For example, a characteristic variable may be undefined or may be defined by a range of values. Therefore, analyzing abstractions used in sophisticated control mechanisms within the context of a given domain requires that the domain's grammar and associated functions be modified to represent the abstractions as problem solving actions. The modified grammar then defines a new convergent search space and comparative analysis can focus on the relative efficiency of problem solving in the original and the modified search spaces.

Once a domain's grammar has been modified to represent available abstract problem solving actions, the subsequent analysis must focus on two issues. The first will be referred to as *correctness*. Analysis must demonstrate that the problem solving actions represented in the modified grammar generate the same (or acceptably different) results as the problem solving actions represented in the original grammar. The second issue is *efficiency*. Analysis must demonstrate that problem solving in the search space defined by the modified grammar is more efficient than problem solving in the search space defined by the original grammar. Note that this methodology implies that abstract problem solving operators are used solely to improve a problem solver's efficiency.

Proof of efficiency can take a number of forms depending on whether top-down or bottom-up (or both) methods are used. In both cases, analysis will focus on the expected costs to connect the search spaces both before and after modifications corresponding to abstract operators are made. If bottom-up methods are being employed, the analysis must start with the low-level components and show that the cost of generating interpretations from these elements has been decreased. If top-down methods are used, the analysis must start with the high-level representation of the set of interpretations and demonstrate that the cost of pruning this set is somehow decreased. As will be seen, both top-down and bottom-up problem solving can be analyzed similarly when viewed from the perspective of states in a search space.

The analysis contained in this section will focus on abstractions used in sophisticated control mechanisms that have been implemented in the *Distributed Vehicle Monitoring Testbed (DVMT)* [6] and that have been implemented to exploit problem structures of non-monotonic domains. Such domains are characteristic of real-world domains such as signal interpretation, robotic audition, image processing, and natural language processing.

[15] In the remainder of this paper, both abstractions and approximations will be referred to simply as abstractions except in cases where it is necessary to differentiate them.

7.1 Goal Processing

Goal processing is a form of hierarchical problem solving [23] that has been incorporated in the DVMT to enable a problem solver to reason about courses of action in ways that are independent of the means for instantiating the actions. After new search states are generated, partial constraints are applied to the states to generate meta-states, or *goals*. Further problem solving actions are then applied to the goals and the original search states are considered to be connected and are no longer used to initiate problem solving activity. In numerous studies, goal processing has been shown to be an effective means for countering local redundancy and uncertainty [4, 5, 6]. In addition, goal processing has been shown to be effective in top-down processing algorithms where goals are used to constrain the actions of data-directed operators [4, 5, 6].

Intuitively, goals can be thought of as "set descriptors." A goal is similar to a state of the original search space, but it is underconstrained in the sense that some of the characteristic variables that normally define states in the search space are unspecified or specified in terms of a range of values. As a consequence, any state with characteristics that fall within the ranges defined by the goal can be thought of as elements of a set represented by the goal. Hence, goals are abstractions of search space states. In the initial DVMT implementation, goals define a set that represents the elements of a state's result set (defined in Section 5.4.1) that can be generated with the application of a single operator. Extending a goal state can be thought of as a branch-and-bound search operation applied to the goal.

Returning to grammar G' from Fig. 15, we see that the structure of the search space defined by this grammar is very simple. In fact, problem solving in this domain would be more appropriately thought of as classification problem solving since it would be trivial to preenumerate the set S from which an interpretation would be chosen. Furthermore, there is no ambiguity, so interpretations could be determined without using the evaluation functions f_p. Therefore, to make the following examples more meaningful, the extended grammar, G'_n, shown in Fig. 49, will be used.

Figure 49.a represents the operator organizational structure for the example domain problem. Each of the numbered macro-operators shown in Fig. 49.a is actually a set of primitive rules – a "best rule," which is listed first, and subsets of masking rules and missing data rules. For example, rule 1 specifies the production $A_n \rightarrow C_n D_n$ and the eight associated masking rules specified by the subscript $n \pm 1$. (n might correspond to a slight variation in position, time, frequency, etc.) Thus, $A_n \rightarrow C_{n-1}D_{n-1} \mid C_{n-1}D_n \mid C_{n-1}D_{n+1} \mid \ldots$, are all primitive operators that will be applied when macro-operator 1 is applied. In addition, some of the macro-operators include primitive missing data rules. These are macro-operators 3 through 6. The primitive missing data rules are 3.b&c, 4.b&c, 5.b&c, and 6.b through g. For now, the evaluation, cost, and distribution functions will be left undefined.

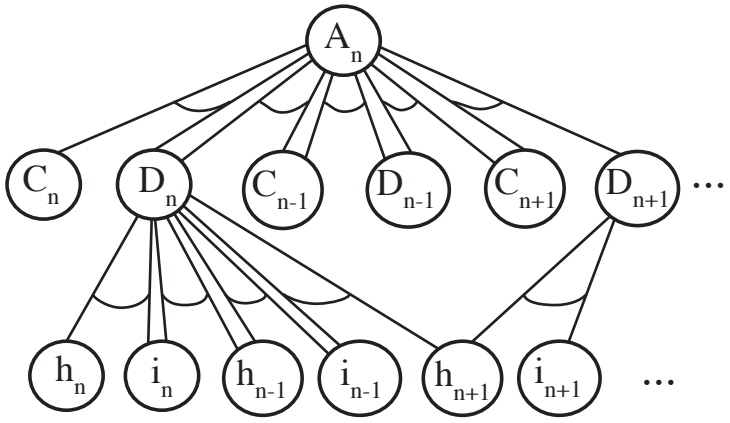
Figures 49.b and 49.c illustrate structures of the grammar in tree form. Figure 49.b depicts structures associated with the masking rules and Fig. 49.c shows structures associated with missing data rules. Note that this grammar is very ambiguous – any input can lead to the derivation of many different interpretation trees. For example, an input of "$f_2 g_2 h_2 i_2$" will result in the generation of several dozen full interpretation trees (see Fig. 50.). Depending on the semantics of the domain, each of these interpretations may be virtually identical or vastly different.

Figures 51 and 52 illustrate the effects of goal processing. These figures are based on the extended interpretation grammar G'_n from Fig. 49. Figure 51 depicts a typical interpretation search for a small set of input data[16]. To connect the search space, op_6 is successively applied to states x_1, y_1, y_2, and z_1.

[16]In these figures, and in the following text, the operator superscript notation represents the i^{th} application of an

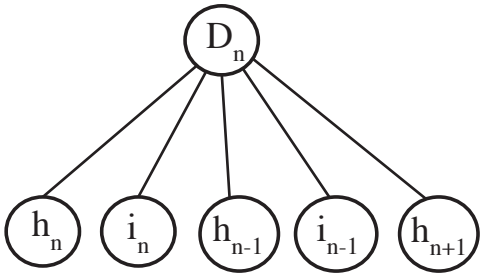
1. $A_n \rightarrow C_{n\pm1}D_{n\pm1}$
2. $B_n \rightarrow D_{n\pm1}E_{n\pm1}W_{n\pm1}$
3.a. $C_n \rightarrow f_{n\pm1}g_{n\pm1}$
3.b. $C_n \rightarrow f_{n\pm1}$
3.c. $C_n \rightarrow g_{n\pm1}$
4.a. $D_n \rightarrow h_{n\pm1}i_{n\pm1}$
4.b. $D_n \rightarrow h_{n\pm1}$
4.c. $D_n \rightarrow i_{n\pm1}$
5.a. $E_n \rightarrow k_{n\pm1}j_{n\pm1}$
5.b. $E_n \rightarrow k_{n\pm1}$
5.c. $E_n \rightarrow j_{n\pm1}$
6.a. $W_n \rightarrow x_{n\pm1}y_{n\pm1}z_{n\pm1}$
6.b. $W_n \rightarrow x_{n\pm1}y_{n\pm1}$
6.c. $W_n \rightarrow x_{n\pm1}z_{n\pm1}$
6.d. $W_n \rightarrow y_{n\pm1}z_{n\pm1}$
6.e. $W_n \rightarrow x_{n\pm1}$
6.f. $W_n \rightarrow y_{n\pm1}$
6.g. $W_n \rightarrow z_{n\pm1}$

a. extended interpretation grammar, G_n'



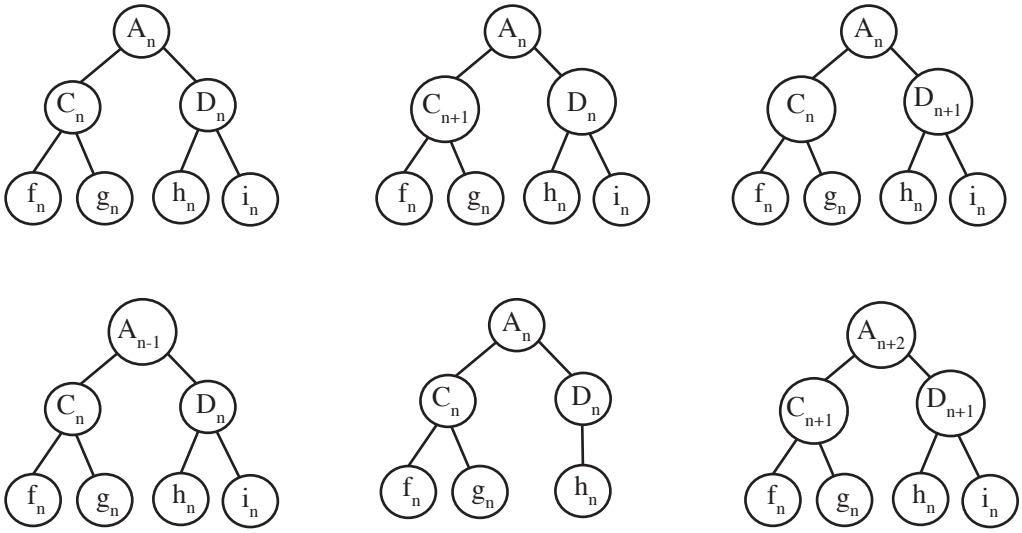
(signal data)

b. example structures from grammar



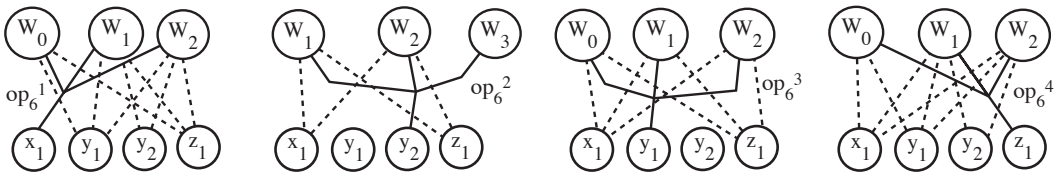
c. additional intermediate structures from grammar

Figure 49: Extended Interpretation Grammar



Some of the interpretation trees resulting from the input "$f_2g_2h_2i_2$."

Figure 50: Example of Interpretations Based on Extended Grammar G'_n



a. operator 6 applied to x_1

b. operator 6 applied to y_2

c. operator 6 applied to y_1

d. operator 6 applied to z_1

(Solid lines represent search paths resulting from operator application and dashed lines represent implied merge operations with supporting data used by the operators.)

Figure 51: Example of Interpretation Search

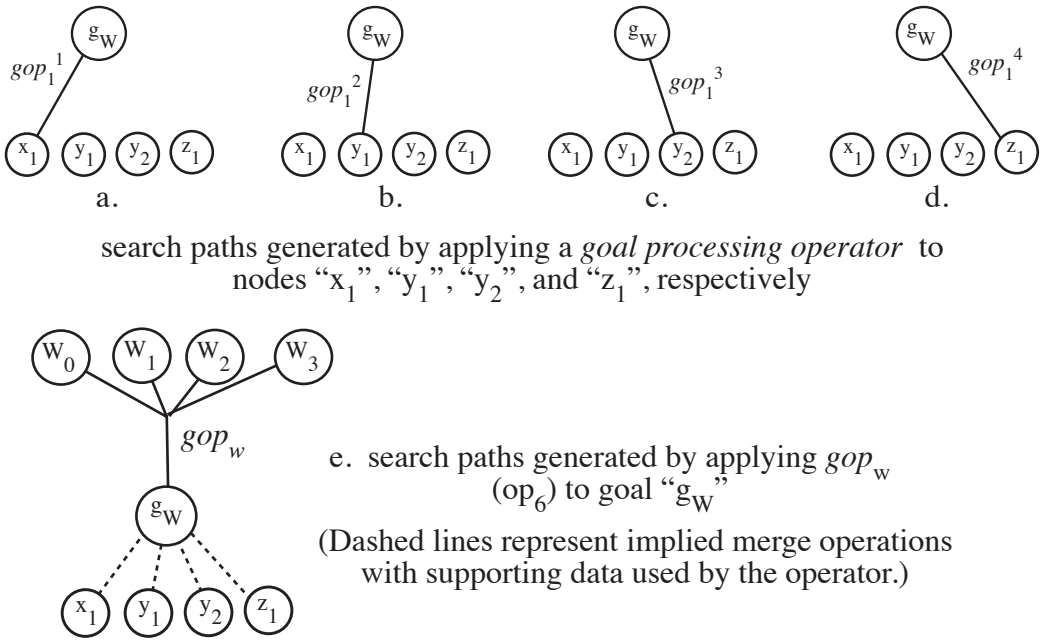


Figure 52: Example of Interpretation Search Using Goal Processing

These operations result in the creation of states W_0, W_1, W_2 and W_3. As shown in the figure, W_0, W_1 and W_2 are actually created multiple times. The dashed lines indicate the information used by the operator in each of the search steps. For example, when op_6 is applied to x_1, the operator uses the states y_1, y_2, and z_1 in its processing.

Figure 52 is an example of how goal processing functions in the same situation. Instead of op_6, a goal processing operator, gop_1, is applied to x_1, y_1, y_2, and z_1, in each case creating a new *goal state*, g_W, and connecting the original search states. g_W can be thought of as an abstract state that represents the problem solver's intention to extend states x_1, y_1, y_2, and z_1. Because of the characteristics of x_1, y_1, y_2, and z_1, this intention is similar for each and can be represented as a single goal state. Another goal processing operator, gop_W, is then applied to g_W. In this situation, gop_W can be a slightly modified op_6, as shown in Fig. 52.e, that combines op_6^1, op_6^2, op_6^3, and op_6^4 in a single operator. The application of gop_W to g_W results in the generation of W_0, W_1, W_2 and W_3. Again, the dashed lines represent the information used by gop_W.

This form of goal processing has several potential advantages. It may require significantly less work, or cost, than the interpretation search process discussed above and shown in Fig. 51. For example, the individual search operations op_6^1, op_6^2, op_6^3, and op_6^4 might have a much higher fixed overhead cost than gop_W. In addition, the individual search operations must redundantly search the database for inputs and many of the results that are produced are also redundant. It may be possible to avoid some of the costs associated with these redundant activities by using a single goal operator.

Furthermore, this form of goal processing allows a problem solver to reason about the goals themselves. Corkill and Lesser discuss the advantages of this capability in [4, 5, 6], and this work is further extended in [26, 27, 9]

Clearly, there are cases where this form of goal processing is not advantageous. For example, in situations where the individual search operations are not redundant or where the overhead of processing

operator.

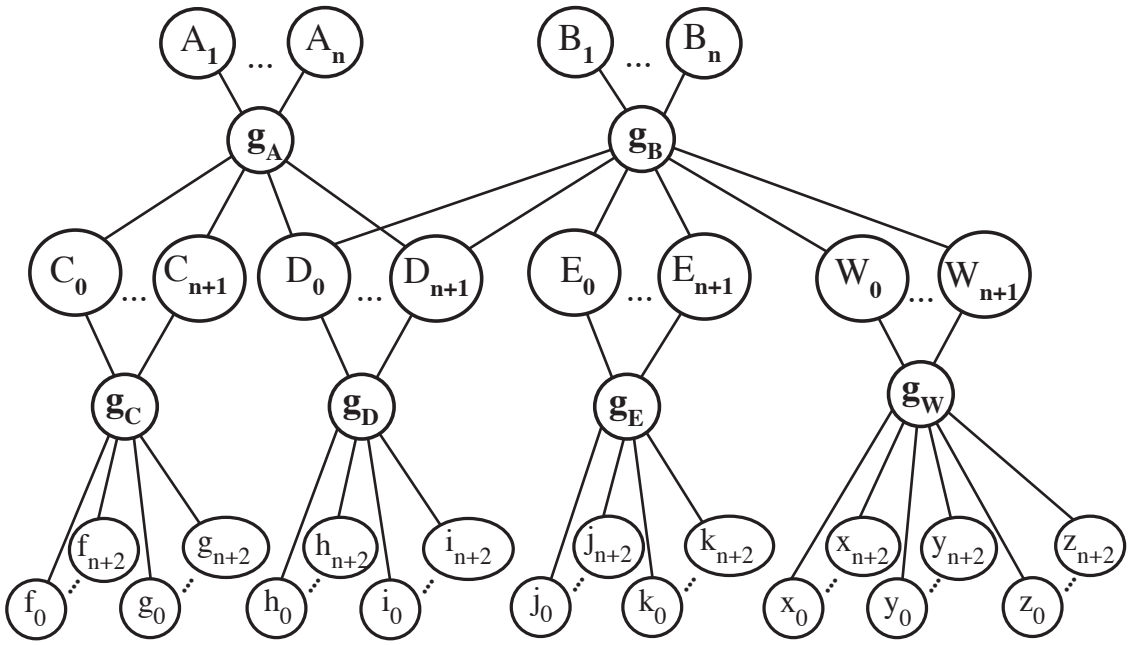


Figure 53: Representing Goal Processing in a Grammar

a goal is greater than the savings. Similarly, if the cost of an operation increases exponentially with the size of the input set, it may be better to implement a solution composed of numerous distinct search tasks where the cardinality of the input sets is limited.

The use of goal processing can be represented in the IDP framework by altering G'_n as shown in Fig. 53. In the transformed grammar, production rules are added using the meta-states g_A, g_B, etc. Each of these goal states is defined in terms of the result sets of the goals' children states. For example, g_W is the union of the elements of the partial result sets of $x_0 \ldots x_{n+2}, y_0 \ldots y_{n+2}$, and $y_0 \ldots y_{n+2}$ that can be generated with a single operator application. A similar definition is used to specify the other goals.

The operators that are applied to the goal states will be defined as the macro-operators from G'_n defined in Fig. 53. Thus, gop_W is applied to goal g_W to generate states $W_0 \ldots W_{n+1}$, gop_E is applied to g_E generate states $E_0 \ldots E_{n+1}$, etc. At the next level of interpretations, gop_A is applied to goal g_A to generate states $A_0 \ldots A_{n+1}$ and gop_B is applied to goal g_B to generate states $B_0 \ldots B_{n+1}$.

Given these definitions, it should be apparent that the use of goal processing does not effect the interpretations that are generated. Since the same operators, with identical inputs, are used to extend the goal states as were used in the original search, the results will be the same. However, the costs will differ. Notice that in the original example, op_6 was applied four times and in the second example with goal processing, gop_W was only applied once, but the goal processing operator was applied four times. Consequently, if the goal processing operator, gop_W, is significantly less expensive than op_6, then goal processing will offer distinct advantages.

Goal processing will be advantageous when the expected cost of connecting the goals plus the cost of generating the goals is less than the cost of connecting the search states without the use of goal processing. The cost of connecting the search states without goal processing is:

Equation 7.1 $cost(op_6^1) + cost(op_6^2) + cost(op_6^3) + cost(op_6^4)$.

(For the sake of simplicity, the cost of merging identical states will be ignored.)

Analysis of these costs will use models similar to the cost models from the DVMT [6]. In the DVMT, the cost of an operator application can be approximated by a constant factor plus a function of the number of inputs and the number of outputs. Specifically:

Equation 7.2 $\forall i, cost(op_i, n, m) = con_i + a * n^2 + b * m,$

where con_i is the constant cost of operator i, n is the number of inputs to op_i, m is the number of outputs generated by op_i, and a and b are cost coefficients.

Assuming that constant costs are 1, Equation 7.1 yields

$$(1 + 16 + 3) + (1 + 9 + 3) + (1 + 9 + 3) + (1 + 16 + 3) = 66. \quad (1)$$

For goal processing, the costs will be:

Equation 7.3 $cost(gop_1^1) + cost(gop_1^2) + cost(gop_1^3) + cost(gop_1^4) + cost(op_6^1).$

Assuming the goal processing operator conforms to Equation 7.2 and the constant cost of gop_1 is 1, Equation 7.3 yields

$$(1 + 1 + 1) + (1 + 1 + 1) + (1 + 1 + 1) + (1 + 1 + 1) + (1 + 16 + 4) = 33. \quad (2)$$

Consequently, in this simple example, goal processing results in a savings of 50%.

Though these figures are approximations, they are representative of the costs of DVMT search operators. The input component, n, of Equation 7.2 reflects the cost of retrieving data from the blackboard and the combinatorial nature of the reasoning processes used by DVMT operators [9, 6]. The output component, m, reflects the cost of writing data to the blackboard.

In this simple example, goal processing has clear performance advantages. However, it is still unclear as to when, in general, goal processing is effective and when it is detrimental to performance. For example, in the DVMT, an analysis indicated that goal processing is not always an effective tool [26, 27]. Subsequent work exploited this observation and resulted in significant performance improvements [8]. This work was specific to one aspect of goal processing in the DVMT domain, and left open questions regarding the general properties of domains where goal processing is useful. More specifically, this analysis did not consider the potential benefits of the *subgoaling* mechanisms described in[6].

In the example presented in this section, the principle difference in cost can be attributed to the fact that without goal processing, op_6 had to be applied four times to connect the low-level states. This is necessary because it is impossible to determine a priori whether or not the application of op_6 will result in the generation of a unique interpretation – i.e., an interpretation that will not be generated by any other application of op_6. In this example, the second application of op_6 generates a unique W_3. This is the result of inherent uncertainty in the form of missing data rules. If there were no missing data rules, the structure of the grammar would be such that op_6 would not have to be applied to every state.

A question that arises from this example is whether or not goal processing can be improved by simply applying op_6 to all the low level data simultaneously. This could be accomplished by creating a new meta-operator that would include all the possible applications of op_6. However, this would limit the problem solver's flexibility by forcing it to always apply op_6 to all the low-level data. This is a viable option for domains that do not offer possibilities for connecting goal states without applying

operators to extend them (i.e., domains where it is not possible to prune goal states). In other words, in domains where there is no opportunity for pruning the available operators that would extend a goal, it might be possible to find a control architecture more efficient than goal processing. In the DVMT, operators that extend goals are pruned under certain conditions. Furthermore, such pruning is done often enough that it is advantageous to use goal processing as described in [6].

8 A Basis for Analysis - An *Optimal Objective Strategy*

As discussed in Section 2, the analysis paradigms supported by the IDP/*UPC* framework all involve the use of four elements: a problem's structure and a problem solver's objective strategy, control architecture, and performance level (or behavior). In preceding sections, we presented the IDP formalism which uses a unified representation to describe a problem's structure and the abstractions and approximations used by a problem solver's control architecture. Furthermore, as was discussed in Section 2 and as will be discussed in the experimental sections which will follow, the performance of a problem solver will be measured in terms of the expected cost of problem solving and the expected probability that the problem solver will find the correct answer, where the correct answer is defined to be the highest rated interpretation. In this section we will define the fourth element, the objective strategy, that is needed in the analysis paradigms. In subsequent sections, we will show how this control architecture can be used with an expanded state representation that explicitly represents certain quantitative properties of a search space that are derived from an IDP specification. We will then experimentally verify the quantitative results determined analytically with the IDP formalism in Section 6.

8.1 Defining Optimal Interpretations

There are a variety of broad objective strategies that could be used as a basis for the analysis paradigms. For example, one class of objective strategies is related to finding *any* solution as quickly as possible, another class of strategies is related to finding the least cost solution. These, and other general objective strategies are discussed more formally in Appendix A.

We will refer to the strategy that we will define and use in subsequent analysis as the *optimal objective strategy*. This name is **not** intended to imply that this is the best possible objective strategy. Rather, it is intended to indicate that the goal of this strategy is to find the best possible solution using the least amount of computing resources. This definition is based on the definition of an interpretation problem given in Section 4.

As a basis for defining the optimal objective strategy, let an Optimal Interpretation be defined as follows:

Definition 8.1 *Optimal Interpretation, O* – Given a problem instance, x, from an IDP domain definition, I, that defines a set, C, of connected search spaces corresponding to correct interpretations of x, the optimal interpretation, O, is such that $\forall c \in C, O \in C, cost(O) \leq cost(c)$, where $cost(c)$ is the cost of applying the set of operators used to generate c.

Intuitively, an interpretation can be thought of as a set of operator applications that connects the search space and determines the highest rated explanation for the observed phenomena. Each of the different elements c of the set C from definition 8.1 represents a different set of operators or a different sequence of operator applications. The optimal interpretation is then the set of operator applications that connects the search space with minimum cost, and that always returns a correct answer.

In some situations, it will be necessary to analyze problem solving strategies that are not guaranteed to return the correct answer. To characterize these design parameters of problem solving systems, we will now define the notation for *allowed* or *expected error*, ϵ.

Definition 8.2 *Allowed Error,* ϵ – For a given IDP, P, the probability that any specific problem solving instance does not return the correct answer.

Thus, for an IDP with $\epsilon = 0.05$, the problem solver will return the correct answer with probability = 0.95. The other 5% of the time, the problem solver will return an answer that is not the "best" in terms of highest utility (or credibility). ϵ will be used extensively during analysis in situations where a control architecture eliminates certain search paths knowing that they might lead to the correct solution. This might be done in situations where the problem solver recognizes that the likelihood of the path leading to the correct solution is very low and where the differences between elements of a set of highly-rated interpretations is insignificant. In this situation, the problem solver is choosing to trade a limited number of incorrect solutions in order to reduce the expected cost of problem solving.

We will now restate the definition of Optimal Interpretation to include consideration of allowable error:

Definition 8.3 *Optimal Interpretation,* O – Given an instance of an IDP, P, that defines a set, I, of connected search spaces corresponding to interpretations of P with probability of correctness $= 1 - \epsilon$, the optimal interpretation, O, is such that $\forall i \in I, O \in I, cost(O) \leq cost(i)$.

An optimal interpretation is now defined to be the set of operator applications that return the correct answer with probability $= 1 - \epsilon$ and that connect the search space with minimal cost. Note that even in situations where the problem solver is allowed a certain amount of error, it is still required to connect the search space. Therefore, any problem solving actions that lead to a non-zero probability of returning an incorrect final solution must have explicit operations that eliminate from consideration some portion of the search space. Thus, correct solutions are in the areas of the search space that have been pruned. (Such a search space, i.e., a search space where the portion of the search space containing the correct solution has been pruned, can be thought of as *overconstrained*.)

In virtually all practical problem solving systems, ϵ plays an important role. In many real-world domains, the cost of exhaustive problem solving is prohibitively expensive. As a consequence, a very common strategy is to simply eliminate certain portions of the search space from consideration at the risk of eliminating the only search paths to a correct interpretation at the same time. Many of the control architectures that will be analyzed with the IDP/*UPC* formalism include allowable errors greater than 0.

8.2 Defining an Optimal Interpretation Objective Strategy

We will now use the definition of an optimal interpretation to define an objective strategy that will be used in subsequent analyses. It is important to note that the objective strategy we define here should not be considered the only possible objective strategy. Many other strategies can be defined and incorporated into the IDP/*UPC* analysis framework.

At any given stage in problem solving, the expected cost to reach termination is equivalent to the expected cost of connecting all open states. This is illustrated in Fig. 54 and will be represented as C throughout this paper. Each step of problem solving reduces C. A given search operator application, op_i, can reduce C in one of two ways. It can succeed in generating new states, which is analogous to

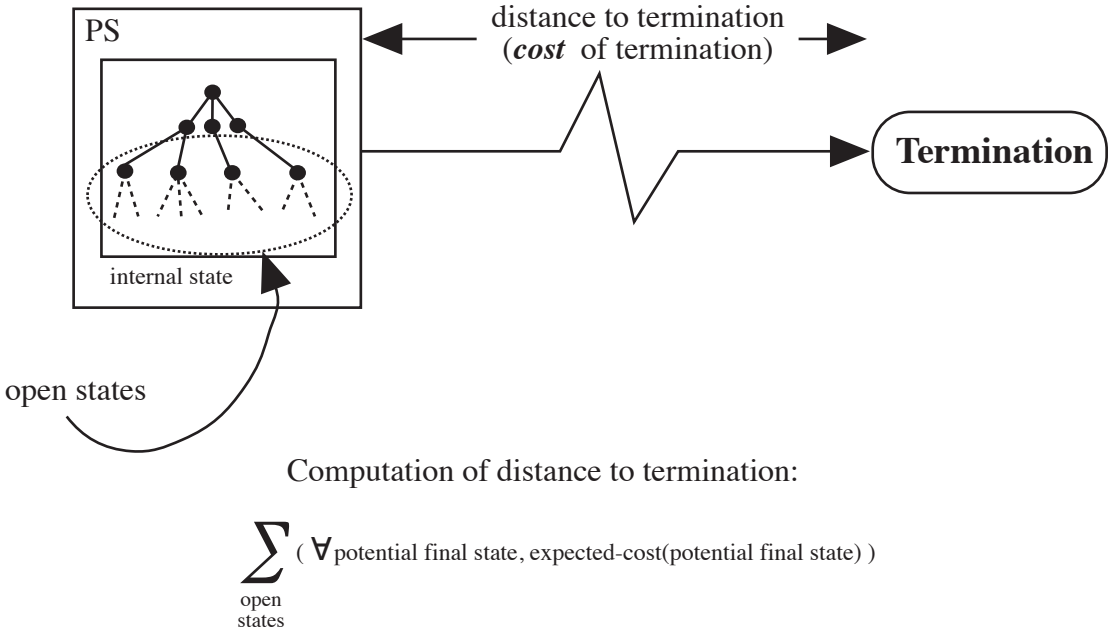


Figure 54: Computing the Distance to Termination, C

traversing search paths in ways that reduce the distance to any final states that can be reached along the paths. Alternatively, the operator can fail, causing all potential paths that required the failed operation to be eliminated from further consideration.

In the first instance, C is reduced to the degree that progress is made in traversing the paths. (The expected cost of path j will be represented as C_j and the degree to which operator op_i reduces C_j will be represented $c_j(op_i)$. The degree to which an operator reduces C will be represented $c(op_i)$.) In the second instance, C is reduced by an amount equal to the expected cost of fully expanding any potential paths that are eliminated.

In convergent search domains, a given operator typically constitutes a segment of multiple paths. In addition, a given operator application might result in the successful extension of some paths, which we will represent as the set E, and the failure of others, which we will represent as the set T. Consequently, the degree to which C is reduced by the application of operator op_i is given by:

Definition 8.4 *Amount operator op_i reduces the expected cost to connect all open states, C, is $c(op_i) = \sum_{\forall j \in E} c_j(op_i) + \sum_{\forall k \in T} C_k$*, where E is the set of paths extended by the application of op_i and T is the set of paths terminated by the application of op_i.

Intuitively, each $c_j(op_i)$ represents the degree to which a path to C_j is successfully extended, and each C_k represents the cost reduction associated with the failure of an attempt to extend path k. i.e., in the case of an unsuccessful path extension, the expected cost of problem solving is reduced by the entire distance remaining in path k.

Thus, *by definition*, the optimal objective strategy can be implemented by applying, at each step of problem solving, the operator, op_i, that, for all i, maximizes the average amount of search space connected per unit cost. Written formally, the choice of operator is made in order to maximize:

Equation 8.1 $\sum_i c(op_i) / \sum_i cost(op_i)$.

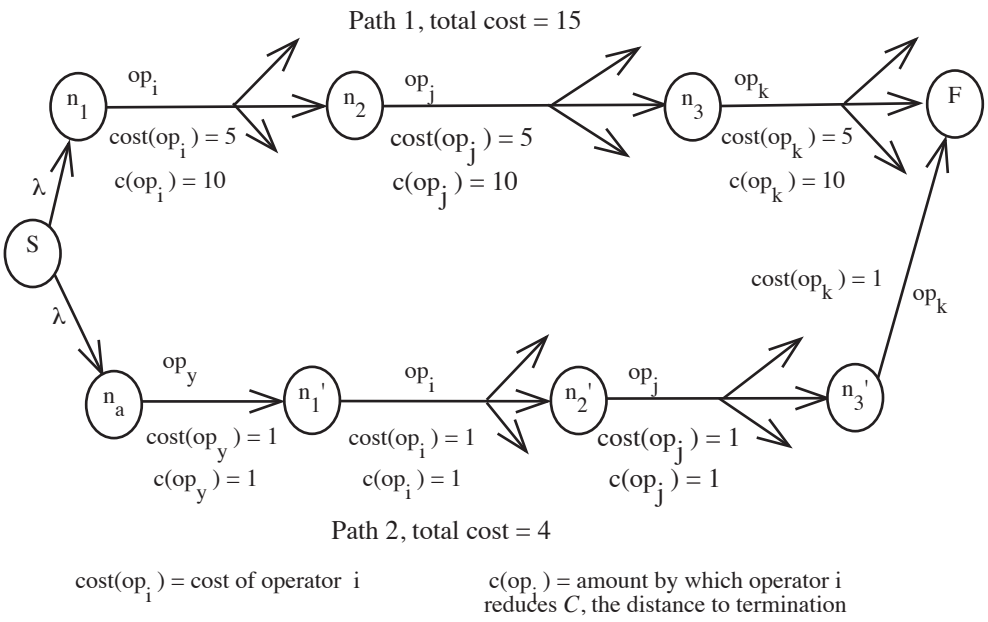


Figure 55: Example of the Non-local Effects of an Operator Application

Note that this represents the overall average amount of "search space connected" per unit of problem solving cost. Determining which operator to execute at *each step* of problem solving in order to achieve this maximum may be very difficult, even impossible. This is especially true in situations where the problem solver has only local information about a state. Even so, let us assume, for now, that the optimal objective strategy can be approximated by a control strategy that, at each step of problem solving, chooses the operator that maximizes:

Equation 8.2 $c(op_i)/cost(op_i)$.

Equation 8.2 is based on a perspective of problem solving that relies solely on *local* information. By local, we mean that this equation only considers the direct effects an operator has on C. It does not take into consideration the effects that an operator may have on other operators that are applied during subsequent processing, e.g., subproblem interactions involving cooperating or competing search paths.

For example, consider the situation shown in Fig. 55. In this figure, there are two alternative problem solving paths, one beginning with the state n_1 and the other beginning with the state n_a. In the first, the initial step of problem solving reduces C by 10 units, and it costs 5 units to apply, consequently, $c(op_i)/cost(op_i)$ = 2. The other steps along this path also reduce C by 10 units and have cost 5, resulting in similar ratios. (Assume that these costs are based on search paths not shown in the figure.)

In the second path, the initial search step has cost 1, but only reduces C by 1, resulting in a ratio of 1. This step will not be taken until all the operators in the first path have been applied. However, examining the second search path from a more comprehensive perspective results in the observation that the second path subsumes the first and, in fact, is less costly overall.

From a local perspective, it would appear that the first path is a better choice than the second path because, for every operator application in the first path, the ratio from Equation 8.1 is greater than the corresponding ratio for the first search step of path 2. However, subsequent steps of the second search

path somehow modify the search space so that the space can actually be connected more efficiently than it could in the first path.

This example is somewhat abstract and may not correspond closely to a specific domain, but it serves to illustrate a simple, but very important, principle. This principle is that, from a more comprehensive (or global) perspective, local optima will exist and maximizing Equation 8.2 locally will not result in globally optimal interpretations. However, the principles embodied in Equation 8.1 can be incorporated with a more comprehensive perspective to define globally optimal objective strategies. In Section 12 we will define the concept of *potential* and we will discuss how it can be used to overcome problems associated with local optima.

The definition of an optimal interpretation objective strategy will now form the basis for analyzing different control architectures. The analysis technique that will be used will involve formulating the abstractions and approximations used in alternative control architectures as IDP structures and then comparing the results of problem solving based on the grammars.

8.3 Local Control Issues - A Brief Discussion

The analytical framework that will be presented in the remainder of this paper is based on computational methods that are derived relative to a specific control strategy. To be precise, the analytical framework that will be described is based on the control strategy described in this section and extensions to it that incorporate the use of *potential*. (These extensions are presented in later sections.) It is important to note that, although it is necessary to specify a local control strategy in order to derive the analytical framework, the framework is not dependent on any single control strategy.

It is also interesting to note that in certain situations the choice of local control strategy does not matter. The framework will provide accurate analysis for *any* local control strategy. This will be true in situations where a problem solving system does not incorporate any problem solving actions that prune certain paths based on a dynamic perspective of problem solving. Because our definition of problem solving requires that a problem solver connect the entire search space in order to reach termination, problem solving systems that do not include any pruning operators essentially conduct an exhaustive search. In such a case, the choice of local problem solving strategy is irrelevant.

In situations where the problem solver has access to pruning actions that are based on predetermined criteria, the analysis tools can again be derived independent of the local control strategy. This is because the choice of problem solving activity will have no impact on the pruning operators. The order in which actions are executed will not affect the pruning criteria, and all actions susceptible to being pruned will eventually be subjected to the pruning criteria.

However, in situations where pruning criteria are determined dynamically, the order in which operators are applied is very significant. For example, in an extreme case, if all problem solving actions were applied before any pruning criteria were established, there would be no point to the pruning actions. From an intuitive perspective, it is advantageous to establish pruning criteria early in order to maximize the number of actions that are pruned before they are executed.

The computation of $E(C)$ presented in Section 6.1 assumes that dynamic pruning criteria are available before any pruning operators are executed. We consider this to be the definition of optimal processing in a domain with dynamic pruning operators. Achieving these results with an actual problem solving system requires a local control algorithm that approximates Equation 8.1. As will be shown in our experimental data presented in Sections 10.1, 10.2, and 12.4 we can implement local control algorithms that approximate the performance specified by Equation 8.1 using the *UPC* formalism and

the concept of *potential*.

9 Implementing IDP Based Problem Solving Systems – the *UPC* Model

Section 4 presented the IDP formalism and Sections 5 and 7 demonstrated how it can represent characteristics of a domain's problem structure. We will now begin to develop the analytical tools that will be used to explain and predict the effectiveness of specific control architectures that use abstractions. The essence of these analyses is that control architectures that use abstractions exploit problem structures and these structures, along with the abstractions used by the control component, can be represented naturally in the IDP formalism. In this section, another formalism, the *UPC* model, is introduced to extend the analytical power of IDP models. In the analysis framework that we are building, the *UPC* and IDP formalisms are closely linked – the IDP formalism models the structure of a domain theory and the *UPC* formalism maps IDP structures to a search space model where the structures are explicitly represented in a manner that can be used in a control architecture's evaluation function. This mapping is based on the local perspective of problem solving of a specific state in the search space and on statistical properties of the domain derived from the formal IDP description. It does not take into account the existence or absence of any other states. Using the *UPC* representation, we can construct problem solving systems capable of achieving the levels of performance predicted by quantitative analysis of IDP domain specifications and the optimal interpretation control strategy. As a consequence, domains with different structures can be compared using identical evaluation function based control architectures or these architectures can be varied to compare performance of different problem solvers within a given domain.

Though the IDP and *UPC* models are closely linked in the analysis framework presented in this paper, they can both be used as stand alone analysis tools. Consequently, the *UPC* formalism will first be presented as a general analysis tool, and subsequent sections will discuss the integration of the IDP and *UPC* models into a powerful framework for the analysis of sophisticated control. The initial presentation of the *UPC* as an independent analysis tool will include examples from IDP models and interpretation problems. This is intended to provide a frame of reference and simplify later sections. It is not intended to imply that the use of the *UPC* model is restricted to interpretation tasks or even that its use be in conjunction with the IDP model.

9.1 Overview of the *UPC* Formalism

The *UPC* model is an extension to the traditional search paradigm that explicitly represents important characteristics of search spaces that are often used implicitly in control architectures. *UPC* models explicitly represent where in a search space a given state lies relative to potential final states, the uncertainty associated with the ability to reach each final state, the expected cost of reaching the final state, and the final state's expected value, or *utility*[17]. The *UPC* information forms the basis for evaluation functions. In interpretation problems, this information can be derived from the specification of a domain theory's structure from Sections 4 and 5, in particular, the functions defining a domain theory's credibility and cost structures.

[17] In the IDP formalism, utility can be thought of as a state's credibility.

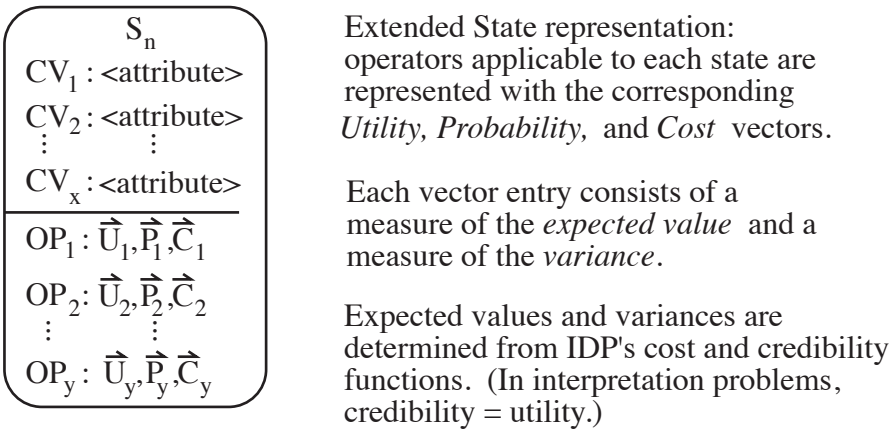


Figure 56: Representation of a Search State in the *UPC* Model

In a *UPC* model, a search space is defined by *characteristic variables*, or *CVs*, that specify the properties of individual states, operators that map (generate) one state to (from) another, and functions of CVs that define final states. The specification of operators is identical to the traditional notion of operators. The specification of the start state, intermediate states, and final states is similar except for an extension to their representations. The extension expands the set of CVs that characterize a state to include a set of vectors, $\{U, P, C\}$, that are characteristics of the operators that can be used to extend a state. Figure 56 is a representation of a search state based on the *UPC* model.

Intuitively, U, P, and C can be thought of as the characteristics that determine the desirability of expanding a given state. They represent the cost and utility structure of a space in a way that can be exploited by the control component to determine which state is the best to expand and which operator to use. The U, P, and C vectors associated with each operator can be thought of as the inputs to the evaluation function that orders problem solving activity. In addition, U, P, and C will be used to formalize control architectures that exploit problem structures such as *uncertainty*, *redundancy*, *interacting subproblems*, and *semi-monotonicity and bounding functions* (as defined in Section 7) within the search paradigm.

U, P, and C provide a map or coordinate system defining each state's location in the search space structure defined by the cost and credibility functions associated with the IDP's characteristic grammar, G. U specifies the direction of a search path (in terms of final states, or interpretations, they lead to) and P defines the probability that the path can be traversed successfully. Thus, every potential solution path[18] containing state s_n is represented by an entry in s_n's U vector that has a corresponding nonzero entry in s_n's P vector. For each path defined by s_n's U and P vectors, C defines *where* on the path s_n lies. If the C entry is small, then s_n is close to the final state. If the C entry is large, then s_n is far from the final state.

9.2 *UPC* States and the IDP Formalism

Using the *UPC* formalism, search states are created dynamically as problem solving operators are applied to existing states. The *UPC* vectors are determined dynamically, when a state is created, and included in the state's CV specification as previously described. In addition, each state corresponds

[18]A solution path is a derivation path that leads to an interpretation.

to an element from the IDP grammar. This correspondence is not necessarily one-to-one. For each element of the IDP grammar, there may be many different corresponding *UPC* states. Each of the states is differentiated by its distinct set of CVs which might include information such as "supporting data," "time of occurrence," "location," etc.

It is important to stress that the *UPC* representation accounts for multiple occurrences of a specific interpretation. In an actual run, multiple instances of an interpretation can be constructed resulting in distinct paths that must be accounted for when computing expected cost. Multiple instances of an interpretation result from ambiguities in a domain grammar that allow a single set of data to be interpreted in a variety of different ways. This is discussed at length in previous sections.

9.3 *UPC* Representation

The *UPC* model specifies that the following vectors are defined for every state in the search space. As shown in Fig. 56, the vector definitions for a given operator are a component of the corresponding state's CV set. Consequently, the *UPC* vectors are incorporated in definitions of the search space's structure.

U_{s_n} = set of *utility expectation* vectors. For each operator op_i that can be applied to state s_n, $U_{s_n}(op_i)$ defines a vector where each element is a value defining the *expected utility*[19] of the j^{th} final state that can be reached from s_n via a path beginning with operator op_i. This vector will also be represented as $u_{n,i}$. In terms of the IDP formalism, elements of the vectors in U_{s_n} correspond to high-level interpretations, T, that correspond to SNTs from the grammar that include the subtree s_n and the utility expectation can be thought of as T's expected credibility or a function of T's expected credibility. U_{s_n} is exhaustive in the sense that *all* final states, as defined by the SNTs of the grammar, that can be reached via a path including the states that are created by applying op_i to s_n are represented by an element in some $u_{n,i}(j) \in U_{s_n}$. In general, the use of SNTs in this role is artificial. The *UPC* values can be thought of as coordinates of a state's position in a search space relative to potential final states. The results that are derived here can also be derived without the concept of SNTs by using only the start state in the calculations. However, we have developed SNTs and use them in our calculations because they provide a more intuitive basis for understanding the values that are computed. The use of SNTs clarifies the representation somewhat for explanation purposes, and it greatly simplifies many system implementation issues (primarily debugging issues).

As defined in Section 4.2, the utility structure of a domain is defined recursively in terms of the production rules of a grammar. For the interpretation problems discussed in this paper, we are using the convention that a state's utility is a function of the utilities of its immediate descendants and the semantic function Γ. For this paper, we compute a state's credibility to be the average of these values. Thus, the expected utility of a given state *B*, and its variance, is derived from the expected utilities and variances generated by *B*'s RHS production rules, p_i, and by the expected value and variance of the function Γ_p. As described in Section 4.2, the utility (credibility) of a state generated by grammar rule i is computed by the function f_i.

More formally,

[19] It is important to note that the corresponding variance of the expected utility is a meaningful consideration within this formalism. However, for the sake of clarity, its significance will be discussed in a separate context. The variance associated with the other vectors, P and C will be similarly treated.

Definition 9.1 *Expected utility of a state* = $\sum_{i=1}^{n} \psi(i) * \mu_i$, where n = number of RHS productions for the state, $\psi(i)$ = the frequency associated with production i, and μ_i = expected value of the utility function, f_i, associated with production i.

These values are used to compute the expected utilities of the elements in U_{s_n}. For a given state, s_n, and operator, op_i, the expected utilities of the final states that can be reached along paths from s_n beginning with op_i are computed as discussed in Section 4.2 by using the actual utility (or credibility) of s_n and the expected utilities of other relevant states.

P_{s_n} = set of *conditional probability* vectors. $p_{n,i}(j)$ represents the likelihood that a path beginning with op_i can be constructed from s_n to the final state corresponding to the expected utility $u_{n,i}(j)$. Using the IDP model, elements of vectors in P_{s_n} can be determined from the distribution function ψ for the IDP's characteristic grammar, G, and from the expected distribution of domain events.

It is important to note the difference between U_{s_n} and P_{s_n}. The entries in the vectors P_{s_n} in no way indicate the 'worth' or 'utility' of a particular state or path. They only indicate the probability of reaching a specific final state via a path that includes s_n and begins with op_i. Thus, a path with a high-probability is not necessarily correct and a path with a low-probability is not necessarily incorrect.

Similarly, the entries in the vectors U_{s_n} do not specify the likelihood that a particular final state can be reached other than to indicate that the probability is nonzero, in which case the final state's utility is represented in a vector in U_{s_n}. Rather, the entries only represent the expected worth (or credibility) of a particular final state. Thus, a path with high-credibility does not necessarily have a high-probability and a path with low-probability does not necessarily have a low-probability.

The definitions of $u_{n,i}$ and $p_{n,i}$ lead to the following (Note: in these definitions, upper case letters indicate elements of a grammar's set of SNTs, which correspond to the final states in that associated search space.):

Definition 9.2 *The expected utility of a path from state n to final state j that begins with* op_i = $u_{n,i}(j) * p_{n,i}(j)$. i.e., The expected utility of a final state multiplied by the conditional probability of successfully reaching the final state given state n.

Definition 9.3 $p_{n,i}(j) = P(j \mid n) R_{prune}(n, j)$. The likelihood that a path exists from state n to final state j that begins with op_i is equal to the conditional probability that final state j can be generated given an n multiplied by a *pruning factor*, $R_{prune}(n, j)$, which represents the probability that the path is not pruned by a bounding function before final state j is generated.

The pruning factor, $R_{prune}(n, j)$, is only used in domains that include bounding operators that are based on dynamic pruning thresholds. In such domains, $R_{prune}(n, j)$ must be computed dynamically. In domains that do not include bounding operators, the pruning factor is 1. The computation of $R_{prune}(n, j)$ is based on a domain statistic, Ω, that represents the probability that a path to a state is pruned by a bounding function in a situation where the path can be created. Ω is defined for elements of the grammar that correspond to bounding functions, to

nonterminals that do not correspond to bounding functions, and to terminals. All definitions of Ω are based on the a priori computations for the expected values of the credibilities of the elements of a grammar. For nonterminal element s that does not correspond to a bounding function, the definition of Ω is:

Definition 9.4 $\Omega(s) = \Sigma_i \psi(i) * \Pi_j \Omega(n_j)$. Where each i corresponds to a production rule with s as a left-hand-side and each n_j is an element of the right-hand-side of the production rule.

For nonterminal element s that corresponds to a bounding function, the definition of Ω is:

Definition 9.5 $\Omega(s) = (1 - P(credibility(n) < t) * \Omega(n)$. Where n is the right-hand-side of the production rule, t is the pruning threshold for the bounding function, and $credibility(n)$ is the expected credibility of grammar element n.

For terminal element s, $\Omega(s) = 1$.

The computation of $R_{prune}(n, j)$ is based on the credibility of n and the values of Ω that are computed for the grammar. Formally,

Definition 9.6 $R_{prune}(n, j) = \Omega(j, n)$. Where $\Omega(j, n)$ in the computation of $\Omega(j)$ with respect to n.

The computation of $\Omega(j, n)$ is straightforward, but it must be done dynamically using the actual credibility of n instead of the a priori expected credibility.

Definition 9.7 $\Omega(s) = \frac{\Sigma_i \psi(i) * \Pi_j \Omega(x_j, j)}{\Sigma_i \psi(i)}$. Where each i corresponds to a production rule with s as a left-hand-side that includes n in a derivation tree, each x_j is an element of the right-hand-side of the production rule. The sum, $\Sigma_i \psi(i)$ is used to normalize the computation relative to n. When computing $\Omega(x_j, j)$, if n is not included in any of the interpretation trees for x_j, the value $\Omega(x_j)$ is used.

In general, conditional probabilities can be computed by:

Definition 9.8 $P(A \mid b) = \frac{P(A \cap b)}{P(b)}$. Conditional probability of state A given state b, where b is a descendant of A (i.e., b is on the RHS of some set of grammar rule applications that begin with A on the LHS). This is the conventional definition of conditional probability that is available in any appropriate textbook.

For IDP models, the following equations can be used to determine conditional probabilities:

Definition 9.9 $P(A) = P(S \rightarrow A)$[20]*, domain specific distribution functions.* Probability of the domain event corresponding to interpretation A occurring. This probability will be specified with domain specific distribution functions. In general, these distributions will be represented with production rules of the grammar associated with the start symbol. The RHSs of these rules will be from the grammar's set of SNTs. Uncertainty regarding this distribution leads to problem solving uncertainty.

[20]The notation used in these equations, i.e., $S \rightarrow A$ is distinct from the production rule notation used to designate grammar rules and should not be confused as production rule notation.

Definition 9.10 $\{RHS(A)\}$ = the set of elements that appear on right-hand-sides of production rules with A on the left-hand-side.

Definition 9.11 $P(b \in \{RHS(A)\}^{\vdash +}) = \sum_{\forall i} P(b \in RHS_i(A)) + \sum_{\forall r'}(P(r' \in \{RHS(A)\}) * P(b \in \{RHS(r')\}^{\vdash +}))$, where $\{RHS(A)\}$ is the set of all RHSs of A, $RHS_i(A)$ is the RHS of the i^{th} production rule of A, $P(b \in RHS_i(A)) = \psi(RHS_i(A))$ if $b \in RHS_i(A)$, 0 otherwise, and each element r' is a nonterminal that appears in a RHS of A that does not also include b. The probability of partial interpretation b being included in any RHS of A, as defined by the distribution function $\psi(A)$. The "$\vdash +$" notation indicates that the definition of RHS is recursive. i.e., $\{RHS(A)\}^{\vdash +}$ represents the transitive closure of all states that can be generated from A. Thus, b can be in an RHS of A, or in the RHS of some element of an RHS of A, etc.

Definition 9.12 $P(b) = P(S \rightarrow b) = \sum_{\forall A} P(A) * P(b \in \{RHS(A)\}^*)$. Probability of partial interpretation b being included in an interpretation.

Definition 9.13 $P(A \rightarrow b) = P(A) * P(b \in \{RHS(A)\}^*)$. Probability that the partial interpretation, b, is generated from full or partial interpretation A, where b is a descendant of A.

Definition 9.14 *Ambiguity* – Given a domain event, A, its interpretation is *ambiguous with* the interpretation of a second domain event, B, when B *subsumes* A (the subsume relationship is specified in Definition 5.16 in Section 5). i.e., A is ambiguous with B when B $\Rightarrow$ A. (The low-level signal data generated by B can be mistaken for an A.) Note that this definition of ambiguity is *not* reflexive. Thus, A being ambiguous with B does not imply that B is ambiguous with A. This definition of ambiguity is consistent with Definition 5.1 and will be used where appropriate.

Definition 9.15 $P(A \cap b) = P(A \rightarrow b) + \sum_{\forall B} P(B \rightarrow b)$. Intersection of domain events A and b, where b is a descendant of A, and where the interpretation of A is *ambiguous with* the interpretation of each B. The intersection of A and b will occur when both A and b are generated during the course of a specific problem solving instance. This will occur when A leads to the generation of b and when the occurrence of a distinct event, B, leads to the generation of b and when A is ambiguous with B. In the case where B leads to the generation of b, b and A still intersect because an A will be generated during processing since A is ambiguous with B.

C_{s_n} = set of *cost* vectors. $c_{n,i}(j)$[21] is a pair of values, represented $c_{n,i}(j,1)$ and $c_{n,i}(j,2)$. The first is the expected cost of generating the path, *when the path can be generated*, from s_n, beginning with op_i, to the final state corresponding to $u_{n,i}(j)$. The second is the expected cost of extending the path *when the final state cannot be reached.* These two values can be used to specify the expected cost of *attempting* to complete a path. Formally,

Definition 9.16 For state s_n, the expected cost of the path corresponding to $u_{n,i}(j)$ is $p_{n,i}(j) * c_{n,i}(j,1) + (1 - p_{n,i}(j)) * c_{n,i}(j,2)$.

[21] We will use the subscript n to represent s_n in order to simplify the notation.

Definition 9.17 The expected cost of a "correct" path = $c_{n,i}(j,1) = E(cost(F)) - E(cost(s_n))$, where F = $u_{n,i}(j)$ – the final state the path is attempting to reach and $E(cost(s_n))$ represents the expected cost of generating state s_n. (Using the notation from Section 8, cost functions would be represented $g(u_{n,i}(j)) - g(s_n)$.) Thus, the estimated cost of reaching the final state (completing the interpretation tree, T) is a function of the cost of deriving the entire interpretation tree minus the cost already incurred to derive the subtree corresponding to s_n.

Definition 9.18 The expected cost of a specific "incorrect" path = $C_{\neg F,R}(s_n) = \sum_{\forall m} P((n \cap m) \mid R) * (E(cost(s_m)) - (\sum_{\forall t} E(cost(s_t)) * P(s_m \rightarrow s_t)))$, where $E(cost(s_m))$ is the expected cost of generating state s_m, the set m comprises the components of F that are not also components of s_n, and the set t comprises the components of each s_m. The term $E(cost(s_m)) - (\sum_{\forall t} E(cost(s_t)) * P(s_m \rightarrow s_t))$ is the incremental cost of each of the components of F. Thus, this definition takes into consideration the probability of each component of F being generated and the incremental cost of each component.

Definition 9.19 *The expected cost of connecting all "incorrect" paths from a state* = $c_{n,i}(j,2) = \sum_{\forall R} \frac{P(R \rightarrow n)}{(P(n) - P(F \cap n))} * C_{\neg F,R}(n)$, where F represents the final state corresponding to $u_{n,i}(j)$, $C_{\neg A,R}(n)$ is the *expected cost* of a failed attempt to generate a path to F given that R is the correct interpretation, R is an element of the set SNT, and where F is not ambiguous with R. Intuitively, $C_{\neg F,R}$ represents the expected cost the problem solver will incur before determining that a path cannot be generated to F. It is necessary to differentiate each R, since the cost of a failed path to F will vary with the different R. For example, in some situations, the cost of a failed path to F may be very small. This may occur when interpretations of "B" are correct. In contrast, the cost of a failed path to F may be very large when the correct interpretation is "C."

In the case where the path does not exist, the expected cost will be dependent on the structure of the domain. In domains with a great deal of ambiguity, this value could be almost as large as (or perhaps much larger than) the expected cost of a correct path. Examples of computing expected costs for paths are presented in Section 9.4.

9.4 Determining *UPC* Vector Values

In this subsection, examples of *UPC* vector determination will be presented. These examples will be based on the original interpretation grammar, G', reproduced in Fig. 57, and the modified interpretation grammar with added rules for noise and missing reproduced in Fig. 60. In this representation, the SNTs of the grammar are A, B, M, N, and O. Note that in this grammar there are no pruning operators, so the computations presented here ignore the computation of the pruning factor, R_{prune}.

9.4.1 *UPC* Vector Values in a Simple Grammar

Figure 58 shows the *UPC* values for two low-level states, h and f, from the interpretation grammar shown in Fig. 15. The grammar is reproduced in Fig. 57 for convenience. In the figure, the subscripts indicate which $SNTs$ the *UPC* values are associated with. For computing the *UPC* values for h, we will assume that h is the only state created so far. We make a similar assumption when computing *UPC* values for f. There is only a single operator available to extend each state and, as a result, there are

Interpretation Grammar G'

0. S → A | B	2. B → DEW
1. A → CD	4. E → jk
3. C → fg	6. W → xyz
5. D → hi	8. j → (signal data)
7. f → (signal data)	10. k → (signal data)
9. g → (signal data)	12. x → (signal data)
11. h → (signal data)	14. y → (signal data)
13. i → (signal data)	15. z → (signal data)

Figure 57: Search Operators Defined by Interpretation Grammar G'

only single U, P and C vectors for each state, as shown in Fig. 58. For state h, the available operator is op_5, which is represented as "D → hi" in Fig. 15. For state f, the available operator is op_3, which is represented as "C → fg."

To simplify the computations used in the next two sections, we will limit the discussion of calculating expected utilities. Thus, for all examples in the next two sections, *UPC* values will always reflect an expected utility of 1.

Intuitively, the assumptions used in the next two sections can be thought of as follows. The low-level state h can be used to derive two interpretations, an A or a B. In this example, the utility of either interpretation will be represented as "1." This is manifested in $u_{h,5}$ as two entries, both equal to 1.

$u_{h,5} = (1_A, 1_B)$	$p_{h,5} = (0.5_A, 0.5_B)$	$c_{h,5} = ((6,3)_A, (10,3)_B)$
$u_{f,3} = (1_A)$	$p_{f,3} = (1_A)$	$c_{f,3} = ((6,0)_A)$

Figure 58: *UPC* Vectors for States from Search Space Defined by G'

By definition, the entries in $p_{h,5}$ correspond to the conditional probabilities of generating paths from state h to each of the final states represented in $u_{h,5}$. Given no prior information about the distribution of domain events corresponding to interpretations of A and B, it will be assumed that the distribution is split evenly between them. Consequently, $P(A) = P(B) = 0.5$. Now the entries in the vector $p_{h,5}$ can be determined.

$p_{h,5}(1)$ is the probability that a path can be constructed from h, beginning with op_5, to final state A. From Definition 9.8,

$$p_{h,5}(1) = P(A \mid h) = \frac{P(A \cap h)}{P(h)}. \tag{3}$$

In G', $P(A \cap h) = P(A \rightarrow h)$, since A is not ambiguous with any other interpretations. So, from Definitions 9.15, 9.13, and 9.11,

$$P(A \cap h) = P(A \rightarrow h) = P(A) * P(h \in \{RHS(A)\}^*) = 0.5 * 1 = 0.5. \tag{4}$$

Note that in this case, h is on the RHS of D with

$$P(h \in \{RHS(D)\}) = 1, \tag{5}$$

and D is in the RHS of A with

$$P(D \in \{RHS(A)\}) = 1. \tag{6}$$

Thus,

$$P(h \in \{RHS(A)\}^*) = 1 * 1 = 1. \tag{7}$$

From Definition 9.12,

$$P(h) = P(h \in \{RHS(A)\}^*) + P(h \in \{RHS(B)\}^*) = 0.5 * 1 + 0.5 * 1 = 1. \tag{8}$$

h is included in an RHS of A or B (recursively) with probability 1, and $P(A) = P(B) = 0.5$.

Thus,

$$P(A \mid h) = \frac{0.5}{1} = 0.5. \tag{9}$$

This is shown in Fig. 58 as $p_{h,5}(1)$.

The computation of $p_{h,5}(2)$, the probability that a path can be constructed from h, beginning with op_5, to final state B, is similar and yields the same result.

Given the utility and probability vectors shown in Fig. 58, it is unclear whether the partial interpretation h is part of an A or a B. To differentiate which of the two events occurred, the problem solver must continue to interpret the data by extending partial interpretation h. If the correct interpretation generated from h is an A, then the data corresponding to B's component set (component sets are defined in Section 5.4.1) will not be generated unless the data are also in A's component set. Conversely, if the correct interpretation is B, then partial interpretations corresponding to A's component set will not be formed unless they are also in B's component set.

To compute the expected cost vectors, we will let the cost of each production rule be 1. The generation of an A requires that 7 production rule operators be executed, each at a cost of 1. However, the existence of an h implies that the cost of generating an h does not have to be incurred again. Consequently, the expected cost to generate an A, when A is the correct interpretation, is

$$c_{h,5}(1,1) = 7 - 1 = 6. \tag{10}$$

Similarly, the expected cost to generate a B, when B is the correct interpretation, is

$$c_{h,5}(2,1) = 11 - 1 = 10. \tag{11}$$

In situations where the correct interpretation is a B, the cost of attempting to generate a path to A, starting with op_5, is, by Definition 9.19,

$$c_{h,5}(j,2) = \sum_{\forall R} \frac{P(R \rightarrow h)}{(P(h) - P(A \cap h))} * C_{\neg A,R} \tag{12}$$

where A is not ambiguous with R.

Therefore,

$$c_{h,5}(1,2) = \frac{P(B \rightarrow h)}{P(h) - P(A \cap h)} * C_{\neg A,B} = \tag{13}$$

$$\frac{0.5}{0.5} * (3) = 3. \tag{14}$$

$C_{\neg A,B}(h)$, the *expected cost of a failed attempt to generate a path from h to A when B is the correct interpretation,* is the cost of generating a D, since $P(D \mid h) = 1$, plus the cost of attempting to generate an f or g. When B is the correct interpretation, attempting to construct a path to A will fail after a cost of 3 is incurred. This cost will be associated with generating a D (cost of 2) with the application of op_5, which will be successful, and the cost of trying to generate a C, which will be unsuccessful. The attempt to generate a C will fail when the problem solver attempts to generate an f or a g. For now, we will assume that the cost of such a failure is 1. After failing to generate an f or g, we will also assume that the problem solver suspends its attempt to generate a C, and, as a result, its attempt to generate an interpretation of A.

Thus, the expected cost of attempting to generate a path to final state A, given an h, is, from Definition 9.16,

$$p_{h,5}(1) * 6 + (1 - p_{h,5}(1)) * 3 = 4.5. \tag{15}$$

The computation of $c_{h,5}(2,2)$ is similar and the expected cost of attempting to generate a path to final state B, given an h, is

$$p_{h,5}(2) * 10 + (1 - p_{h,5}(2)) * 3 = 6.5. \tag{16}$$

In contrast to h, the low-level state f can only be used to generate an A. Consequently, f's utility vector has only a single entry corresponding to an A. Since there is no other possible interpretation of an f, the probability of reaching the final state associated with an interpretation of A is 1.0. This value can also be computed from Definitions 9.8 through 9.12. Finally, the cost vector has a single entry corresponding to the interpretation of A and this is computed as previously described,

$$cost(A) - cost(f) = 6. \tag{17}$$

Since a path always exists from state f to an A, the cost of failing to reach A is 0. Consequently, $c_{f,3}(1,1) = 6$ and $c_{f,3}(1,2) = 0$.

Given an h and an f with these *UPC* values, it should be noticed that f is not included in any ambiguous interpretations. As can be seen from f's *UPC* values, this simplifies problem solving greatly. In this domain, given an f, a problem solver can postulate an interpretation of A without conducting any additional problem solving. However, given an h, the problem solver must still differentiate the possible interpretations A and B.

9.4.2 *UPC* Vector Values with Noise and Missing Data

Now consider the *UPC* values for h and f given the domain theory represented by the grammar shown in Fig. 59. (This grammar is shown graphically in Fig. 60.) This grammar is identical to G' but rules have been added corresponding to noise and missing data. As a result of these rules, the *UPC* values for h and f are as shown in Fig. 61. As with the previous example, these values were computed based on the assumption that they were the only states created so far.

Again, final states will be assigned utility 1. op_5 can lead to two final states, A and B. This is indicated in $u_{h,5}(1)$ and $u_{h,5}(2)$, respectively. Likewise, op_{17} can lead to an M and op_{21} can lead to an O. This is represented in $u_{h,17}(1)$ and $u_{h,21}(1)$.

1. A → CD	7. f → (signal data)	16. M → Y
2. B → DEW	8. j → (signal data)	17.0 Y → qr
3.0 C → fg	9. g → (signal data)	17.1 Y → qhri
3.1. C → fgq	10. k → (signal data)	18. N → Z
4. E → jk	11. h → (signal data)	19. Z → xy
5.0 D → hi	12. x → (signal data)	20. O → X
5.1. D → rhi	13. i → (signal data)	21.1. X → fgh
6.0 W → xyz	14. y → (signal data)	21.2. X → fg
6.1. W → xy	15. z → (signal data)	

Interpretation Grammar G' with noise and missing data rules

Figure 59: G' with Added Noise and Missing Data Rules

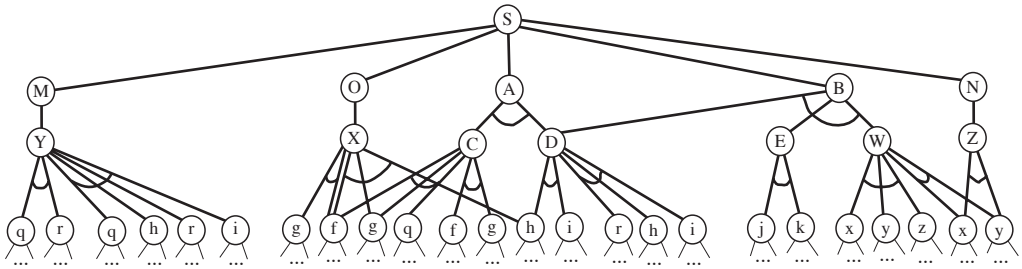


Figure 60: Graphical Representation of G'

$u_{h,5} = (1_A, 1_B)$	$p_{h,5} = (0.33_A, 0.33_B)$	$c_{h,5} = ((7,3)_A, (10,3)_B)$
$u_{h,17} = (1_M)$	$p_{h,17} = (0.25_M)$	$c_{h,17} = ((5,1)_M)$
$u_{h,21} = (1_O)$	$p_{h,21} = (0.5_O)$	$c_{h,21} = ((4,1)_O)$
$u_{f,3} = (1_A)$	$p_{f,3} = (0.5_A)$	$c_{f,3} = ((7,3)_A)$
$u_{f,21} = (1_O)$	$p_{f,3} = (1.0_O)$	$c_{f,3} = ((3.75,0)_O)$

Figure 61: *UPC* Vectors for Two States from Interpretation Grammar G'

To generate the conditional probability vectors, it was again assumed that the domain events that correspond to the interpretations A, B, M, N, and O are evenly distributed, i.e., $P(A) = P(B) = P(M) = P(N) = P(O) = 0.2$. Furthermore, for this example, the ψ distribution for the possible RHSs of the nonterminals Y, X, C, D, and W will be 0.5. For example, ψ(Y→qr) = 0.5 and ψ(Y→qhri) = 0.5.

Given this information and from the previous definitions, the probability that a path from h beginning with op_5 can reach final state A is

$$p_{h,5}(1) = \frac{P(A \cap h)}{P(h)}. \tag{18}$$

In this grammar, $P(A \cap h) = P(A \rightarrow h)$, since A is not ambiguous with any other interpretations.

$$\begin{aligned} P(A \rightarrow h) &= P(A) * P(h \in \{RHS(A)\}^*) = 0.2 * 1 = 0.2 && (19)\\ P(h) &= P(h \in \{RHS(A)\}^*) + P(h \in \{RHS(B)\}^*) + P(h \in \{RHS(M)\}^*) + \\ &\quad P(h \in \{RHS(O)\}^*) && (20)\\ &= (0.2 * 1) + (0.2 * 1) + (0.2 * 0.5) + (0.2 * 0.5) = 0.6 && (21) \end{aligned}$$

Thus,

$$p_{h,5}(1) = \frac{0.2}{0.6} = 0.33. \tag{22}$$

The computation of $p_{h,5}(2)$, the probability that final state B can be reached, is similar and also yields 0.33.

The probability that a path from h beginning with op_{17} can reach final state M is

$$p_{h,17}(1) = \frac{P(M \cap h)}{P(h)}. \tag{23}$$

From Definition 9.13,

$$P(M \cap h) = P(M \rightarrow h) + P(A \rightarrow \{qrh\}) = \tag{24}$$

$$P(M \rightarrow h) + P(A) * P(q \in \{RHS(A)\}^*) * P(r \in \{RHS(A)\}^*) * P(h \in \{RHS(A)\}^*), \tag{25}$$

since M is ambiguous with A when the domain event A results in the generation of a q and an r in addition to an h.

$$P(M \rightarrow h) = P(M) * P(h \in \{RHS(M)\}^*) = 0.2 * 0.5 = 0.1 \tag{26}$$

and

$$P(A \rightarrow \{qrh\}) = \tag{27}$$

$$P(A) * P(r \in \{RHS(A)\}^*) * P(q \in \{RHS(A)\}^*) * P(h \in \{RHS(A)\}^*) = \tag{28}$$

$$0.2 * 0.5 * 0.5 * 1.0 = 0.05. \tag{29}$$

Thus, using $P(h)$ calculated above,

$$p_{h,17}(1) = \frac{0.1 + 0.05}{0.6} = 0.25. \tag{30}$$

The computation of $p_{h,21}(1)$, which is the probability that a path from h beginning with op_{21} can reach final state O, must take into account the fact that the interpretation of an O is always ambiguous with the interpretation of an A, i.e., A $\Rightarrow$ O. (This is true because f and g are always components of $\{RHS(A)\}^*$ and f and g are the terminal symbols that lead to the interpretation of an O.) Thus,

$$P(O \mid h) = \frac{P(O \cap h)}{P(h)} = \frac{P(O \rightarrow h) + P(A \rightarrow h)}{P(h)}, \tag{31}$$

$$P(O \rightarrow h) + P(A \rightarrow h) = \tag{32}$$

$$P(O) * P(h \in \{RHS(O)\}^*) + P(A) * P(h \in \{RHS(A)\}^*) = \tag{33}$$

$$0.2 * 0.5 + 0.2 * 1 = 0.3 \tag{34}$$

Thus,

$$p_{h,21}(1) = \frac{0.3}{0.6} = 0.5. \tag{35}$$

The expected costs shown for h take into account the distribution function, ψ. When h is included in the derivation of an A or a B, the expected cost varies depending on the distribution of the RHSs for the partial interpretations C, D, and W. The RHSs of D are "hi" and "rhi," the RHSs of C are "fgq" and "fg," and the RHSs of W are "xyz" and "xy." For this example, let

$$\psi(D) = \begin{cases} 0.5 & \text{for D} \rightarrow \text{hi} \\ 0.5 & \text{for D} \rightarrow \text{rhi} \end{cases} \tag{36}$$

$$\psi(C) = \begin{cases} 0.5 & \text{for C} \rightarrow \text{fg} \\ 0.5 & \text{for C} \rightarrow \text{fgq} \end{cases} \tag{37}$$

$$\psi(W) = \begin{cases} 0.5 & \text{for W} \rightarrow \text{xyz} \\ 0.5 & \text{for W} \rightarrow \text{xy} \end{cases} \tag{38}$$

The interpretation trees for A and B that are derived from these rules are shown in Fig. 62. There are four distinct interpretation trees for both A and B that include h in their component sets. In general, such derivations could be semantically different and, as a result, could constitute different final states. In this example, all interpretations of A or B are considered to be represented by the same final state. Thus, there is a single entry in h's *UPC* vectors for a final state corresponding to an A and, likewise, one for a B.

Given the distributions described above, the conditional probability of each of these interpretation trees being correct given an A or a B, respectively, is 0.25 (i.e., $0.5 * 0.5 = 0.25$), and the expected derivation cost for an A from state h is

$$c_{h,5}(1,1) = (0.25 * 7) + (0.25 * 8) + (0.25 * 8) + (0.25 * 9) - 1 = 7. \tag{39}$$

Similarly, the expected derivation cost for a B from state h is

$$c_{h,5}(1,1) = (0.25 * 11) + (0.25 * 10) + (0.25 * 12) + (0.25 * 11) - 1 = 10. \tag{40}$$

The expected costs when these final states cannot be reached are specified by Definition 9.19

$$c_{h,5}(1,2) = \sum_{\forall R} \frac{P(R \rightarrow h)}{(P(h) - P(A \cap h))} * C_{\neg A,R} \tag{41}$$

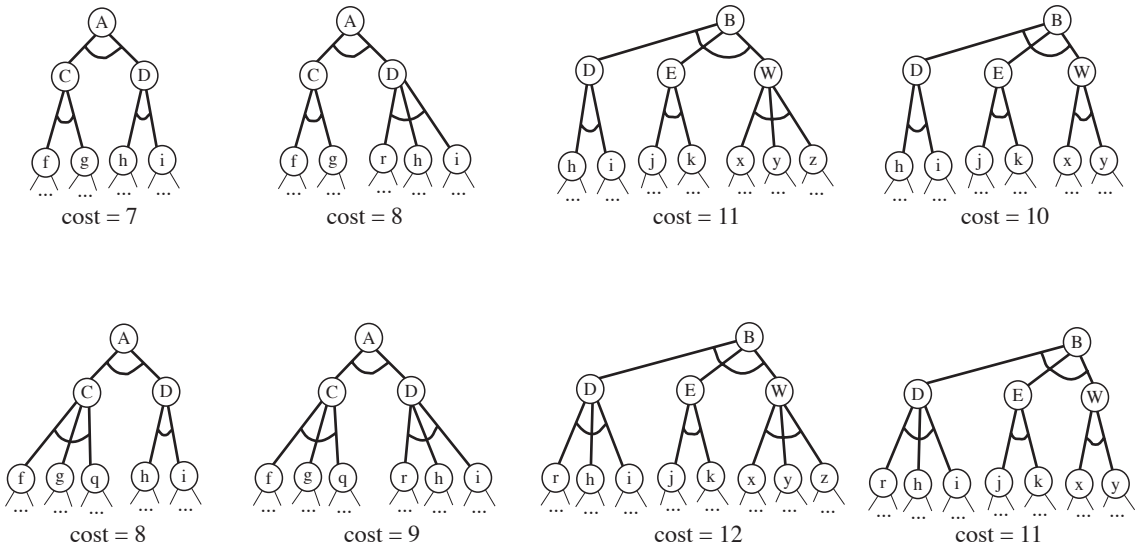


Figure 62: Interpretation Trees for Domain Events A and B

where A is not ambiguous with R.

In the case where a path to A cannot be generated and B is correct, op_5 will successfully generate a D, but will fail to generate an f or g. The cost of generating the D is 2, with probability 0.5, or 3, also with probability 0.5. This is specified by

$$\psi(D) = \begin{cases} 0.5 & \text{for D} \rightarrow \text{hi} \\ 0.5 & \text{for D} \rightarrow \text{rhi} \end{cases} \tag{42}$$

In the case where the RHS is "hi," the cost of generating a D will be 2; the cost of generating an i plus the cost of generating a D, or $1 + 1 = 2$. In the case where the RHS is "rhi," the cost of generating a D will be 3; the cost of generating an r, plus the cost of generating an i, plus the cost of generating a D, or $1 + 1 + 1 = 3$. The cost of failing to generate g or f is 1. Thus, the cost of failing to generate a path to A given B is

$$0.5 * 2 + 0.5 * 3 + 1 = 3.5. \tag{43}$$

In the case where M is true, op_5 will again succeed, as before, and the path will fail when an attempt is made to generate an f or g. However, when M is correct the cost of generating a D is 3. This is due to the fact that h only appears on the RHS of M with an r, a q, and an i. Therefore, the cost of failing to generate a path to A when M is the correct interpretation is

$$3 + 1 = 4.0. \tag{44}$$

In the case where O is is true, op_5 will fail, at a cost of 1, and the cost of failing to generate a path to A when O is the correct interpretation is 1.

Therefore, from Definition 9.19,

$$c_{h,5}(1,2) = \frac{P(B \rightarrow h)}{(P(h) - P(A \cap h))} * (3.5) + \frac{P(M \rightarrow h)}{(P(h) - P(A \cap h))} * (4) + \frac{P(O \rightarrow h)}{(P(h) - P(A \cap h))} * (1.0) = \quad (45)$$

$$\frac{0.2}{(0.6 - 0.2)} * (3.5) + \frac{0.1}{(0.6 - 0.2)} * (4) + \frac{0.1}{(0.6 - 0.2)} * (1.0) = \quad (46)$$

$$(0.5 * 3.5) + (0.25 * 4) + (0.25 * 1.0) = 3. \quad (47)$$

The expected cost of attempting to generate a path to final state A, given an h, is, from Definition 9.16,

$$p_{h,5}(1) * 7 + (1 - p_{h,5}(1)) * 3 = 0.33 * 7 + 0.67 * 3 = 4.33. \quad (48)$$

Computing the expected cost of attempting to generate a path to final state B, given an h, requires computing $c_{h,5}(2,2)$. The expected costs that will be incurred for failed paths are the same as for A, so

$$c_{h,5}(2,2) = \frac{P(A \rightarrow h)}{(P(h) - P(B \cap h))} * (3.5) + \frac{P(M \rightarrow h)}{(P(h) - P(B \cap h))} * (4) + \frac{P(O \rightarrow h)}{(P(h) - P(B \cap h))} * (1.0) = \quad (49)$$

$$\frac{0.2}{(0.6 - 0.2)} * (3.5) + \frac{0.1}{(0.6 - 0.2)} * (4) + \frac{0.1}{(0.6 - 0.2)} * (1.0) = \quad (50)$$

$$(0.5 * 3.5) + (0.25 * 4) + (0.25 * 1.0) = 3. \quad (51)$$

Therefore, the expected cost of attempting to generate a path to final state B, given an h, is,

$$p_{h,5}(2) * 10 + (1 - p_{h,5}(2)) * 3 = 0.33 * 10 + 0.67 * 3 = 5.33. \quad (52)$$

Computing the expected cost of attempting to generate a path from h to M, beginning with op_{17};

$$c_{h,17}(1,1) = 5.0 \quad (53)$$

There is only one derivation for M, given an h, and it has cost 5.

To compute $c_{h,17}(1,2)$, requires $C_{\neg M,B}$, $C_{\neg M,A}$, and $C_{\neg M,O}$. The value of each of these is 1. The cost of failing to generate an M is 1 – the problem solver trys to generate a Y and fails. In addition, the cost of failing to construct a path to M must take into consideration the fact that M is ambiguous with A when A leads to the generation of an r and a q. Therefore,

$$c_{h,17}(1,2) = \frac{P(A \rightarrow h) - P(A \rightarrow \{qrh\})}{(P(h) - P(M \cap h))} * (1) + \frac{P(B \rightarrow h)}{(P(h) - P(M \cap h))} * (1) + \frac{P(O \rightarrow h)}{(P(h) - P(M \cap h))} * (1) = \quad (54)$$

$$\frac{0.2 - 0.05}{0.6 - 0.15} + \frac{0.2}{0.6 - 0.15} + \frac{0.1}{0.6 - 0.15} = 1. \quad (55)$$

Therefore, the expected cost of attempting to generate a path to final state M, given an h, is,

$$p_{h,17}(1) * 5 + (1 - p_{h,17}(1)) * 1 = 0.25 * 5 + 0.75 * 1 = 2. \quad (56)$$

The expected cost of attempting to generate a path from h to O, beginning with op_{21} is computed from;

$$c_{h,21}(1, 1) = 4.0 \quad (57)$$

There is only one derivation for O, given an h, and it has cost 4.

The cost of failing to construct a path to O given an h must take into consideration the fact that O is ambiguous with A. Furthermore, the cost of failing to generate an O given B or M is 1. (The problem solver fails to generate an X with cost 1.) Therefore,

$$c_{h,21}(1, 2) = \frac{P(B \rightarrow h)}{(P(h) - P(O \cap h))} * (1) + \frac{P(M \rightarrow h)}{(P(h) - P(O \cap h))} * (1) = \quad (58)$$

$$\frac{0.2}{0.6 - 0.3} + \frac{0.1}{0.6 - 0.3} = 1. \quad (59)$$

Therefore, the expected cost of attempting to generate a path to final state O, given an h, is,

$$p_{h,21}(1) * 4 + (1 - p_{h,21}(1)) * 1 = 0.5 * 4 + 0.5 * 1 = 2.5. \quad (60)$$

The *UPC* values for state f computed in a similar way, taking into account that f is ambiguous with A, and are shown in Fig. 61. Based on these values, the expected cost of attempting to generate a path to final state A, given an f, is

$$p_{f,3}(1) * 7 + (1 - p_{f,3}(1)) * 3 = 0.5 * 7 + 0.5 * 3 = 5. \quad (61)$$

The expected cost of attempting to generate a path to final state O, given an f, is

$$p_{f,21}(1) * 3.75 + (1 - p_{f,21}(1)) * 0 = 1.0 * 3.75 + 0 = 3.75. \quad (62)$$

9.5 Discussion

The examples in the preceding two subsections demonstrate how *UPC* values are calculated for states in an interpretation problem that does not include pruning operators. In addition, they show how costs increase as a result of problem structures associated with noise and missing data. (The effects from masking and distortion are identical.)

For example, compare the costs of *connecting* state h in the first case with similar costs in the second case. As defined in Section 4, an interpretation problem solver terminates only after all the search states have been connected. This means that all potential paths have been either explored or pruned. In the first example, h is connected only after the potential paths to A and B have been explored, and the cost of connecting h is 11. This is computed by summing the expected costs of each of the potential paths from h to final states. Specifically, two final states can be reached from h, A and B. The expected cost of the path to final state A is 4.5 (from equation 9.4.1) and the expected cost of the path to final state B is 6.5 (from equation 9.4.1).

In the second case, after the introduction of noise and missing data, the cost of connecting h is 14.15 (from equations 9.4.2 and 9.4.2). Similarly, the cost of connecting state f increases from 6 to 8.75.

In this simple example, the effects of noise and missing data were relatively modest. In fact, they were intentionally kept modest to improve readability. In real-world domains, the effects of uncertainty on the cost of connecting states are much more significant. In some domains, the increase in costs can be polynomial or exponential. An example of such a domain will be given in a subsequent section.

These two examples demonstrate a very important observation that will be addressed in the remainder of this paper. This observation is related to the *way* in which problem solving costs increase. Cost increases can be divided into three categories,

False Positives – One of the causes for increased problem is the result of ambiguity. Specifically, in the second example, summarized in Fig. 61, the sum of the probabilities of reaching one of four final states, A, B, M, or O, from h is greater than one! This means that for some inputs, more than one final state can be reached. By definition, in interpretation problems, all final states that can be generated *must* be generated and their utilities compared in order to determine which is the correct interpretation. Therefore, when there are multiple competing final interpretations, the work associated with generating the incorrect, or "false," interpretations is wasted.

False Negatives – Another cause for increased problem solving costs is the result of semi-ambiguity, i.e., partial paths that require a non-zero amount of work to eliminate. We refer to these phenomena as "false negatives" because they are incorrect search paths that do not lead to complete interpretations of the data. In the examples presented here, the effects of these false negatives can be seen in the cost vectors in Fig. 61. In the vectors shown, the costs of attempting to generate a path to a particular final state when the path cannot be generated have increased, especially for state f, over the costs shown in Fig. 58.

Redundancy – One of the more significant causes of increased problem solving costs is that associated with ambiguity that results in multiple search paths leading to the same final state. Though not demonstrated explicitly here, this phenomena is similar to the False Positive category, but it is also applicable to correct solutions. In a redundant domain, there may be multiple search paths to a correct interpretation. The effect on the expected cost of problem solving is similar to the effects of False Positives.

10 Experimental Verification

To verify the analysis framework presented in the preceding sections, we have constructed an IDP/*UPC* problem solving testbed. The basic structure of the testbed is shown in Fig. 63. A domain problem structure is specified in the form of a phrase-structured grammar with associated distribution, utility (credibility), and cost functions corresponding to each rule of the grammar. In addition, the problem solver's model of the problem domain is also specified as a phrase-structured grammar with associated functions.

The Domain Simulator uses the specification of the problem domain to generate problem instances. The problem solver uses its model of the problem domain's structure to interpret each problem instance. The problem solving actions available are specified as production rules of the grammar. In addition,

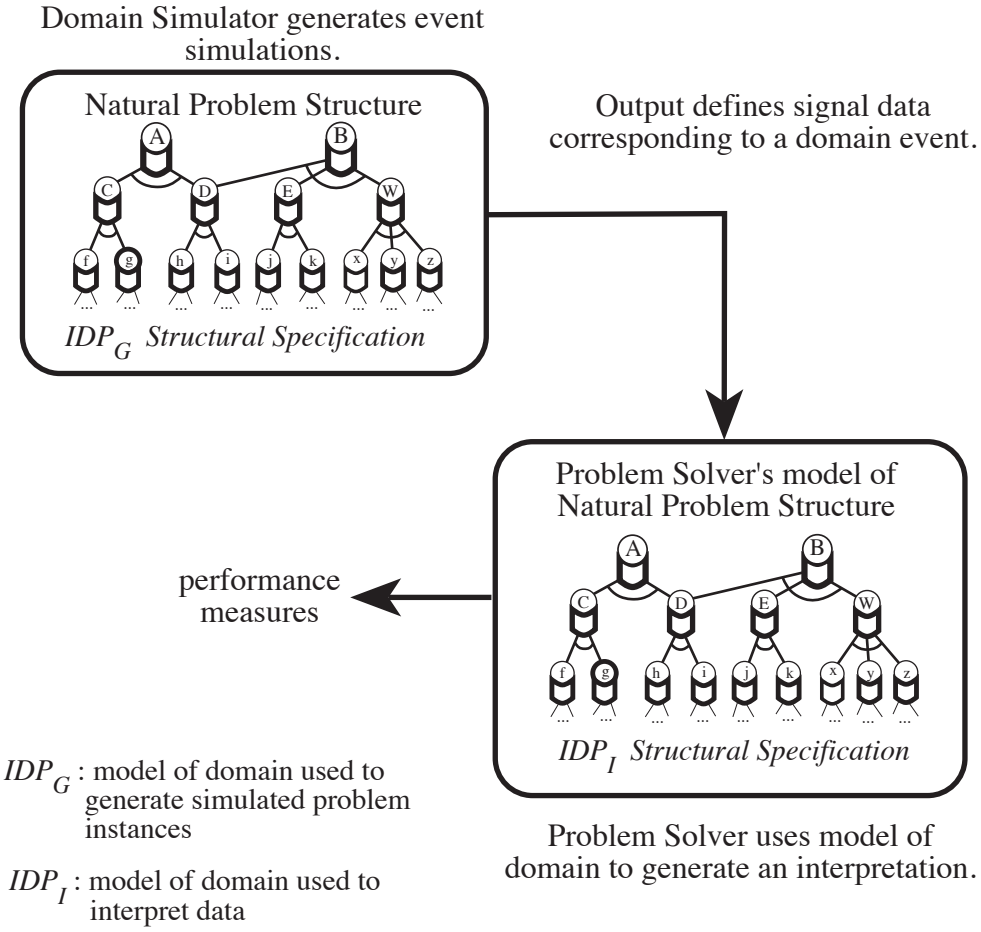


Figure 63: Overview of the IDP Model as the Foundation of an Experimental Testbed

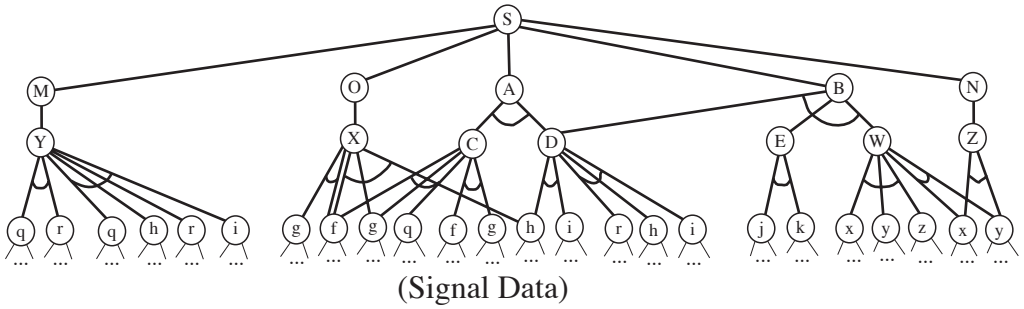


Figure 64: Interpretation Grammar G' with Added Noise and Missing Data Rules

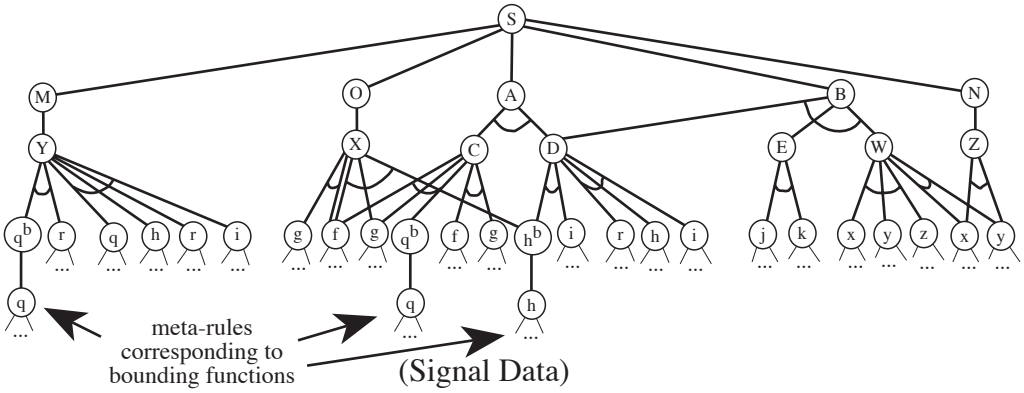


Figure 65: Example of Bounding Function Incorporated in a Grammar

each production rule of the grammar has a credibility function associated with it that is used to generate ratings of intermediate and final problem solving states.

The objective strategy that constitutes the basic control component of the problem solver is a simple *best-first* algorithm that attempts to generate an optimal interpretation based on equation 8.2. In our problem solving system, credibility is calculated dynamically,as defined in Section 9.3, and this calculation is used to determine *UPC* values. The conditional probabilities and expected cost components of the *UPC* vectors are computed a priori. The domain characteristics that change from run to run are represented with the feature list convention [15].

Using this testbed, we have conducted two sets of verification/validation experiments using the grammars shown in Figures 64 and 65. The first set of experiments were designed to verify the basic probability and cost estimation functions from preceding sections and used simplifying assumptions about the utility structure of the domain. The second group of experiments focused on the effects of using incorrect models. In these experiments, the problem solver's model of the actual problem domain's structure is distorted in a variety of ways. The intent was to investigate the manner in which a problem solver's performance is affected by a deviation from an ideal domain theory. The two sets of experiments are summarized in the next two subsections.

10.1 Experiment Set 1

The results of the first set of verification experiments are shown in Table 2. The column labelled "Grammar" indicates the domain and problem solver specification used in the experiment. $E(Cost)$

indicates the expected cost of problem solving in the domain based on an analysis of the grammar. "Avg. Cost" shows the actual average cost of problem solving, for 100 samples (each of 50 problem solving instances), in the domain. The "Sig" column indicates whether any difference in the Expected Cost and the Average Cost is statistically significant[22] based on a t test with a 95% confidence level. In each experiment, the hypotheses that are tested are $H_0 : \mu_{cost} = E(C)$ and $H_1 : \mu_{cost} \neq E(C)$ where μ_{cost} is the normalized population mean (each element of the population consisting of 50 problem solving instances), and $E(C)$ is the analytically predicted normalized population mean[23]. For experiments labelled significant, the null hypothesis, H_0, is rejected. The last column shows the number of correct answers that were found.

In these experiments, the grammar employed a credibility structure that simplified the hand computations used to verify the experimental results. In the simplified structure, productions corresponding to noise and missing data in the environment had the same credibility as "correct" interpretations. There was no credibility reduction associated with missing data or noise. As will be seen, this caused the results of credibility based pruning experiments to be somewhat disappointing, since the correlation between low-credibility partial results and correct interpretations was the same for both "correct" partial results and "incorrect" partial results. Experiments in the next subsection will investigate the effects of pruning in domains where the correlation between low-credibility partial results and "correct" interpretations is lower for "incorrect" partial interpretations. In these domains, a problem solver can benefit significantly by pruning low rated data, i.e., the expected cost of problem solving will decrease significantly but the expected percentage of correct answers does not decrease dramatically.

10.1.1 Experiments 1, 2 and 3

In Experiment 1, every problem solving task, including pruning tasks, was assigned a constant cost of 10. This value was chosen to simplify the verification process. In general, the experimental testbed allows tasks to have arbitrarily complex cost functions. Experiment 2 was similar to Experiment 1 except that bounding functions were added. The bounding functions also had a cost of 10 and eliminated from consideration paths with expected credibilities of less than 0.5.

In Experiment 2, the use of a bounding function reduced the cost of problem solving, but it also decreased the number of correct answers found by the system. This is because the bounding functions eliminated certain paths, which reduced the cost, but some correct paths as well as some incorrect paths were included in the set of eliminated paths.

In each of the first three experiments, the parameters of the environment grammar, specifically the distribution functions, were the same as those in the problem solver's grammar.

10.1.2 Experiments 2 and 3

In Experiments 2 and 3, the bounding functions used were simple "threshold cutoffs." If a state had a rating below the threshold, T, the state would not be generated. In these experiments, the credibility structure of the problem solver's grammar was identical to that of the environment grammar used to generate problem instances.

[22] Note that the calculation of significance does not include any consideration of the percentage of correct answers found.

[23] In order to use the t statistic, it is necessary to use variables with normal distributions. The expected cost of a *single* problem solving instance in the grammars we test is not distributed normally. However, by the *Central Limit Theorem*, we can approximate a normal distribution by defining each element of the population to be 50 problem solving instances. Thus, each experiment consisted of 100 samples and each sample included 50 problem solving instances.

In Experiment 3, the cost of the bounding functions was reduced to 1. This reduced the cost of problem solving more, but it had no effect on the number of correct answers that were found. This is because the same paths were pruned in Experiment 3 as in Experiment 2, the cost reduction was associated with the reduction of the cost of executing a bounding function.

As shown by the percentage of correct answers found in these experiments, the form of bounding function used in these experiments is probably not appropriate. As was discussed above, this is because the bounding functions are pruning both "correct" and "incorrect" interpretations indiscriminately.

10.1.3 Experiments 4 and 5

In Experiments 4 and 5, the problem solver's grammar was unchanged, but the distribution parameters in the environment grammar were altered. In Experiments 1, 2, and 3, the distribution values were always split evenly between all possible alternatives. In Experiments 4 and 5, the distributions were changed to make certain alternatives more likely. In these experiments, for each grammar rule, the more credible right-hand-side (rhs) was assigned a distribution value of 0.9 and the other possible rhss split the remaining 0.1 evenly.

10.1.4 Experiments 6 and 7

In Experiments 6 and 7, the distribution functions used by the environment grammar and problem solver's grammar were identical, but the credibility generation functions differed. In the previous experiments, the credibility generation functions were based on a random function that produced an "average normalized credibility" of 0.5. The bounding function used would prune a state "q" or "h" with a rating less than 0.3.

In Experiments 6 and 7, the same bounding functions were used, but the credibility generation functions were changed in the environment grammar. In Experiment 6, the credibility function was altered to generate "average normalized credibilities" of 0.25. In Experiment 7, they generated "average normalized credibilities" of 0.75.

In Experiment 6, this lead to lower average problem solving costs, compared with experiment 3, as many paths were pruned, and fewer correct answers, as many of the pruned paths were actually correct paths. This was consistent with expectations. By lowering the expected credibility of partial results and by retaining the same bounding threshold, many more paths are pruned. Again, since the problem solver cannot differentiate between "correct" and "incorrect" partial interpretations based on their credibilities, the pruned paths included large numbers of both "correct" and "incorrect" paths. This resulted in the shown reduction in correct answers found.

In Experiment 7, the opposite was true. Problem solving costs increased, compared with experiment 3, as fewer paths were pruned, but fewer correct paths were pruned and the percentage of correct answers increased.

10.2 Experiment Set 2

In this set of experiments, the simple credibility function used in the first set of experiments was replaced with more intuitively correct credibility functions. Using these credibility functions, productions from the grammar associated with noise and missing data generate lower credibilities than those generated by "correct" productions. This change had a significant effect on the experimental results, as shown in Table 3.

Exp	Generation				Interpretation				Sig	% C
	G	Dist	U	$E(C)$	G	Dist	U	Avg. C		
1	1	even	0.5	201	1	even	0.5	203	N	100
2	2	even	0.5	189	2	even	0.5	187	N	80
3	3	even	0.5	180	3	even	0.5	181	N	80
4	1	skew	0.5	368	1	even	0.5	369	N	100
5	3	skew	0.5	198	3	even	0.5	198	N	96
6	3	even	0.25	157	3	even	0.5	156	N	50
7	3	even	0.75	194	3	even	0.5	193	N	92

Abbreviations

Exp: Experiment
Generation: Description of IDP Domain Grammar used to generate problem instances.
Interpretation: Description of IDP Interpretation Grammar specifying the problem solver.
G: The problem solving grammar used;
1: G'
2: G' and bounding functions with cost 10,
3: G' and bounding functions with cost 1,
Dist: Distribution of Domain Events;
even: domain events evenly distributed
skew: distribution skewed to more credible events,
U; expected problem instance credibility;
0.5: problem instances have expected credibility 0.5
0.25: problem instances have expected credibility 0.25
0.75: problem instances have expected credibility 0.75
$E(C)$: Expected Cost of problem solving for given grammar
Avg. C: actual average cost for 100 samples of 50 random problem instances each
Sig: Whether or not the difference between expected cost and
the actual average cost was statistically significant
Y: yes, there is a statistically significant difference
N: no, there is not a statistically significant difference
% C: percentage of correct answers found

Table 2: Results of Verification Experiments – Set 1

All of these experiments were based on an expected credibility of 0.5 and a bounding function with an a priori threshold of 0.33. Thus, certain partial results with credibilities lower than 0.33 are pruned.

In contrast to the first set of experiments, in this set of experiments, the correlation between a partial interpretation with a low credibility and a correct solution is much lower for "incorrect" partial results than for "correct" partial results. This is reflected in the results in the general improvement in overall problem solving performance, i.e., the expected cost of problem solving decreases and the percentage of correct answers increases.

In Experiment Set 2, Experiments 8, 9, and 10 are the baselines that other experimental results are compared with. Specifically, Experiments 9 - 17 can be compared with Experiment 8 to observe the effects of altering the distribution functions used by the problem solver so that they are different from those used to generate problem instances and of altering credibility functions in a similar fashion. Experiments 9 and 10 show only the effects of altering the distribution functions. Experiments 11 and 12 are restricted to demonstrating the effects of altering the credibility functions used by a problem solver.

To best judge the effect of modifications to the distribution functions or the credibility functions, Experiments 13, 14, and 17 should be similarly compared with the baseline established in Experiment 9. These experiments all use the same grammar, but different credibility functions. The credibility functions used in Experiment 9 are correct and Experiments 13 and 14 show the effects of using incorrect credibility functions that either overestimate or underestimate the credibility of results generated with noise and missing data rules. In Experiment 17, the credibility functions used are simply "bad." They rate some partial results too high, and some too low. Experiments 15 and 16 should be compared with the baseline established in Experiment 10.

10.2.1 Experiment 8

Experiment 8 serves as the baseline for the second set of experiments. In this experiment, the problem solver's model of the domain structure is exactly the same as that used to generate problem instances. The bounding functions used still prune some correct paths, but not as many as in the first set of experiments.

10.2.2 Experiment 9

In this experiment, the distribution of noise and missing data in the domain was increased, but the model used by the problem solver was not changed. The increase in noise caused a small increase in the cost of problem solving when compared with Experiment 8. Also, more correct paths were mistakenly pruned. This was due to the fact that increasing the distribution of noise and missing data decreased the number of higher rated correct problem instances. Consequently, more correct paths had lower ratings and were mistakenly pruned.

10.2.3 Experiment 10

The distribution of noise and missing data in the domain was decreased, and again the model used by the problem solver was not changed. There was a slight increase in the cost of problem solving related to fewer paths being pruned, but there was also a decrease in cost associated with fewer and less expensive incorrect paths, especially those associated with noise. These two effects canceled each other out and the difference between the cost of problem solving in this experiment and the cost from

Exp	Generation				Interpretation				Sig	% C
	G	Dist	U	$E(C)$	G	Dist	U	Avg. C		
8	3	even	0.5	191	3	even	0.5	190	N	96
9	3	skew1	0.5	195	3	even	0.5	195	N	91
10	3	skew2	0.5	189	3	even	0.5	190	N	98
11	3	even	0.5	179	3	even	low	179	N	94
12	3	even	0.5	203	3	even	high	202	N	98
13	3	skew1	0.5	191	3	even	low	191	N	88
14	3	skew1	0.5	209	3	even	high	210	N	98
15	3	skew2	0.5	185	3	even	low	185	N	95
16	3	skew2	0.5	198	3	even	high	199	N	98
17	3	skew1	0.5	203	3	even	bad	202	N	92

Abbreviations

Exp: Experiment

G: The problem solving grammar used;
1: G'
2: G' and bounding functions with cost 10,
3: G' and bounding functions with cost 1,

Dist: Distribution of Domain Events;
even: domain events evenly distributed
skew1: distribution skewed to more noise and missing data
skew2: distribution skewed to less noise and missing data

U; expected problem instance credibility;
0.5: problem instances have expected credibility 0.5
low: problem solver rates "bad data" lower
high: problem solver rates "bad data" higher
bad: problem solver rates some "bad data" higher, and some "good data" lower

$E(C)$: Expected Cost of problem solving for given grammar

Avg. C: actual average cost for 100 samples of 50 random problem instances each

Sig: Whether or not the difference between expected cost and the actual average cost was statistically significant
Y: yes, there is a statistically significant difference
N: no, there is not a statistically significant difference

% Correct: percentage of correct answers found

Table 3: Results of Verification Experiments – Set 2

Experiment 8 was small. There was an increase in the percentage of correct answers found. This resulted from fewer paths being pruned. i.e., given that there was less noise and missing data, fewer "bad paths" were explored. This lead to a reduction in the cost of problem solving and an increase in the number of correct answers found.

10.2.4 Experiment 11

The problem solver's model of the distribution of domain events was the same as that used to generate problem instances, but the problem solver's model of credibility rated partial results generated from missing data lower than it should have. The effects of this were beneficial. In a sense, the problem solver rated "bad" data lower than it should have. The effect was that the problem solver pruned the bad data more often and reduced the overall cost of problem solving compared to Experiment 8. Unfortunately, the problem solver also rated some "good" data lower than it should have, and it pruned this as well, resulting in fewer correct answers found.

10.2.5 Experiment 12

Again, the problem solver's model of the distribution of domain events was the same as that used to generate problem instances, but the problem solver's model of credibility rated partial results generated from missing data higher than it should have. This resulted in the problem solver pruning fewer partial results, both "correct" and "incorrect." Consequently, the cost of problem solving is higher than in Experiment 8.

10.2.6 Experiment 13

In this experiment, the problem solver's model of the distribution of domain events and credibility are both incorrect. The domain actually generates more missing data and noise than the problem solver expects, and the problem solver rates intermediate results based on the missing data too low. The cost of problem solving increases slightly because there is more noise to process, but because some intermediate results are rated lower, more pruning occurs. The net effect on cost is insignificant when compared with Experiment 8. However, when compared with Experiment 9, which is a more appropriate baseline, the cost of problem solving here is lower. This is because both 9 and 13 generate more noise and missing data than 8, but in 13, the noise and missing data is pruned more often. The pruning, however, is often of "correct" partial results, and the number of correct answers decreases when compared to both Experiments 8 and 9.

10.2.7 Experiment 14

In this experiment, the domain actually generates more missing data and noise than the problem solver expects, and the problem solver rates intermediate results based on the missing data too high. The result is an increase in cost compared with both Experiments 8 and 9. There is a greater amount of expensive noise to process, and the problem solver prunes fewer intermediate results. However, fewer correct paths are pruned resulting in a higher success rate.

10.2.8 Experiment 15

The domain generates less noise and missing data than expected, and the problem solver rates partial results based on missing data lower than it should. This results in a decrease in the cost of problem solving when compared with either Experiment 8 or the more appropriate baseline, Experiment 10. More correct results are pruned resulting in a lower number of correct answers being found. The difference is greater when compared with 10 than with 8.

10.2.9 Experiment 16

The domain generates less noise and missing data than expected, and the problem solver rates partial results based on missing data higher than it should. This results in a slight decrease in the cost of problem solving resulting from less noise to process, but fewer paths are pruned and the net effect is an increase in the cost of problem solving. The increase is significant when compared with either experiment 8 or 10. However, since fewer paths are pruned, the number of correct answers increases.

10.2.10 Experiment 17

In this last experiment, the domain generates more noise and missing data than expected, and it rates partial results based on noise higher than it should. In addition, the problem solver rates correct partial interpretations lower than it should. As a consequence, the cost of problem solving increases due to the increase in the amount of noise that must be processed and fewer correct answers are found, as more correct paths are pruned.

10.3 Discussion

Of the preceding experiments, 17 is probably the most realistic. The results from experiments 8 through 16 are generally consistent with the results from experiments 1 through 7. However, experiments 12 through 16 are more representative of real world domains where a problem solver's model of a domain's structure is slightly off both in terms of modeling the distribution of noise and missing data and in terms of modeling the distribution of credibility.

These domains are all unrealistic in the sense that *all* rules of a particular type were treated the same. For instance, if one missing data rule was set to generate credibility that was too high, then they all were set to generate credibility that was too high. In real domains, it is probably the case that some rules overestimate the correct credibility ratings and some underestimate. The same can be said for a problem solver's model of distribution of missing data and noise.

Experiment 17 is more realistic in the sense that we expect the inaccuracies in a problem solver's model of a domain to both increase the cost of problem solving and to decrease the number of correct answers produced.

These experiments are somewhat limited by the grammar used. The grammar used in experiments 1 through 17 was chosen to preserve consistency with previous sections of the paper. It should be remembered, however, that this is a simple grammar that does not represent the full effects of noise and missing data, not to mention distortion effects. These will be studied at greater length in subsequent papers.

More generally, these experiments statistically verify that the closed form equations used to determine expected costs analytically are correct. The experiments also show that when the domain model

is manipulated in various ways, the IDP/*UPC* analysis results conform to intuitive expectations.

11 Extending the *UPC* Formalism

This section presents an extension to the *UPC* formalism that will model the abstract and approximate reasoning strategies in interpretation problems. These modeling techniques will be particularly useful for formulating meta-operators used in IDPs (defined in Section 5.4) as part of a search problem. This will include meta-operators used implicitly by the control component. The emphasis of this section will be on defining the extensions. Subsequent sections will discuss the implications these extensions have for problem solving strategies.

At this point it is important to stress that the IDP formalism and the *UPC* formalism are independent methods for formalizing the character of a problem domain. In the IDP/*UPC* framework, the two formalisms are linked because the IDP formalism is used to generate *UPC* values. In general, this does not have to be the case. The IDP formalism can be used to analyze problem domains independent of the use of the *UPC* formalism and vice-versa. For other domains, it may not be as easy to represent base-level and meta-level operators in a unified representation as has been done with interpretation domains in this paper.

In extended *UPC* models, the problem solver has the option to *project* the base search space to a new, abstract search space where it can efficiently solve a simplified version of the problem. However, it is possible that the solution found in the projected space may not be of an acceptable form, so it is *mapped* back to the base search space. In general, this method will improve the overall efficiency of problem solving if, as a result of mapping the solution (or partial result) from a projected space back to the base space, problem solving in the base space is constrained in some way. For example, a partial result from a projected space can provide a more global perspective that can be used by problem solving activities in the base search space.

The extension to the *UPC* formalism is represented in Fig. 66. In the implied paradigm, the problem solver has the option to project the base space and conduct processing in the abstract space, mapping results back to the base search space.

Section 11.1 presents background material regarding the development and use of projection spaces. Section 11.2 presents a formal definition of projection space extensions to *UPC* models.

11.1 Background

The effort to formally incorporate notions of projected or abstracted search spaces into the traditional model is based on *Approximate Processing* concepts described in [12, 28, 29, 9] and on *goal processing* concepts described in [27]. Approximate processing is based on exploiting the structure of a search space to form abstractions of the space with well understood effects. A problem solver capable of exploiting approximate processing has access to "simplified" operators that it can use to search the abstracted version of a given search space. The results of searching the abstract space can be mapped back to the original space and used to enhance problem solving in that space. Experiments with approximate processing [29, 9] showed that significant efficiencies can be gained with careful exploitation of approximate processing mechanisms.

This paradigm can be viewed as a form of hierarchical problem solving, such as that discussed by Newell [31], Minsky [30], and Knoblock [24].

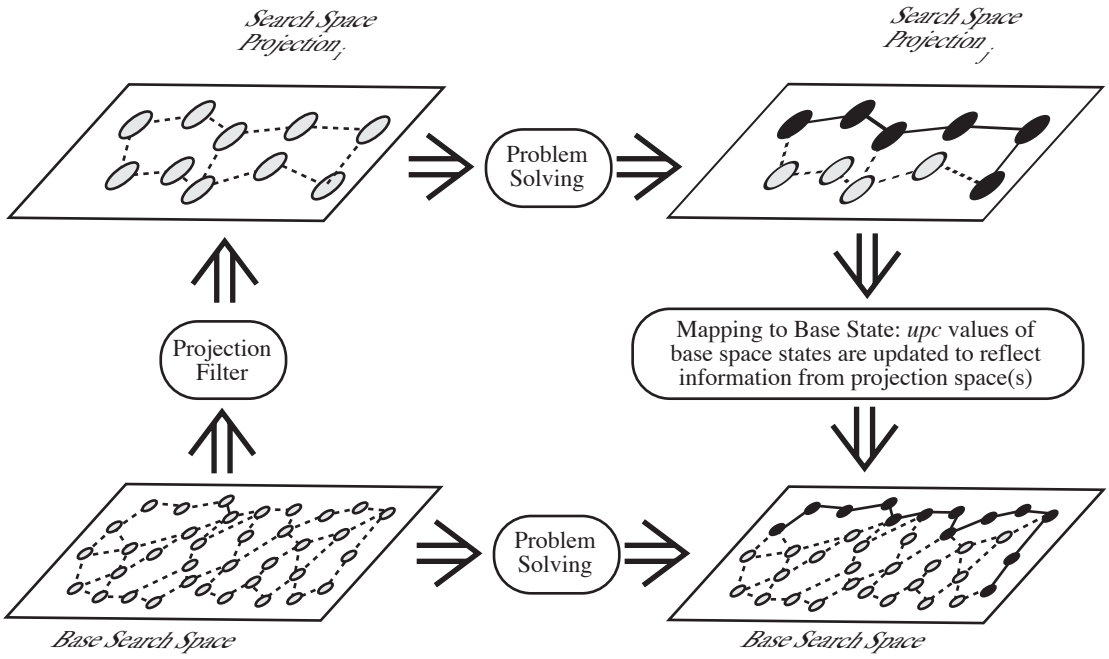


Figure 66: The Search Paradigm Implied by the Extended *UPC* Formalism

11.2 Formalizing Projection Spaces

The extended *UPC* formalism is intended to explicitly represent characteristics of search spaces that are used implicitly in control architectures. Specifically, the new formalism explicitly represents problem structures defined by subproblem relationships in a way that enables a problem solver to exploit them. (For IDPs, the structures that will be exploited by abstract processing are defined in Section 5.4.) The new characterization of a search problem is based on the four-tuple $< \mathcal{S}, \Omega, \omega, \Phi >$, where;

$\mathcal{S}$ = the *start state*; $\mathcal{S}$ is defined by the input data to the problem solver and the initial values of any relevant CVs.

Ω = the *base search space* with associated CVs and operators. Ω corresponds to the traditional notion of a search space. It is defined by CVs that specify the characteristics of individual states (including the *UPC* vectors), operators that map one state to another, and functions of CVs that define final states. In terms of an IDP, Ω is defined by the interpretation grammar.

ω = a set of *projections*, or *abstractions*, of the base search space, each with their associated CVs and operators. Final states are those from the base space that can be reached via mapping operators. A given search space projection ω_i is defined by two sets of operators, $OP_{(\Omega,\omega_i)}$ and $OP_{(\omega_i,\omega_i)}$. $OP_{(\Omega,\omega_i)}$ is the set of operators that map states from Ω to states in ω_i. $OP_{(\omega_i,\omega_i)}$ is the set of operators that map states in ω_i to other states in ω_i. As with Ω, each state in a projected search space is characterized by a set of CVs and a set of *UPC* vectors.

An appropriate metaphor for a projection of a search space is that it is transformed by "projecting" it through a filter defined by the set of operators $OP_{(\Omega,\omega_i)}$. The result is a blurring of the original space into a simpler, less clearly defined search space. With a well designed filter, states with similar

properties will be merged into abstract states and relationships among states will cause sought after states to become more apparent and states with undesirable properties to be eliminated entirely.

Ideally, the results of projection will be a space that is several orders of magnitude less costly to search and that can be mapped back to the original space in a way that reduces problem solving costs. The strategy then is to find a solution (or partial solution) in the projected search space and to somehow use this solution as a guide to problem solving in Ω. If the cost of finding a solution in a projected search space is less than the cost saved, then the strategy is beneficial. Subsequent sections will discuss the general properties of search spaces where abstract processing can be beneficial.

Φ = a set of *mapping functions* from projection spaces back to the base search space. The objective of problem solving in an abstract projection space is the generation of constraints that can somehow be used to restrict problem solving in the base search space. Functions in Φ can be thought of as the mechanisms that map constraints from an abstract space, ω_i, back to the base search space, Ω. This can be done by creating new states in Ω, or by modifying existing states. A taxonomy of mapping strategies is defined in Appendix B.

Given these definitions of $< \mathcal{S}, \Omega, \omega, \Phi >$, the *UPC* formalism can now be represented as shown in Fig. 67. Figure 67.a summarizes the representation of a state. A state in the search space is defined by a set of CVs, where each CV_i is a characteristic variable with a corresponding attribute value. In addition, each state has a set of *UPC* vectors corresponding to each operator available to extend the state. It is important to note that, even though *UPC* vectors only include information corresponding to paths to final states in the base space with non-zero utility, the *UPC* vectors of a state will include information corresponding to *all* paths, even paths that traverse one or more projection spaces, extending from the state. The *UPC* representation will unify the base search space and all relevant projection spaces.

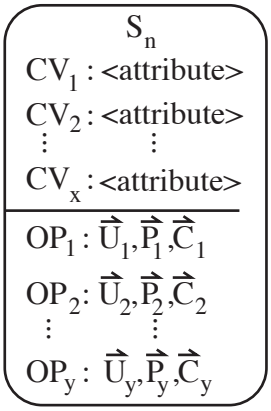
Figure 67.b is a representation of how the *state of the problem solver,* S_{PS}, is defined in the *UPC* formalism. In essence, S_{PS} solver is defined by the status of problem solving in the base search space, Ω, and all the projection spaces, ω_i.

11.3 Projection Space Example

Figure 68 shows a version of grammar G' with noise and missing data and Fig. 69 shows meta-operator additions to G'. In the problem instance defined by these grammar rules, $\mathcal{S}$ is the set of raw input data from sensors and Ω is defined by the rules of G' shown in Fig. 68. There is a single projection space, ω_1, and it is defined by $OP_{(\Omega,\omega_1)} = \{op_{32}, op_{33}, op_{34}, op_{35}\}$ and $OP_{(\omega_1,\omega_1)} = \{op_{30}, op_{31}\}$. The operators corresponding to rules 32, 33, 34, and 35 of G' project the base space to an abstract space and rules 30 and 31 map states in the projection space to other states in the projection space. The set $\Phi = \{op_{28}, op_{29}\}$. The operators corresponding to rules 28 and 29 map the results of problem solving in the projection space back to the base space.

The *UPC* values for abstract states are calculated in the same manner as the calculation of *UPC* values for base space states. Figure 69 shows meta-operator additions to the grammar G' that will be used in an example to illustrate this. *UPC* values for abstract state D^h are shown in Fig. 70.

a.

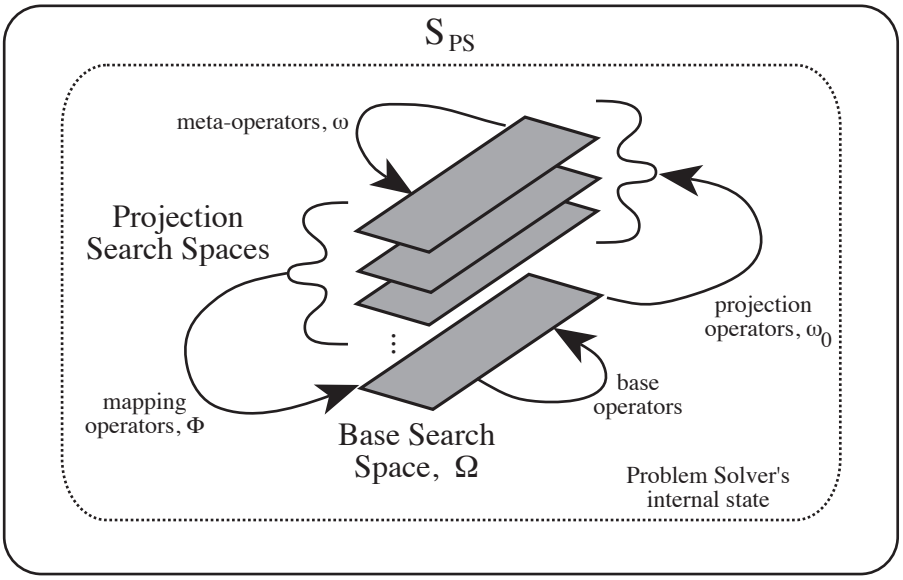


Extended State representation: operators applicable to each state are represented with the corresponding *Utility, Probability,* and *Cost* vectors.

Each vector entry consists of a measure of the *expected value* and a measure of the *variance*.

Expected values and variances are determined from IDP's cost and credibility functions. (In interpretation problems, credibility = utility.)

b.



UPC Summary

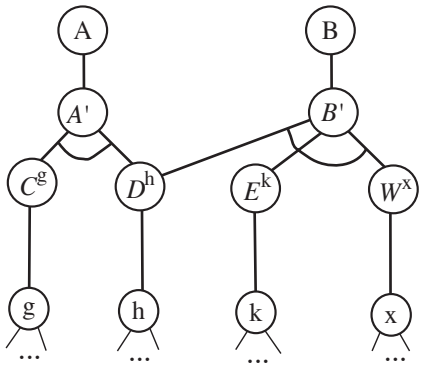
Figure 67: Overview of Extensions to the *UPC* Formalism

1. A → CD	7. f → (signal data)	16. M → Y
2. B → DEW	8. j → (signal data)	17.0 Y → qr
3.0 C → fg	9. g → (signal data)	17.1 Y → qhri
3.1. C → fgq	10. k → (signal data)	18. N → Z
4. E → jk	11. h → (signal data)	19. Z → xy
5.0 D → hi	12. x → (signal data)	20. O → X
5.1. D → rhi	13. i → (signal data)	21.1. X → fgh
6.0 W → xyz	14. y → (signal data)	21.2. X → fg
6.1. W → xy	15. z → (signal data)	

Interpretation Grammar G' with noise and missing data rules

Figure 68: G' Noise and Missing Data Rules

28. A→ A'
29. B→ B'
30. $A' \rightarrow C^{g}D^{h}$
31. $B' \rightarrow D^{h}E^{k}W^{x}$
32. $C^{g} \rightarrow$ g
33. $D^{h} \rightarrow$ h
34. $E^{k} \rightarrow$ k
35. $W^{x} \rightarrow$ x



(Signal Data)

Figure 69: Meta-Operators for Grammar G'

$u_{D^h,30} = (1_A)$	$p_{D^h,30} = (0.5_A)$	$c_{D^h,30} = ((9,1)_A)$
$u_{D^h,31} = (1_B)$	$p_{D^h,31} = (0.33_B)$	$c_{D^h,31} = ((13,1)_B)$

Figure 70: *UPC* Vectors for Abstract State D

The conditional probability of A given D^h is, from Definition 9.13,

$$P(A \mid D^h) = \frac{P(A \cap D^h)}{P(D^h)} = \tag{63}$$

$$\frac{P(A \rightarrow D^h) + P(O \rightarrow D^h)}{0.6} = \tag{64}$$

$$\frac{0.3}{0.6} = 0.5. \tag{65}$$

A is ambiguous with O, so

$$P(A \cap D^h) = P(A \rightarrow D^h + P(O \rightarrow D^h)). \tag{66}$$

$$P(A \rightarrow D^h) = P(A) * 1 = 0.2. \tag{67}$$

$$P(O \rightarrow D^h) = P(O) * 0.5 = 0.1. \tag{68}$$

Furthermore, $P(D^h) = P(h) = 0.6$. This is from the observation that the only element on the RHS of D^h is h, so

$$P(D^h \mid h) = 1 \Rightarrow P(D^h) = P(h). \tag{69}$$

The expected cost of a path to A' from D^h is 3. This assumes that each of the operators shown in Fig. 69 has cost 1. The expected cost of a path from A' to A is the expected cost of a generating an A minus the expected costs of g and h, or 6. Thus,

$$c_{D^h,30}(1,1) = 9. \tag{70}$$

The expected cost of a failed attempt to generate an A is always 1. When the problem solver attempts to extend D^h, it first tries to generate a path to A' including a g. The cost of failing to generate a g is 1, and, when g fails, the problem solver immediately ceases its attempt to generate a path from D^h to A.

The expected cost of attempting to generate a path from D^h to A is, from Definition 9.16,

$$p_{D^h,30} * 9 + (1 - p_{D^h,30}) * 1 = 0.5 * 9 + 0.5 = 5. \tag{71}$$

The *UPC* values for a path from D^h to B are calculated in a similar manner and are shown in Fig. 70. Given these values, the expected cost of attempting to generate a path from D^h to B is

$$p_{D^h,31} * 13 + (1 - p_{D^h,31}) * 1 = 0.33 * 13 + 0.67 * 1 = 4.96. \tag{72}$$

It is important to note that the addition of meta-operators also changes the *UPC* vectors for state h. (There are no meta-operators defined that are applicable to state f, so its *UPC* values do not change.) These changes, shown in Fig. 71, reflect the paths from h through the abstract states A' and B' to final states A and B. However, these additions to h's *UPC* vectors do not increase the cost of connecting h. This is because the cost of connecting a base space state is defined in terms of the costs of generating base space paths. Therefore, the addition of potential paths in projection spaces are not included in the calculation for the cost of connecting a state.

It is also important to note that the addition of potential paths in projection spaces can *reduce* the cost of connecting a state. This is a very important point and it will be discussed in more detail in the next section.

$u_{h,5} = (1_A, 1_B)$	$p_{h,5} = (0.33_A, 0.33_B)$	$c_{h,5} = ((7,3)_A, (10,3)_B)$
$u_{h,17} = (1_M)$	$p_{h,17} = (0.25_M)$	$c_{h,17} = ((5,1)_M)$
$u_{h,21} = (1_O)$	$p_{h,21} = (0.5_O)$	$c_{h,21} = ((4,1)_O)$
$u_{h,33} = (1_A, 1_B)$	$p_{h,33} = (0.5_A, 0.33_B)$	$c_{h,33} = ((10,2)_A(14,2)_B)$

Figure 71: *UPC* Vectors for State h Given Meta-Operator Extensions

12 Potential - The Basis for Control

As described in Section 8, an implicit (or in some cases explicit) objective of every interpretation problem solver is optimal processing. However, based on the definition of interpretation problem solving from Section 4, it might seem as if there is little an interpretation problem solver can do to enhance its efficiency. The reason for this is that interpretation problem solvers must, by definition, explore *every possible solution path* (i.e., connect the base space) in order to determine a solution. Recall that in Section 4 interpretation problems were defined as discrete optimization problems where a problem solver has to identify the "best" element of a set of interpretations, S. It is not sufficient for the problem solver to determine a single interpretation; the problem solver must consider *all* possible solutions and determine which is the "best." Given this requirement, the costs of problem solving appear to be dependent solely on characteristics of the domain such as its size, the cost of operator applications, etc., and are beyond the influence of the problem solver.

Fortunately, for many domains, control architectures can be formulated that do not require exhaustive enumeration of every search path in the base space and that enable an interpretation problem solver to connect every state in Ω very efficiently. These architectures use implicit enumeration actions that prune paths based on the structure of the search space without fully extending the paths. The IDP/*UPC* framework supports the analysis of a class of implicit enumeration strategies for interpretation problems that we defined as sophisticated control architectures. In Section 5, we demonstrated this analysis for a simple pruning algorithm.

In Section 8, Equations 8.4 and 8.1 defined the basis for making optimal control decisions from a local perspective. These equations specify that, from a local perspective, optimal processing is achieved when the operator chosen for execution maximizes the ratio $c(op_i)/cost(op_i)$ where $c(op_i)$ is the degree to which op_i reduces the expected cost of connecting all open states, and $cost(op_i)$ is the cost of executing op_i.

As discussed in Section 8.2, in many cases, the locally optimal control decision will not result in globally optimal problem solving. This occurs in situations where an operator on a search path does more than simply expand a state and extend a search path – where the operator actually *increases the information available to a problem solver regarding the interrelationships between partial solutions.* i.e., the operator increases the understanding of a partial solution's global significance. In these situations, the operator does not have to actually extend a search path in order to move the problem solver closer to termination, rather, the operator may alter the search space in some way that reduces the cost of problem solving (or increases the effectiveness of problem solving efforts) for some other set of operators. We will refer to this property of an operator as its *potential.*

The example used to demonstrate the problems associated with locally optimal control decisions is reproduced in Fig. 72. In this example, op_y alters the search space in some way to make it less costly

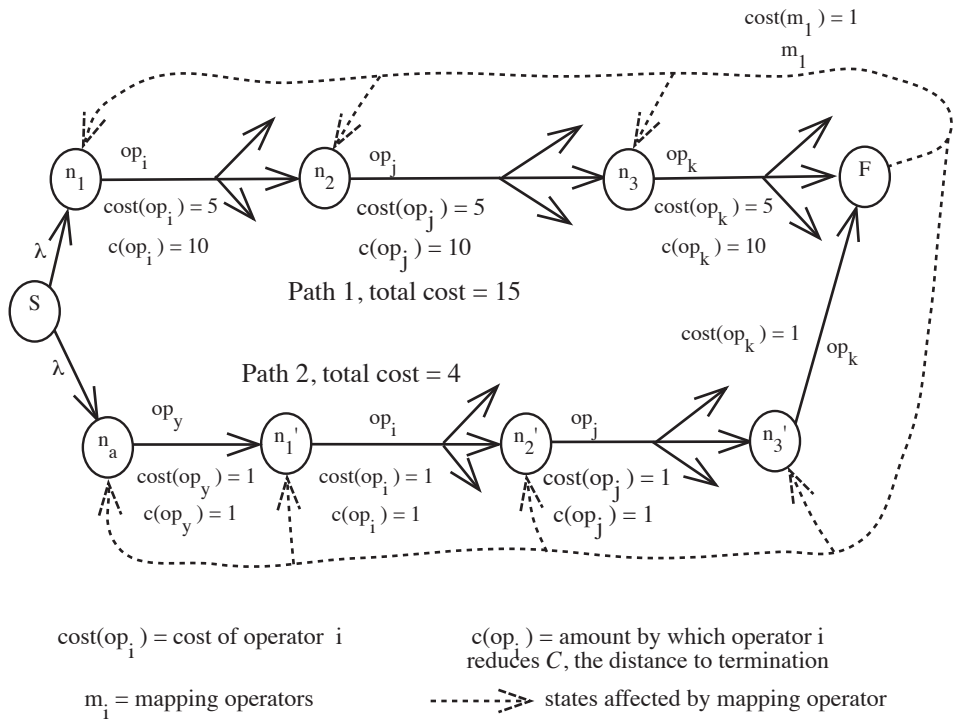


Figure 72: Example of the Non-local Effects of an Operator Application

to execute a series of operators that generate final state F. In addition, the dotted lines extending from F in Fig. 72 indicate operators that map the results of generating F back to other search states and reduce the subsequent cost of problem solving. For example, the mapping operations may eliminate redundant activities. This could be implemented by mapping operators that are applied to state F that eliminate from consideration operators that generate paths solely to final states that are identical to F. Given these mapping operators, it now becomes feasible to overcome problems associated with locally optimal control decisions by taking into consideration the long-term implications of an operator. In this specific instance, op_y is superior to op_i because the long-term effects of its execution will be to generate final state F at a lower-cost and this will lead to the execution of a mapping operator that will prune redundant paths, including the path that begins with op_i. In such a situation, a problem solver must have some way of quantifying the effects of applying op_y that reflects the long-term benefits, i.e., the potential, of the operator.

The concept of potential is a critical element of the IDP/*UPC* analysis framework. It takes into account the changes in the *UPC* representation of base space states that occur as a result of the added information provided by an operator or a sequence of operator applications. Thus, potential is a mechanism that allows us to understand the interrelationships that exist between the current set of states (i.e., the search paths that have been created so far) and the states that can be derived from them. This includes an understanding of the long-term effects of an action. Potential relates to operations at all levels of processing, i.e., base space and projection space, but it is of particular importance to meta-operators that use abstractions or approximations. This is because potential can be used to address the question of how to evaluate the contribution made by meta-operators in terms that are consistent with the evaluation of problem solving actions that directly connect states in the base space. In general,

meta-operators are not associated with effects that can be quantified in the same way as the effects of base space operators. The effects of meta-operators are related more to long-term reductions in problem solving cost or increases in solution quality. Although a meta-operator may have no immediate effect on any base space search paths, which might appear to make it an undesirable choice of action, it may have a very significant long-term effect that reduces the expected cost of problem solving dramatically, making it a very good choice of action. In contrast, base space operators can be thought of as explicit enumeration mechanisms and they are associated with immediate effects resulting in the extension of base space search paths that can be easily quantified.

Formally,

Definition 12.1 *Potential of operator op_i applied to state s_n, $Pot(op_i, s_n)$ = $FTN_{\forall S'}$ ($P(S' \mid s_n)$, $Potential(S')$, $cost_g(S', s_n)$, $cost_m(S')$),* where FTN is a function that is described below, each element S' is a state that can be reached from s_n that increases the information available to a problem solver regarding the interrelationships between partial solutions (i.e., a state with potential), $P(S' \mid s_n)$ is the probability of generating S' given s_n, $Potential(S')$ is a measure of the degree to which S' reduces the cost of problem solving, $cost_g(S', s_n)$ is the expected cost of generating S' given s_n, and $cost_m(S')$ is the cost of realizing $Potential(S')$, i.e., the cost of mapping S' back to the base space.

It is important to point out that this definition of potential is based solely on the characteristics of s_n and the statistical properties of the domain calculated from the IDP definition. It does not take into account the existence, or absence, of any other states.

The general computation of $Pot(op_i, s_n)$ is $(P(S' \mid s_n) * (Potential(S') - cost_g(S', s_n) - cost_m(S')))$. This is for situations where the cardinality of S' is 1. For situations where an operator, op_i, represents the first step on paths to multiple S'_i, the computation is more complicated. In these situations, the function FTN determines $Pot(op_i, s_n)$ based on the relationships between the states S'_i. These relationships are defined in terms of the paths from s_n to the states S'_i (i.e., the costs $cost_g(S', s_n)$) and the set of states that are affected when the potential of the states S'_i is mapped back to the base space. Figure 73 depicts the possible relationships between states S'_i. Figure 73 represents the possible relationships in terms of search space paths and base space states.

Intuitively, it is easy to think of the benefits of potential as being cumulative. If this were true, the the distance to termination should be reduced by the sum of the potential, $Potential(S'_i)$, of all states S'_i. However, Fig. 73 shows a situation where this is not correct. Specifically, when the sets of states affected by the mapping functions that propagate the potential of the states, S'_i, back to the base space interact, some of the benefits of the potential might overlap or be redundant. In this case, summing the potential of the states, S'_i, would give an overestimate of the expected benefits.

Similarly, it is easy to develop an intuitive perspective of the costs of generating the states, S'_i, as a sum. However, Fig. 73 shows situations where this is not correct. When the paths to the states, S'_i, interact or overlap, summing results in an overestimate of the costs and an underestimate of the benefits of the potential associated with the states, S'_i.

The general formulas for calculating the expected costs of reaching states with potential, S_i, and the effects of mapping the potential back to the base space, are:

Definition 12.2 *Expected cost of generating states, S_i = $cost_g(S'_1, s_n)$ + $cost_g(S'_2, s_n)$ + ... + $cost_g(S'_m, s_n)$ - $cost_\cap(S'_1, S'_2, \ldots S'_m, s_n)$,* where m is the number of states, S_i, and $cost_\cap(S'_i, S'_j, \ldots, s_n)$ is the intersection of the paths from s_n to the states S'_i, S'_j, ... In other words, the expected costs are summed, then all possible interactions are computed and subtracted from this cost. In situations where the

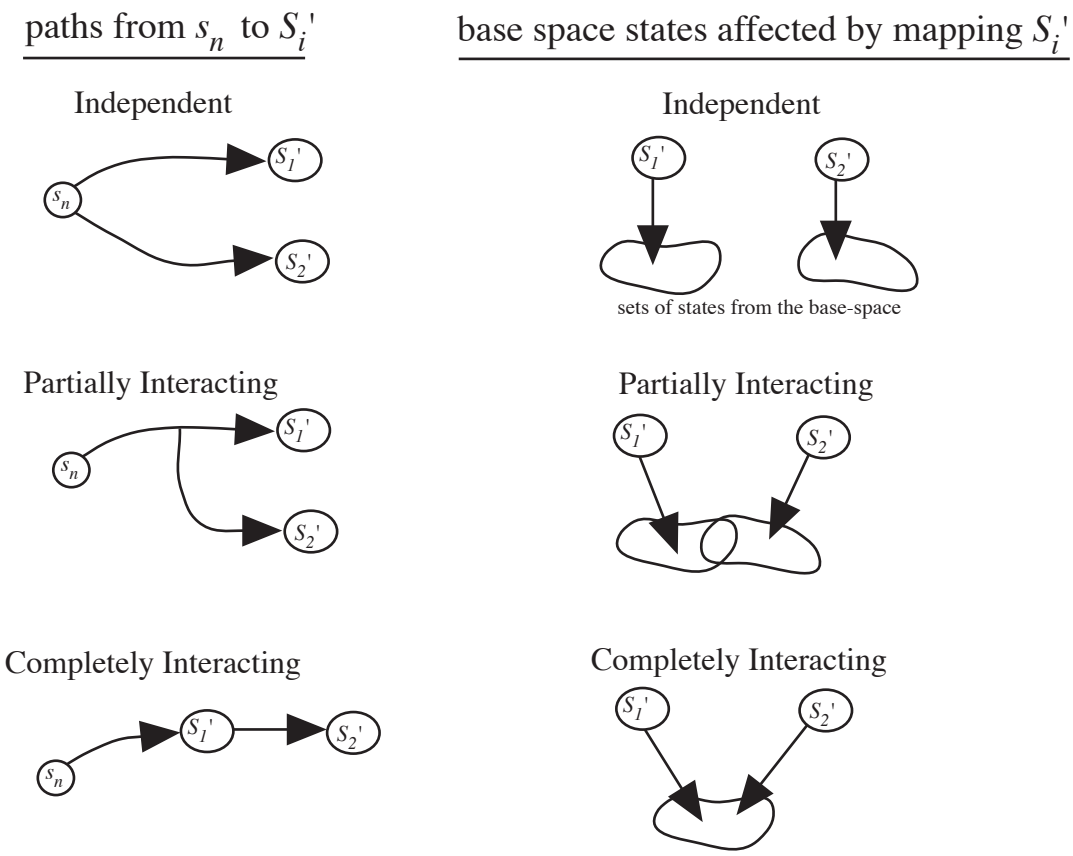


Figure 73: Relationships Between States with Potential

states, S_i are independent, the intersection terms are all 0 and this simplifies to the sum of the costs. In situations where the paths to the states are completely interacting, this simplifies to the maximum cost path.

Definition 12.3 *Expected benefits of mapping potential from states S_i back to base space* = $Potential(S'_1)$ + $Potential(S'_2)$ + ... + $Potential(S'_m)$ - $potential_{\cap}(S'_1, S'_2, \ldots S'_m)$, where m is the number of states, S_i, and $potential_{\cap}(S'_i, S'_j, \ldots)$ is the intersection of the expected potential derived from mapping the states $S'_i, S'_j, \ldots$ back to the base space. In other words, the expected benefits of the potential of the states, S'_i are summed, then all possible interactions are computed and subtracted from this cost. In situations where the states, S_i are independent, this is the sum of the potentials. In situations where the sets of base space states are completely interacting, this is the maximum of the $Potential(S'_i)$.

In general, the costs $cost_m(S'_i)$ of mapping the potential of states back to the base space must be treated in a manner similar to that discussed above.

In the experiments in the following sections, the paths to states with potential are all independent, so their costs are summed, and the sets of states affected by mapping functions are all completely interacting, so the maximum potential is used.

The following definition will be used in discussions of potential.

Definition 12.4 *Distance to Termination,* C - The expected amount of processing an interpretation problem solver must perform before an answer is determined. More formally, *C* is the sum of the expected costs of connecting all open states in Ω.

For operator op_i, $Pot(op_i, s_n)$ represents the degree to which it reduces C by altering the nature of a search space S_{PS} exclusive of extending any search paths. We will give an example of calculating $Pot(op_i, s_n)$ in Section 12.3. $cost_m(S')$ represents the cost of mapping operators that would be used to realize the potential of an abstract state in a projection space. This cost is also associated with actions in the base space, such as bounding operators, that are also used to realize $Pot(op_i, s_n)$.

Given the definitions of $Pot(op_i, s_n)$ and C, all potential operators can be judged based on their expected impact on the distance to termination. Each potential operator will extend one or more paths[24] by an expected amount and the "value" of an operator can be determined by the degree to which this amount reduces the distance to termination. Figure 74 illustrates the computation of distance to termination.

A problem solver's distance to termination can be determined by summing, for each intermediate state not fully expanded, the expected cost of extending paths to all potential final states that can be reached from the intermediate state. Figure 74 illustrates the calculation of distance to termination. In addition, the computation of distance to termination must factor in some notion of the *potential* of available courses of action.

The potential associated with a path is based on the subsequent availability of a mapping operator capable of recognizing and exploiting the emerging structure of the problem instance. For example, by modifying the *UPC* values of the base space. (An example will be presented in the next subsection and a taxonomy of mapping operators is defined in Appendix B.) Intuitively, the process of recognizing an

[24] In more complex grammars, for example, those that include *noise* (which is defined in Section 5.1.2), it is possible for a single operator execution to extend multiple paths from a state.

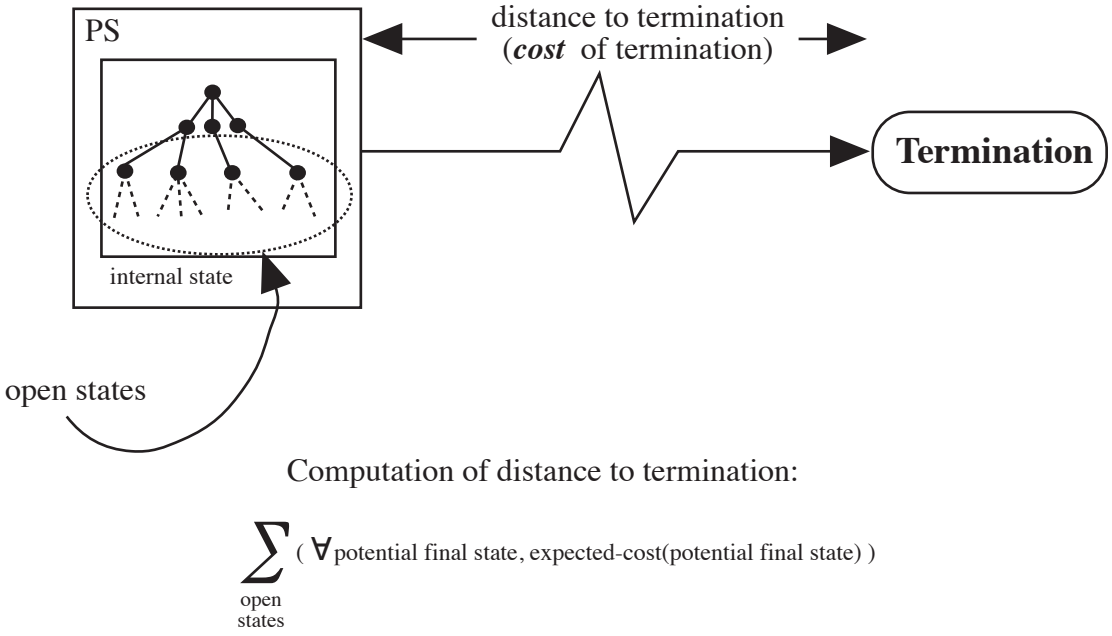


Figure 74: Representation of a Problem Solver's Distance to Termination

emerging problem structure is analogous to viewing the search space from a high vantage point where it is possible to discern general features of the problem space that can be used to guide and improve the efficiency of problem solving efforts. In effect, this extends the representation of a search space to a three dimensional space where the topology of a search space is defined by potential.

The relationship between potential and the distance to termination is illustrated in Fig. 75. In this figure, C represents the expected cost of problem solving, i.e., the distance to termination. C' represents the cost of problem solving in a new search space, one created by the application of operator op_i. If op_i does not alter the properties of the search space S_{PS} in any appreciable way, then $C = C' + cost(op_i)$. In other words, the only difference between the two search spaces S_{PS} and S'_{PS} (which is the result of applying op_i to S_{PS} is that paths associated with op_i have been extended. If op_i does alter the properties of the search space S_{PS} in a way such that $C > C' + cost(op_i)$, then the difference is what we refer to as potential.

As an example, consider a domain with bounding functions that are not based on a predetermined threshold, but that are based on the results of problem solving. If, for example, the domain uses the credibility of any full interpretations it derives to prune paths during subsequent search, the creation of a full interpretation alters the nature of the search space in a significant way. In essence, the creation of a full interpretation that is used to prune other potential search operations transforms the existing search space, S_{PS}, into a new search space, S'_{PS} that is (hopefully!) less costly to search.

12.1 Calculating Potential

As discussed in the previous subsection, the potential of a state, $Potential(S')$, is based on the degree to which the existence of S' reduces the expected cost of problem solving. This can be formalized as:

Definition 12.5 *Potential of a state,* $S = Potential(S') = C_{s_n,t} - C_{S',t',s_n,t}$, where $C_{s_n,t}$ represents the expected cost of problem solving given the existence of state s_n at time t, $C_{S',t',s_n,t}$ represents the

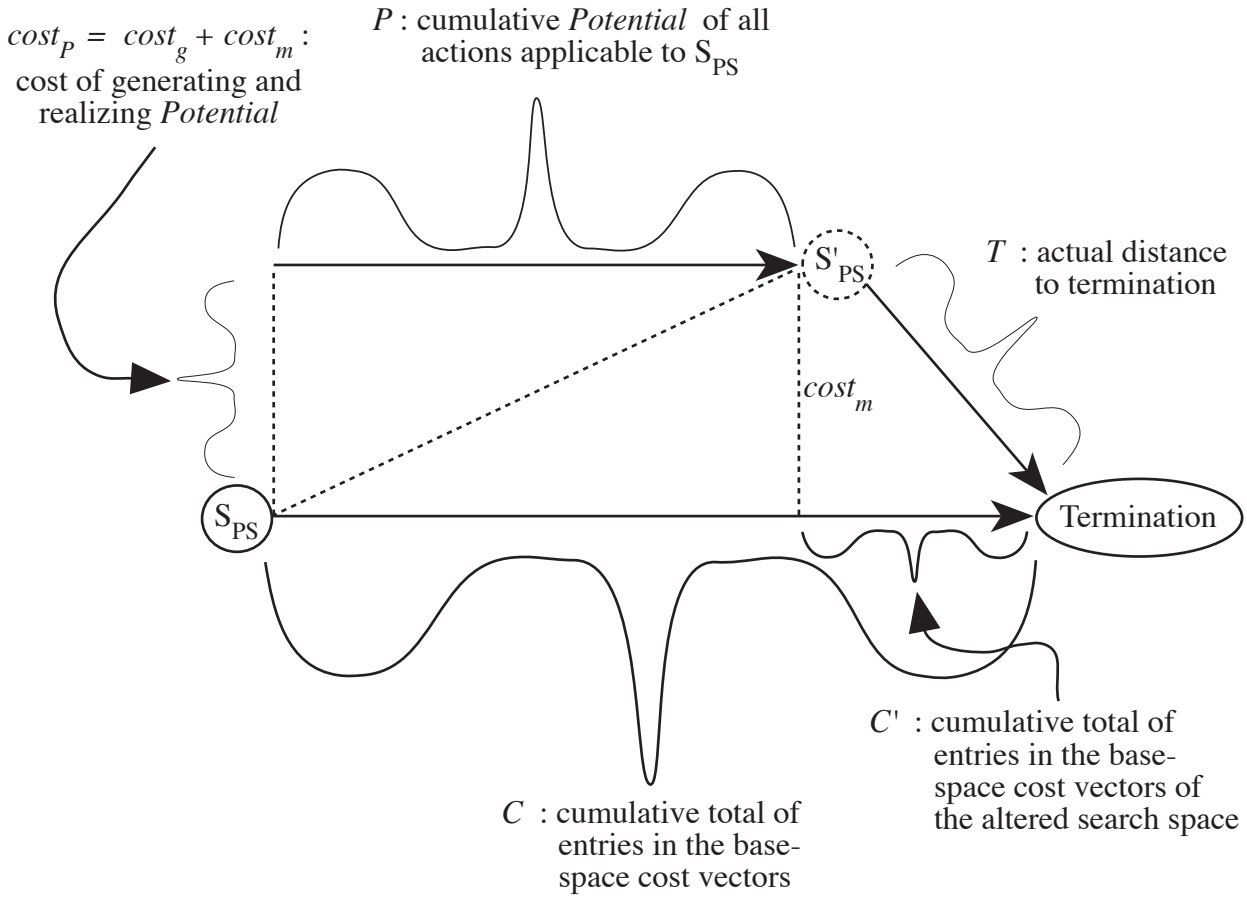


Figure 75: A Basic Representation of Potential and Distance to Termination

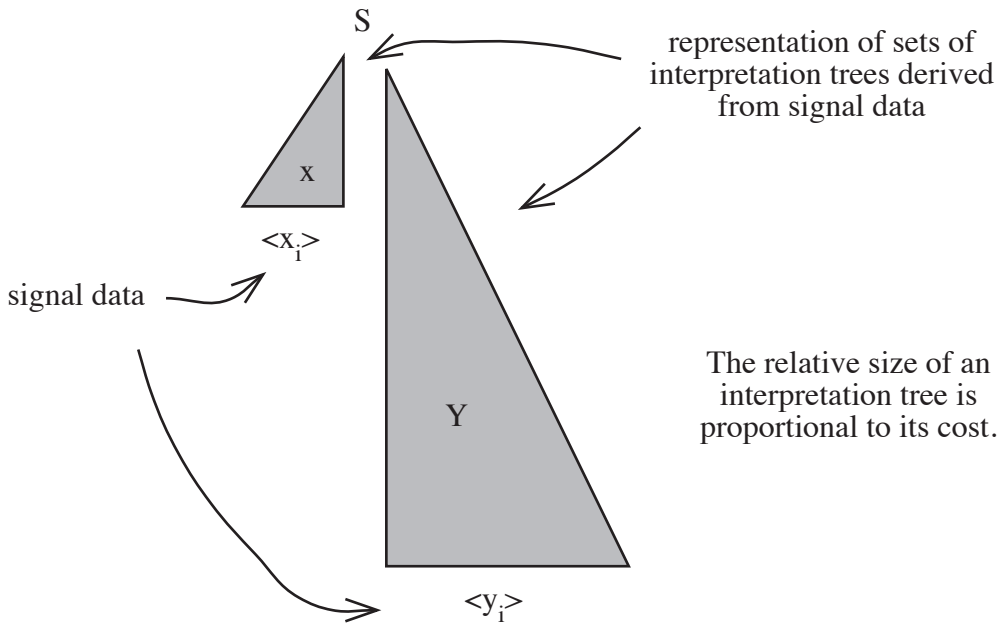


Figure 76: Implied Information Associated with a State

expected cost of problem solving given the existence of state s_n at time t and state S' at time t' and where $t < t'$.

This definition is significant for several reasons. Perhaps the most important is the distinction it makes between $C_{s_n,t}$ and $E(C)$. This distinction is necessary because the existence of s_n may imply a great deal about the nature of a search space. For example, consider the situation shown in Fig. 76. The figure illustrates two different sets of interpretation trees, one which includes the terminals x_i and one which includes the terminals y_i. The interpretations that include an x_i do not include a y_i, and vice-versa. Assume that a given problem instance is equally likely to involve an interpretation from either set and that the expected cost of generating interpretations for problem instances from the X set is much less than the Y set.

In this scenario, assuming that there is no overlap between the two sets of interpretations, $E(C)$ will be an average of the expected costs of interpretations of each of the sets. However, this average will be much larger than the expected cost for interpretations from the X set and it will be much smaller than the expected cost for interpretations from the Y set.

Now consider what happens when a specific state, x_j is generated. The problem solver would be able to determine that the expected distance to termination should be based on the average cost of interpretations for the set of X interpretations, not on $E(C)$. Likewise, if a specific state y_k is generated, the problem solver would be able to determine that the expected distance to termination should be based on the average cost of interpretations for the set of Y interpretations. Consequently, it should be clear that the exact computation of $Potential(S')$ must consider the implications the existence of the local state, s_n, has on the expected distance to termination. The exact computation must similarly consider the implications S' has on the expected distance to termination.

Furthermore, the time, t at which a state, s_n is created and the time, t', at which a state, S' is expected to be created must also factor into the computation of $Potential(S')$. If t is relatively large, indicating that the distance to termination is short, the expected benefits associated with the existence

of S' may be minor. For example, all the paths that may have been pruned given the existence of S' may have already been generated. Obviously, the same is true of t'.

On the other hand, if t is relatively small, then the benefits associated with the existence of S' may be more significant. This is an especially relevant point in domains where a state's potential is a function of its credibility. The the early existence of s_n may indicate that it has an exceptionally high credibility and, if there is a strong correlation between the credibility of s_n and the credibility of S', S' may have an exceptionally high potential.

It may also be necessary to consider the characteristics of a state directly. For example, rather than making inferences about the nature of a search space based on implied information about the relative value of a credibility, it may be necessary to use a direct consideration. This may also extend to other properties of a state, depending on a domain.

We have extended the algorithms presented in Section 6 so that they can be used to calculate $C_{s_n,t}$ and $C_{S',t',s_n,t}$. The extensions involve using s_n and S' to map the grammar, IDP$_G$, used to calculate $E(C)$, to a new grammar IDP$_{G'}$ that is used to calculate $C_{s_n,t}$ or $C_{S',t',s_n,t}$. Similarly, the values of t and t' (and other relevant characteristics of a state, such as credibility) are used to map IDP$_{G'}$ to another new grammar, IDP$_{G''}$. These mappings do not involve the creation of entirely new grammars. Rather, they primarily involve adjusting the distribution functions in the existing grammar.

The first mapping and expected distance to termination calculation is accomplished as follows:

- For state s_n, determine the likelihood of reaching all the potential final states (SNTs) on paths from s_n.
- Normalize these values.
- Replace the distribution function ψ_S, which specifies the distribution of the SNTs in the grammar, with the normalized values. For SNTs that are not on paths from s_n, set their distributions to 0.
- Replace the distribution function ψ_{s_n}, which specifies the distribution of subtrees generated from s_n, with 0. This indicates that the state, s_n, has already been generated and the associated cost should be excluded from further estimations of distance to termination. In effect, this makes s_n a terminal symbol (if it is not already a terminal symbol).
- Calculate C_{s_n} using the algorithm for calculating $E(C)$ from Section 6.1.

The second mapping and expected distance to termination calculation is a little more complex. It requires that a fundamental transformation be made to the original grammar, IDP$_G$. The new grammar will be represented as IDP$_{G''}$ The transformation is shown in Fig. 77. In effect, for each nonterminal of the grammar, N, a new rule is added with the form $N \rightarrow N^{nt} \mid N^t$. In the new rule, N^{nt} represents a nonterminal and N^t represents a terminal symbol. In all other rules of the grammar that include N on the left-hand-side of the rule, N is replaced with N^{nt}. (Note that there should not be any recursive rules, so N will not be on both the left and right hand side of the same rule.) The distribution function, ψ_N, specifies whether an N should be interpreted as a nonterminal (N^{nt}), in which case it can be used to generate additional interpretation subtrees, or whether an N should be interpreted as a terminal symbol, N^t. When N is interpreted as a terminal symbol, it means that the state already exists and the costs associated with deriving it should be ignored.

To calculate $E(C)$, the distribution function for N will always generate N^{nt}. This will result in a grammar that produces problem instances that are identical to the original grammar, IDP$_G$. After

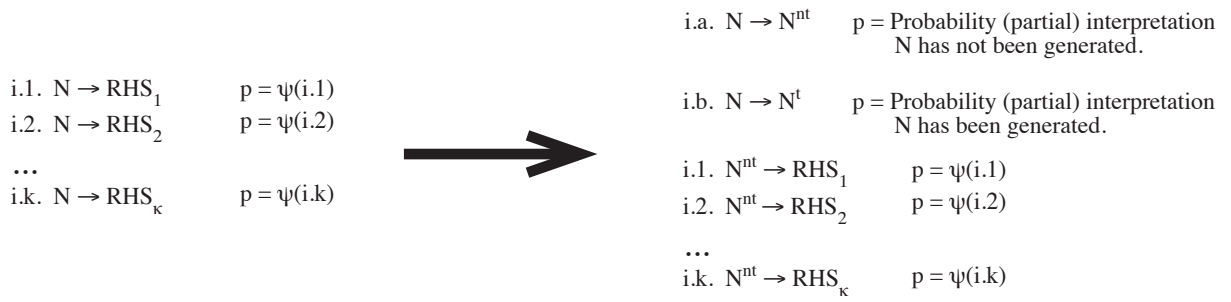


Figure 77: Grammar Transformation for Calculating Potential

some time, t, the current state of problem solving is modeled by adjusting the distribution function for N. The adjustment is a function of time, t, and other factors including the characteristics of a state such as credibility. The adjustment will increase the probability that an N generates an N^{nt}. This corresponds to the probability that the partial interpretation corresponding to N has been generated at time, t. Now, to calculate the expected distance to termination, $C_{S',t',s_n,t}$ the following procedure is used:

- For each nonterminal element, N, of the grammar, adjust the distribution function ψ_N, which specifies whether or not a partial interpretation corresponding to N has been generated. The adjustment is made based on a function of time, t, and other relevant factors such as the credibility of the state s_n, etc.
- Calculate $C_{S',t',s_n,t}$ using the algorithm for calculating $E(C)$ from Section 6.1. (Note that the calculation of $C_{s_n,t}$ must also use this technique.)

In the experiments described in this paper, we use approximations of the methods described in this section. These approximations are described in subsequent sections.

12.2 An Example of Potential

Consider the implications associated with the structure of a complex search space. In particular, consider the fundamental structure of a convergent search space. In these domains, the measures relating a state to the potential final states it implies are not dependent solely on the characteristics of the state. They also depend to a large degree on the existence, and characteristics, of other intermediate states in the search space. As a consequence, the calculation of the measures must be made dynamically.

This will be illustrated with a simple example using the problem solving grammar introduced in Section 4. The rules of this grammar are shown again in Fig. 78.

In this grammar, A and B are the SNTs, or solution-nonterminals, that define the potential final states in the search space. For this example, assume that S's distribution functions are both 0.5. Thus, the start symbol generates an A with probability 0.5 and a B with probability 0.5.

Figure 79 represents the *UPC* values for paths from a state, h, of the convergent search space derived from this structure. The two potential final states associated with h are A and B. Some of the necessary details to compute full *UPC* values for the paths from h to A and from h to B have been omitted, and the values for cost and utility are shown as functions as a result. However, in this example we will concentrate on the values for the expected probability of successfully reaching a potential final state. It

Interpretation Grammar *G'*

0. S → A | B	2. B → DEW
1. A → CD	4. E → jk
3. C → fg	6. W → xyz
5. D → hi	8. j → (signal data)
7. f → (signal data)	10. k → (signal data)
9. g → (signal data)	12. x → (signal data)
11. h → (signal data)	14. y → (signal data)
13. i → (signal data)	15. z → (signal data)

Figure 78: Interpretation Search Operators Shown as a Set of Production Rules

should be clear that, given an h, the probability of generating an A is the same as the probability of generating a B. Both are 0.5.

Though this is a relatively trivial example, it is, by definition, a complex domain since the search paths interact. Specifically, consider the case of the problem solver's internal database containing both an h and an f. These states interact in that they only coexist in one situation, when the complete interpretation is an A. This is because, by definition, an interpretation must explain all the data and, in this example grammar, an A and a B cannot be generated in the same derivation. Therefore, the only interpretation that includes both an h and an f is A. This observation is represented in Fig. 79. Here, an abstract state called f, h is shown along with the associated *UPC* values.

Now consider the implications of this observation, some of which are illustrated in Fig. 80. In this figure, we assume that the expected cost of generating an f or an h is 1, the expected cost of generating an A is 8, and the expected cost of generating a B is 11. For this example, we will ignore credibility calculations.

In Fig. 80, the distance to termination is determined by summing the expected costs of the paths from states h and f to their associated potential final states. For this example, we will use the simplified calculation:

E(cost from s_n to F) $= P(F \mid s_n) * (E(cost_F) - cost_{s_n})$

i.e., the expected cost of a path from intermediate state s_n to a potential final state F is the probability that the final state can be reached from s_n multiplied by the expected cost of deriving the final state minus the cost of generating s_n. (Adjusting the expected cost of generating the final state is necessary because the cost of generating the intermediate state, s_n, is included in the expected cost of generating the final state, but this cost has already been incurred and will not recur as a path is extended from s_n to the final state. A full discussion of calculating *UPC* values accurately is given in Section 9.)

Now consider the implications of the existence of the abstract state f, h. Given this state, the problem solver would know that B could not be a solution (for the reasons discussed above) and it could adjust the *UPC* values for states f and h accordingly. The relationship represented by the state f, h is therefore an important one. To exploit this relationship, an operator would be added to the grammar that creates the state f, h and a mapping operator would also be added that would propagate the effects of f, h back to the base space. In this situation, the existence of states f and h would be associated with a new operator. However, the execution of this operator and the corresponding mapping function, are not related to the cost of termination in the sense that they directly reduce the

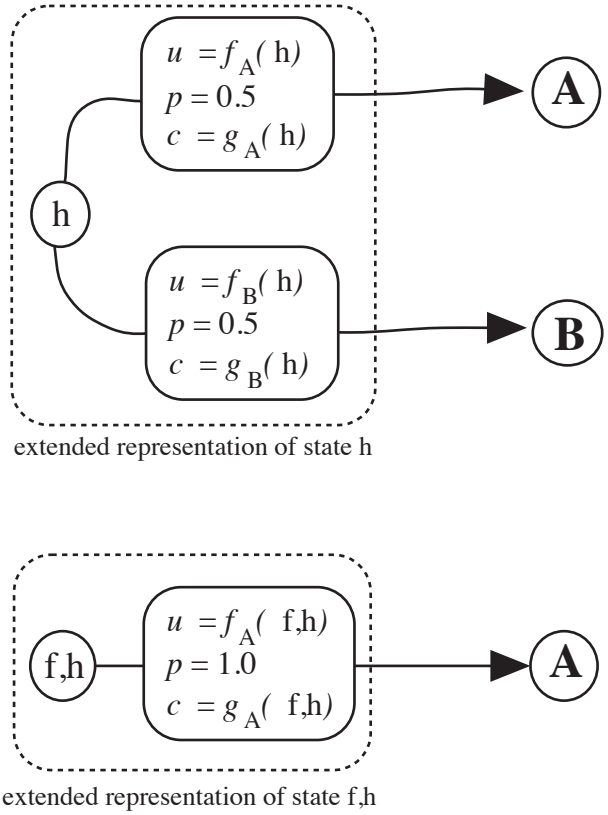
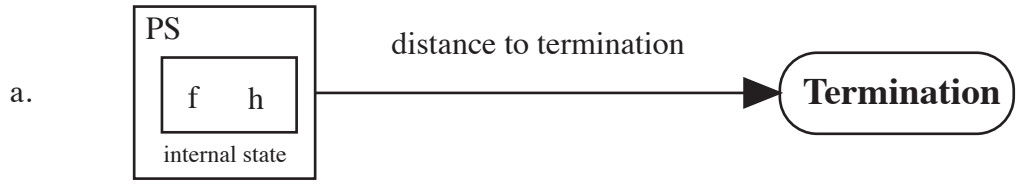


Figure 79: Representation of the *UPC* Values for Base- and Abstract States



Computation of distance to termination:

open state: f ⇒ potential final state A

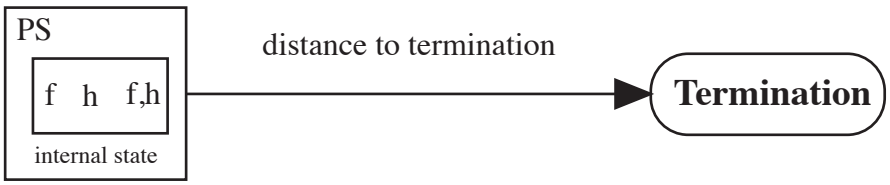
cost associated with open state f =
P(A | f)*(cost(A) - Cost(f)) = 1*(8 - 1) = 7

open state: h ⇒ potential final states A, B

cost associated with open state h =
P(A | h)*(cost(A) - Cost(h)) = 0.5*(8 - 1) = 3.5 +
P(B | h)*(cost(B) - Cost(h)) = 0.5*(11 - 1) = 5

Total distance to termination = 7 + 3.5 + 5 = 15.5

Figure 80: Calculating Distance to Termination



Computation of distance to termination
(after state f,h has been generated and
mapped back to the base-space):

connected state: f ⇒ nil

cost associated with
connected state f = 0

open state: h ⇒ potential final states A

cost associated with open state h =
P(A | f,h)*(cost(A) - Cost(h) - Cost(f)) =
1*(8 - 1-1) = 6

P(B | f,h)*(cost(B) - Cost(h) - Cost(f)) =
0*(11 - 1 -1) = 0

Total distance to termination = 6

Figure 81: Effects of Abstract Processing on Distance to Termination

cost of termination by extending paths in the base space and, as a consequence, the problem solver's control component has no way to evaluate their worth compared with alternative actions. Thus, the question becomes, how do we represent the value of the operator that generates the state f, h? This value is what we refer to as *potential* and we will compute it as the expected reduction in cost associated with the existence of a state, or $P(state) * (\text{expected-cost-reduction})$.

The calculation of potential is illustrated in Fig. 81. In Fig. 81, the distance to termination is calculated based on the *UPC* values of states f and h after they have been modified to reflect the existence of state f, h. The difference between the distances to termination for the two examples is $15.5 - 6 = 9.5$ In other words, the expected cost to termination from the problem solving states in Fig. 80 is 9.5 units more than the expected cost to termination from the problem solving states in Fig. 81. If we assume that the cost of generating state f, h is 1 and the cost of mapping the implications of f, h is also 1, then the expected cost reduction associated with f, h is $9.5 - 1 - 1 = 7.5$. Furthermore, the probability that f, h can be generated given an h is 0.5, so, from state h, the potential of the operator that will generate state f, h is $0.5 * 7.5 = 3.75$.

It is also important to observe that abstractions and approximations are useless without operators designed explicitly to exploit them. In other words, it does no good to climb a tree for a better look at the surrounding territory if you have no way to get down from the tree. In the IDP/*UPC* framework, we refer to the functions that transfer the results of problem solving in an abstract space back to the base space as *mapping functions.* In Section 12.3, the mapping functions will be critical components for calculating the potential of an action. In other words, the potential of a given action will be based on the degree to which subsequent mapping functions can exploit the results of the action to modify the base space in a way that reduces the cost of problem solving.

12.3 Incorporating Potential in Control Decisions

Using Definition 12.1, we have altered the objective strategy used in the simulator to include a consideration of an action's potential as well as its direct effects on the distance to termination, C. Based on the definition for the calculation of potential from Section 12.1, we have developed mechanisms for calculating potential for two forms of processing, bounding functions and approximate processing operators. (The experiments described in this paper we use approximations of potential. The approximate calculations were used in order to decrease the computational cost of the experiments. We are currently developing experimental systems that are not based on approximate computations.) In this section, we will discuss the approximate calculation of the potential associated with bounding functions. As discussed previously, the paths to states with potential are all independent, so their costs are determined individually and summed where appropriate, and the sets of states affected by mapping potential back to the base space are completely interacting, so the maximum potential is used to determine $Pot(s_n, S')$ where appropriate. In the next section, we will present the preliminary results of experiments that incorporate this information in their problem solving decisions.

The original objective strategy was defined in Section 8.2 and it was based on choosing the operator that maximized the equation $\frac{c(op_i)}{cost(op_i)}$, where $c(op_i)$ is the degree to which op_i directly reduces C and $cost(op_i)$ is the cost of executing op_i. The evaluation function for operators used for this set of experiments was changed to $\frac{(c(op_i)+Pot(op_i,s_n))}{cost(op_i)}$, where $Pot(op_i, s_n)$ is the potential of operator op_i applied to state s_n.

The potential associated with bounding functions is based on the expected utility of the final states, F_i, that can be reached along paths including op_i. To calculate this potential, the problem solver first calculates the cost of problem solving in a search space where the bounding functions have a cutoff threshold equal to the expected credibility of the potential final states, F_i, that can be reached along paths including op_i. This calculation is made for each of the potential final states and will be called C_{F_i}. For each of the final states, F_i, its potential, $Potential(F_i)$, can be approximated by $C - C_{F_i}$, where C is the current expected cost to termination. (Note that the exact value of $Potential(F_i)$ is given by $C_{s_n,t} - C_{F_i,s_n,t}$, where $C_{s_n,t}$ is the expected cost of problem solving given that state s_n has been generated at time t, $C_{F_i,t',s_n,t}$ is the expected cost of problem solving given that state s_n has been generated at time t and that state F_i has been generated at time t', and $t < t'$.) Next, the problem solver references the expected cost of reaching the final states from the *UPC* vectors. This is the cost of generating the potential. From Definition 12.1 this quantity is referred to as $cost_g(F_i, s_n)$. For the grammar used in these experiments, the bounding functions are "built in" – they are executed as part of the normal course of problem solving and are already factored into the calculation of C_{F_i}. Therefore, the cost of realizing the potential, $cost_m(F_i)$, is 0.

Given that a specific potential final state can be reached, the actual distance to termination is then $C_{F_i} + cost_g(F_i, s_n)$, and the *net potential* of the action is the probability that the specific potential final state can be reached multiplied by $Potential(F_i)$ minus the cost of reaching the final state, or $P(F_i \mid s_n) * (Potential(F_i) - cost_g)$. For a given operator, the maximum value is taken for all potential final states that can be reached along paths from s_n that include the operator. This value is used in the problem solver's operator rating function as $Pot(op_i, s_n)$.

To summarize this calculation, the following are the steps used to compute the approximate potential of an operator, op_i, applied to a state, s_n:

1. Determine C, the current expected cost to termination. $C = E(C)$– expected-cost(s_n).

2. For s_n, determine the expected utility of the potential final states, F_i, that can be reached along paths from s_n starting with op_i. This is done dynamically by referencing the *UPC* values of s_n. The expected utility of F_i appears in the utility vector corresponding to op_i. Each F_i corresponds to an S' in Definition 12.1.

3. For each of the potential final states, F_i, compute C_{F_i}; the expected cost of problem solving for an IDP with bounding functions that use a cutoff threshold equal to the expected credibility of F_i. $C_{F_i} = E(C_{F_i})-$ expected-cost(F_i). The cost of problem solving given F_i is the cost of connecting the search space using a bounding function cutoff threshold equal to F_i's utility minus the cost of generating F_i, which already exists.

4. For each of the potential final states, F_i, compute $Potential(F_i)$ = C - C_{F_i}.

5. For each potential final state, F_i, that can be reached from s_n, the cost, $cost_g(F_i, s_n)$, of generating the associated $Potential(F_i)$ is the expected cost of a path from s_n to F_i. This value is referenced in the *UPC* values.

6. Finally, $Pot(op_i, s_n) = max_{\forall F_i} P(F_i \mid s_n) * (Potential(F_i) - cost_g(F_i))$.

12.4 Initial Experiments with Potential

We have implemented the process for calculating potential described above and we have used it in several experiments. These initial experiments were intended to explore the effects of the use of potential. In these experiments, $E(C)$ was calculated using a pruning threshold that is a function of the expected credibility of the "correct" result of a problem solving instance and the equations and procedures defined in Section 6. In general, this will yield an approximation of the actual value of $E(C)$. However, in situations where the final state(s) used to specify a pruning threshold is (are) generated before any of the states that they eventually prune are created, this calculation will be exact. This is not an unreasonable assumption to make, even in non-monotonic domains. This is especially true if the pruning threshold is a function of the credibility of the final state(s) such as "pruning threshold = 0.5 * best credibility of the final states generated so far."

In these experiments, conditional probabilites and values for $\Omega(s)$ for all elements of the grammar are computed a priori. The problem solver computes expected credibilities, costs, and probabilities dynamically, using the previously computed values for conditional probability and Ω.

The most striking effect is in the increased time required to conduct an experiment. Given that the rating of every problem solving action requires costly computations, i.e., the computation of C_{F_i}, this result was not surprising. Furthermore, we have found that the cost of the computation of $C(S'_{PS})$ can be reduced significantly through the use of dynamic programming techniques.

Another significant result is that the use of Potential substantially reduces the cost of problem solving. This is consistent with our intuitions that bounding functions reduce the expected cost of problem solving, but the grammar used in these experiments was designed specifically to demonstrate the advantages of bounding functions and this result may not be consistent with real-world performance. (To demonstrate the advantages of bounding functions, potential interpretations derived using noise and missing data rules were given credibility ratings significantly lower than other paths. In addition, the distribution of noise and missing data in the generation of problem instances was less than 10%. In other words, most of the correct interpretations derived did not involve the use of noise or missing data rules.)

Exp	Generation				Interpretation				Sig	% C
	G	Dist	U	$E(C)$	G	Dist	U	Avg. C		
1	1	even	0.5	201	1	even	0.5	200	N	100
18	4	even	0.5	179	4	even	0.5	182	N	100
19	5	even	0.5	2,231	5	even	0.5	2,234	N	100
20	6	even	0.5	1,639	6	even	0.5	1,658	N	100

Abbreviations

Exp: Experiment
G: The problem solving grammar used;
1: G'
4: G' and dynamic bounding functions with cost 1
5: $G2$
6: $G2$ and dynamic bounding functions with cost 0.01
Dist: Distribution of Domain Events;
even: domain events evenly distributed
U; expected problem instance credibility;
0.5: problem instances have expected credibility 0.5
$E(C)$: Expected Cost of problem solving for given grammar
Avg. C: actual average cost for 100 samples of 50 random problem instances each
Sig: Whether or not the difference between expected cost and
the actual average cost was statistically significant
Y: yes, there is a statistically significant difference
N: no, there is not a statistically significant difference
% Correct: percentage of correct answers found

Table 4: Results of Verification Experiments – Potential

Table 4 summarizes experiments 18, 19, and 20; the experiments with potential and bounding functions. The experimental methods were identical to those used in the first set of experiments.

12.4.1 Experiments 1 and 18

In Experiment 18, the grammar used was identical to previously used versions of G' except that the bounding function used was based on dynamic thresholds. Experiment 1 is included for comparison. In 18, the problem solver ran faster with the bounding functions and it did not mistakenly prune any correct paths. Compared with Experiment 1, the performance improvement is noticeable. It is interesting to note that the use of dynamic bounding functions improved the percentage of correct answers generated when compared with Experiments 8 through 17.

12.4.2 Experiment 19

For this experiment, we extended the grammar G' by including new nonterminals that have RHSs with multiple occurrences of nonterminals from G'. This extension was intended to approximate a grammar

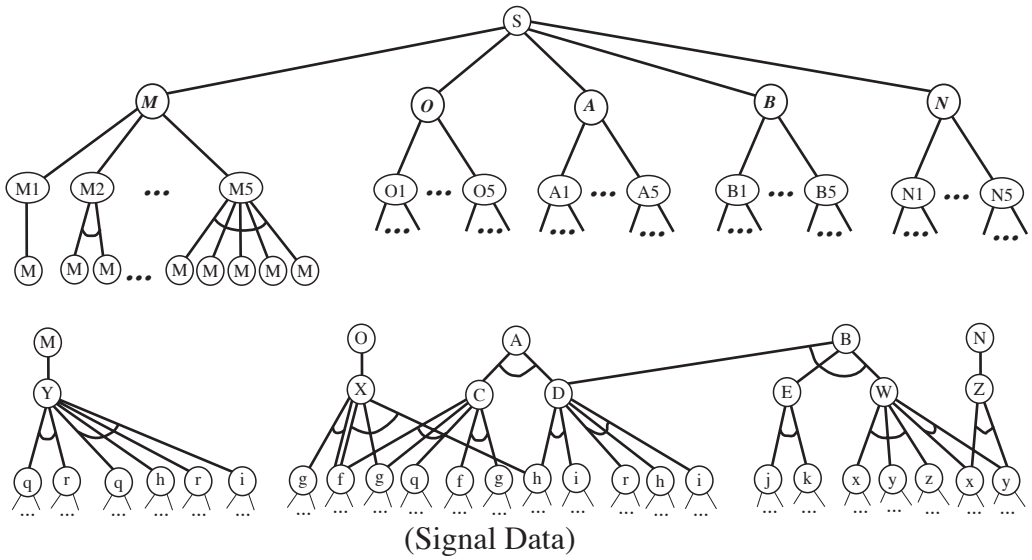


Figure 82: Interpretation Grammar $G2$

representative of a vehicle tracking domain such as the domain associated with the DVMT [6] For each of the SNTs of G', M, O, A, B, N, we added nonterminals of the form A1, A2, A3, A4, A5 where the symbol could be any one of the SNTs and the number indicates the number of time-locations in a vehicle track. For example, one of the rules that was added is $A5 \rightarrow AAAAA$. We refer to this grammar as $G2$. Figure 82 shows the expanded grammar. The SNTs in this grammar are A, B, M, N, and O. As seen in the table, these additions to the grammar increased the expected cost of problem solving considerably. No bounding functions were used in this experiment.

12.4.3 Experiment 20

In this experiment, we added dynamic bounding functions with cost 0.01 throughout the grammar. The effects of this are quite dramatic. Specifically, the expected (and actual) cost of problem solving decreased by approximately 33% when compared with Experiment 19. It is also interesting to note that the percentage of correct answers found was 100.

13 Open Issues

We are currently investigating a number of open issues related to the concepts discussed in this paper. One of the issues is related to the *cost* of computability. The techniques for computing potential used in the experiments described in this paper are too expensive to be used in a real problem solving system. Consequently, the cost of determining *UPC* values in general is too expensive for real problem solving systems. We are attempting to address this issue by defining more comprehensive search space structures (which we refer to as *macro-structures*) that characterize the overall state of problem solving at various points and that can be used to efficiently estimate the effect a precise calculation of potential would have on *UPC* values. Such an estimate would have a level of accuracy that would approximate some form of optimal processing. To define the macro-structures, we are formalizing the concepts of *control uncertainty* and *solution uncertainty* in terms of the *UPC* formalism.

Using dynamic estimates of control uncertainty and solution uncertainty, we are addressing the general issue of, "When is it worthwhile for the control component to use some calculated estimate of potential in its decision making?" In some domains, it may be advantageous to do this for every single search state. In other domains, it may only be necessary to do this for certain states. For example, a domain may be structured in such a way that the problem solver can determine what action to take for certain states a priori. Similarly, the problem solver may be able to determine the best course of action for states with certain characteristics that are determined dynamically. For example, the problem solver may be able to determine a priori that it should always prune states with a credibility lower than some threshold. Intuitively, it is clear that in many situations the decision to include a calculation of potential will need to be based on the current set of states or on dynamically determined properties of the current set of states.

More specifically, we are addressing this issue by defining the concept of ρ that characterizes the degree to which the estimates of *UPC* values used by a problem solver deviate from the values that would be derived from an ideal domain theory, i.e., from values that take precise calculations of potential into consideration. In many ways, ρ is analogous to the correlation between U, P, and C vectors derived from an ideal domain theory and the estimates of these values used by a problem solver, $\hat{U}$, $\hat{P}$, and $\hat{C}$. However, ρ's representation is based on the additional problem solving cost incurred resulting from inaccuracies in the estimation of *UPC* values and is "inverted" compared to correlation measures. Thus, in situations where $\rho = 0$, $\hat{U}$, $\hat{P}$, and $\hat{C}$ are equivalent to U, P, and C and the problem solving will be optimal in the sense that it will be equivalent to problem solving based on *UPC* values derived from the ideal domain theory. In situations where $\rho > 0$, $\hat{U}$, $\hat{P}$, and $\hat{C}$ and U, P, and C are different and problem solving will not be optimal. Over long periods of problem solving, a problem solver with a ρ value that is greater than 0 will incur unneeded costs and/or generate less than optimal utility.

In general, the performance of a problem solver will vary in a way that is inversely proportional to the value of ρ. When ρ is small (i.e., close to 0), the problem solver will perform well. When ρ is large, the problem solver will perform poorly. Furthermore, when the performance of two problem solvers is compared, the problem solver with the higher ρ value will incur additional costs and/or generate less utility.

The concept of ρ is related to the computation of potential (and *UPC* values) because we are using it to analyze the advantages and disadvantages associated with expending resources to obtain a better estimate of *UPC* values. We are addressing issues such as, "When should a problem solver improve its estimates of *UPC* values by executing information gathering actions and when should it simply extend paths in the base search space?"

In general, we expect that there are many ways to determine when potential is an important consideration for the control decision making process. These ways are all sensitive to the structure of a search space including the operators that exist, both base-level and meta-operators, and their interactions with other operators.

Another issue that we are addressing is associated with the analysis tools we define in this paper and their applicability to complex real world problems. We are addressing this issue by implementing an IDP grammar for the RESUN problem solving system[22]. We will use this implementation to test the existing analysis tools and to derive new ones. One of the important problems is the use of infinite grammars needed to characterize real-world problem domains. We are addressing this issue by defining analysis tools that are used for data that is divided into time slices of fixed length. We then increase the length incrementally and attempt to identify if the output of the analysis tools approaches some asymptotic level.

As we examine more complex real-world domains, we are beginning to deal with the issue of developing design theories based on an increasingly sophisticated understanding of problem structures. In particular, through experimentation and analysis, we are finding many commonalities between disparate domains that seem to indicate that certain control architectures can be used to form a basis for a very powerful and general form of problem solving. Specifically, it is becoming apparent that approximate processing [28, 9] and goal processing [4, 5, 26, 27] are forms of processing that can be generalized to many real-world domains. This general strategy is one in which a problem solver uses approximate processing operators to gain a comprehensive view of the data it must interpret, then uses this perspective to guide subsequent problem solving. We are currently examining the implications of these observations and we are attempting to discover principles that will lead to the development of design theories supporting the automatic construction of problem solving operators such as approximate processing operators. Eventually, we hope to demonstrate how this use of the IDP and *UPC* formalisms will support the synthesis of new, more flexible control architectures.

As discussed in previous sections, the expression "optimal processing" used in this paper describes a problem solver's intent more so than its actual performance. We have not yet proven that the problem solving techniques based on potential that we use are actually optimal in the sense that they always define the best course of action. However, we will continue to use the definition of optimal processing presented in this paper as a baseline for comparative purposes while we address extensions to this notion of optimal processing.

In this paper, we described experiments where the IDP models used by the problem solver are different from the same parameters in IDP models that accurately predict the generation of domain data. These investigations involved generation/interpretation grammars with rules that, though they might have different parameters, had identical right- and left-hand-side elements. We are extending the analysis framework to support closed form analysis of the performance of systems where the IDP model used by the problem solver differs more significantly from an IDP model that accurately predicts the generation of domain data. In particular, we are developing analysis tools for situations where, in addition to differences in parameter values, the rule sets of the generation and interpretation grammars are different.

14 Conclusion

The most important conclusion that can be drawn from the work presented in this paper is that the performance of a problem solving system can be analyzed based on formal descriptions of the structure of the domain in which the problem solver is applied and the structure of the sophisticated control architectures used by the problem solver. This analysis can take the form of both predicting the performance levels of the problem solver using analytically derived closed form equations and explaining the performance levels relative to the causes represented as structures in the formal models of the domain and the sophisticated control architectures. In addition, the framework introduced to support this analysis can also be used for conducting experiments to determine the structure of a specific domain or the objective strategy being used by a problem solver.

The framework that is introduced is composed of two formalisms, the *Interpretation Decision Problem (IDP)*, which models the characteristics and problem structure of a domain, and the *UPC* formalism, which provides a general model of control and problem solving. We showed how domain structures such as noise and missing data are represented in the IDP formalism and how the *UPC*

formalism explicitly represents the effects of these structures so that certain relationships between a state and the potential final states that it implies can be quantified in a way that can be exploited by an evaluation function used to rate alternative problem solving actions. In addition, we have shown how the traditional notion of a search space can be extended to incorporate abstract and approximate states, and the operators that create, modify, and exploit them, in a unified representation including traditional forms of search-based problem solving. This representation has been made possible by the introduction of the concept of potential, which is a mechanism that allows us to understand the interrelationships that exist between the current set of states (i.e., the search paths that have been created so far) and the states that can be derived from them. This includes an understanding of the long-term effects of an action. Potential enables meta-operators associated with abstract or approximate states to be evaluated in a manner that is consistent with the evaluation of problem solving actions that directly extend partial solutions.

The formal analysis techniques have been validated in the experiments we present. In addition, we experimentally explored issues associated with the use of incorrect domain theories. Specifically, we examined the performance implications resulting from situations where a problem solver has a model of a domain that differs from the actual domain. We examined deviations in credibility structures and in distribution structures and we showed that these effects could be predicted, explained, and understood in terms of structures defined with the IDP/*UPC* framework as another way of validating the basic correctness of our framework.

The formal approach that we have adopted in this work and the emphasis we place on the structure of a problem domain and its relationship to problem solving performance has given us new insight into the nature of search based problem solving. In particular, problem solving and the sophisticated control techniques that are based on the use of abstractions and approximations can now be viewed from a unified perspective. This unified perspective has important implications for analyzing control architectures such as approximate processing, goal processing, and other meta-level control architectures. Using the IDP/*UPC* framework, or other formal approaches, it is possible to compare and contrast alternative architectures in a range of different domains in order to develop an understanding of how a particular architecture might be generalized. This ability could eventually shift the emphasis of AI research to the exploration, and formalization, of the characteristics of natural problem domains.

Finally, in developing the IDP/*UPC* framework, we have developed a better understanding of sophisticated problem solving and its relationship to other forms of problem solving. The IDP/*UPC* framework was designed intentionally as an extension to the framework introduced by Kanal and Kumar [25] and used to develop a taxonomy of search-based problem solving techniques. By extending this framework to include the sophisticated control mechanisms used in AI problem solving, it is now possible to view many AI control architectures as generalizations of problem solving techniques described formally in other fields, such as operations research. We intend to explore this issue in subsequent publications.

15 Acknowledgments

We would like to thank Norm Carver for the contributions he made during the preparation of this paper. We would also like to thank Frank Klassner and Tuomas Sandholm for their comments and suggestions and for proof reading this paper.

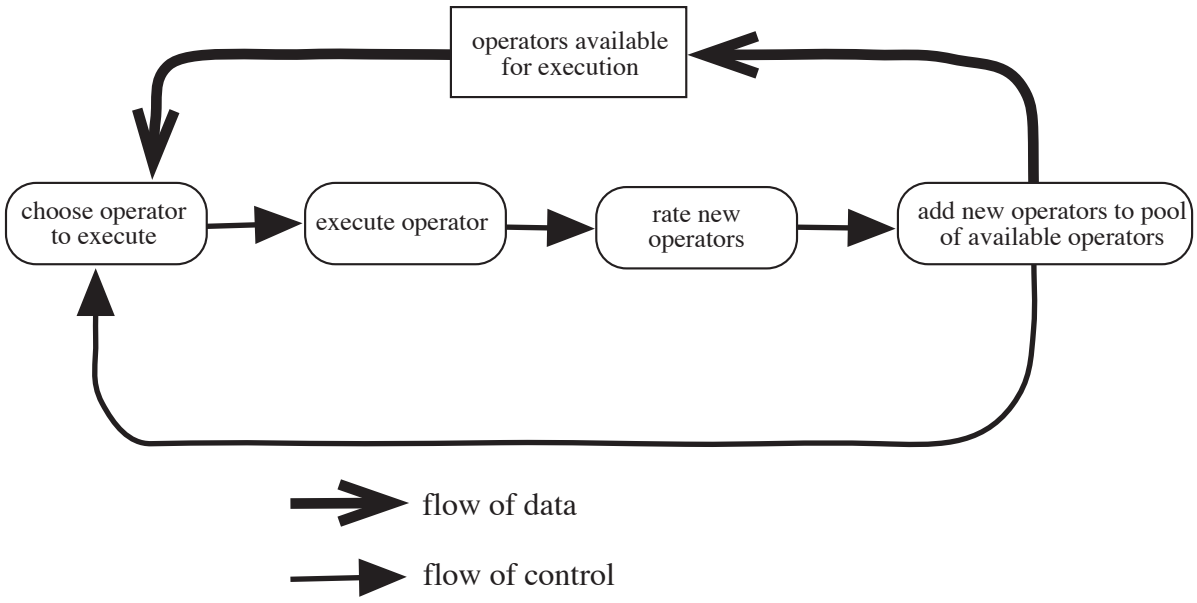


Figure 83: The Basic Control Cycle

Appendix

A General Objective Strategies and the *UPC* Formalism

This section will define a general taxonomy of objective strategies in terms of the *UPC* model that can be used to analyze alternative problem solving strategies. To accomplish this, the process of *control* will now be defined as *the process of searching for the best operator to apply.* The space searched by the control mechanism is defined by the *UPC* vectors and will be referred to as *operator space.* When an operator is executed, it usually causes other operators to become eligible for execution. These are rated and then added to a pool of available operators. This pool is identical to operator space. In the IDP/*UPC* framework, the search for the best operator is a linear search of operator space. (In the actual implementation, newly rated operators are inserted into a list of operators sorted by their evaluation function ratings. Consequently, the process of search operator space for the best operator is equivalent to selecting the first element of the sorted list.) The basic control cycle used in the IDP/*UPC* framework is reproduced from a previous section in Fig. 83.

The evaluation of an operator is based on the objective function used by a problem solver. Thus, investigating the properties associated with alternative objective strategies, which we define as the *experimentation(agent objective strategy) analysis paradigm* in Section 2, involves the use of alternative evaluation functions. For the experiments described in this paper, the objective strategy used is based on the optimal strategy defined in Section 8. The following subsections specify a taxonomy of objective strategies based on the use of evaluation functions and *UPC* values. This taxonomy can be used in the analysis framework to construct control architectures and simulation tools to investigate the effects of alternative objective strategies. Ultimately, we hope to use this taxonomy to generalize control architectures to new domains and for other general analytical purposes.

A.1 Total Utility Optimality (TUO)

The TUO strategy is to maximize the total amount of utility generated regardless of cost. In the TUO strategy, there is an implication that the problem solver should find the optimal utility as efficiently as possible. (The IDP optimality specification of Section 8 is essentially the TUO strategy with explicit efficiency requirements.) This strategy can be thought of as always choosing for expansion the path that most efficiently leads to the maximum expected utility. Formally,

Definition A.1 *TUO Objective Strategy -* $max_{\forall ni}(s_n(u_i))$*, where* $s_n(u_i)$ *is the expected utility of potential final state* i *that can be reached by extending a path from* s_n*.* The problem solver will choose the operator that extends a path leading to the potential final state with the highest expected utility. In the case of ties, the problem solver will choose the path with the lowest expected cost.

The entries in the *UPC* vectors are *expected values* and that the actual values vary. Consequently, a problem solver will not know the real utility of a final state until it reaches that state and, once a problem solver reaches a final state that is optimal according to the estimated values, it cannot, without additional processing, rule out the possibility that a different final state exists with a higher utility. Given that a domain may exhibit extreme variance, a problem solver must take this fact into consideration. This may require the problem solver to conduct additional search to *prove* that a specific final state has the highest utility.

The TUO strategy is best suited for domains that do not include some consideration of cost when computing the utility generated by the system. More specifically, the TUO strategy is not well suited for domains that require real-time responsiveness.

A.2 Utility per Unit Cost Optimality (UUCO)

This strategy is to pursue a course of action that maximizes the expected utility per unit of cost expended. Formally,

Definition A.2 *UUCO Objective Strategy -* $max_{\forall n,i}(\frac{(s_n(u_i)*s_n(p_i))}{s_n(c_i)})$*, where* $s_n(p_i)$ *is the expected probability of generating potential final state* i *by extending a path from state* s_n*, and* $s_n(c_i)$ *is the expected cost of the path to potential final state* i*.* The problem solver will choose the operator that is expected to generate the most utility per unit cost.

The UUCO strategy is best suited for domains where the ultimate utility generated by a problem solver is a function of cost. This is especially applicable for domains where time is the primary cost and where there are implicit (or explicit) real-time responsiveness requirements. For example, for a speech understanding system, there are implied requirements that the system respond to spoken input in a timely fashion.

Situations where this strategy is appropriate are sometimes described as *satisficing* problems [36]. For interpretation tasks, satisficing strategies are based on the assumption that a "good" interpretation, i.e., one that is within some ϵ of the optimal (or "correct") interpretation, has a semantic interpretation that is very similar to that of the correct interpretation. As a result, the problem solver's best course of action may be to return a good answer in a timely manner, rather than an optimal answer requiring significantly more resources to derive.

The UUCO strategy is not well suited for domains where processing costs are irrelevant or where it is imperative that the highest quality solution be found.

A.3 Minimum Cost (MC)

The MC strategy is to minimize the total cost of reaching a final state.

Definition A.3 *MC Objective Strategy* - $max_{\forall n,i}(s_n(c_i))$. The problem solver will choose the operator that extends the search path with the least expected cost that leads to any final state.

This strategy can be used when any final state is acceptable, especially for domains where real-time responsiveness is required.

B Mapping Strategies

In the experiments that are described in this paper, and in the examples and related discussion, the mapping strategies that are used do not create any new states in the base search space. They only update *UPC* values based on the information derived from abstract states in projection search spaces. This, however, is not a requirement of the IDP/*UPC* framework, it is merely the convention that has been used for this paper. The following is a taxonomy of classes of mapping strategies that we have identified and that can be incorporated in the IDP/*UPC* analysis framework in ways that are consistent with the definitions presented in this paper.

Sophisticated Control - The most straight forward form of mapping is when the problem solver uses the results of search in projection search spaces to increase the accuracy of the *UPC* estimates in the base search space and thereby increase the efficiency of problem solving. In this method, states in the base search space are linked to states in a projection space and the mapping function uses information from states in the the projection search space to modify *UPC* values in the base space. Problem solving in the base space is then directed by a standard control architecture, such as TUO.

As described previously, this is the mapping strategy used throughout this paper. The process of mapping a state in a projection space to states in the base space is represented by meta-operators of an IDP grammar with an abstract state as a RHS of a base space state. The base space state on the LHS of the rule defining the meta-operator defines the information that is used to update *UPC* values, and the component set of the abstract state on the RHS of the rule defines the states that have their *UPC* values modified.

Refinement - An alternative approach is to modify the control architecture to include operators that map states in a projection space back to newly created states in the base space. In general, these operators are very costly and can be thought of as collections of operators from the base space. (As described previously, the semantic functions corresponding to the base space operators must be executed to generate an interpretation.) Consequently, refinement can also be thought of as transforming a search problem to a *sequencing problem*. This method essentially transforms the abstract solution into a *plan* for solving the base search problem. Such a plan would have well defined tasks that correspond to base space operators as well as mechanisms for verifying that the plan was working.

Constraint Directed Search - A hybrid approach involves treating the projection space solution as a constraint network, such as the constraint networks defined by Fox [13]. Selective refinement

is used to map portions of the projection space solution back to the base search space, or sophisticated control based search can be used to achieve a similar result. The partial result that is generated in the base space is then projected back to the abstract space and incorporated into the abstract solution. The constraints implied by the more precise partial result from the base space are then propagated to the components of the abstract solution. This process can continue until a solution is determined.

Approximate Processing (Satisficing) - Another direct method for using the results of projection space search is to return them as the actual result of problem solving. This method is applicable for domains that admit satisficing solutions [36]. In particular, approximate processing can be used when an acceptable solution can sacrifice precision, certainty or completeness [28, 29, 9].

References

[1] Hans Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12:23–40, 1979.

[2] Norman Carver and Victor Lesser. The Evolution of Blackboard Control. *Expert Systems with Applications*, 7(1), 1991. Special issue on The Blackboard Paradigm and Its Applications (also available as Technical Report 92-71, Computer Science Department, University of Massachusetts, 1992).

[3] Norman Carver and Victor R. Lesser. Planning for the control of an interpretation system. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1993. Special Issue on Scheduling, Planning, and Control.

[4] Daniel D. Corkill and Victor R. Lesser. A goal-directed Hearsay-II architecture: Unifying data-directed and goal-directed control. Technical Report 81-15, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, June 1981.

[5] Daniel D. Corkill, Victor R. Lesser, and Eva Hudlická. Unifying data-directed and goal-directed control: An example and experiments. In *Proceedings of the National Conference on Artificial Intelligence*, pages 143–147, Pittsburgh, Pennsylvania, August 1982.

[6] Daniel David Corkill. *A Framework for Organizational Self-Design in Distributed Problem Solving Networks*. PhD thesis, University of Massachusetts, February 1983. (Also published as Technical Report 82-33, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, December 1982.).

[7] Randall Davis. Meta-rules: Reasoning about control. *Artificial Intelligence*, 15:179–222, 1980.

[8] Keith Decker, Marty Humphrey, and Victor Lesser. Experimenting with control in the DVMT. Technical Report 89-00, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, March 1989.

[9] Keith S. Decker, Victor R. Lesser, and Robert C. Whitehair. Extending a blackboard architecture for approximate processing. *The Journal of Real-Time Systems*, 2(1/2):47–79, 1990. Also COINS TR-89-115.

[10] Edmund H. Durfee. *A Unified Approach to Dynamic Coordination: Planning Actions and Interactions in a Distributed Problem Solving Network*. PhD thesis, University of Massachusetts, September 1987. (Also published as Technical Report 87-84, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, September,1987.).

[11] Edmund H. Durfee and Victor R. Lesser. Incremental planning to control a blackboard-based problem solver. In *Proceedings of the National Conference on Artificial Intelligence*, pages 58–64, Philadelphia, Pennsylvania, August 1986.

[12] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12(2):213–253, June 1980.

[13] Mark S. Fox. *Constraint-directed Search: A Case Study of Job-Shop Scheduling*. PhD thesis, Carnegie-Mellon University, 1983. (Also published as Technical Report CMU-CS-83-161, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213.).

[14] King Sun Fu. *Syntactic Pattern Recognition and Applications*. PH, 1982.

[15] Gerald Gazdar, Geoffrey K. Pullum, and Ivan A. Sag. Auxiliaries and related phenomena in a restrictive theory of grammar. *Language*, 58(3):591–638, 1982.

[16] Michael R. Genesereth. An overview of meta-level architecture. In *Proceedings of the National Conference on Artificial Intelligence*, pages 119–124, Washington, D.C., August 1983.

[17] Michael R. Genesereth and David E. Smith. Meta-level architecture. Technical report, Computer Science Department, Stanford University, Stanford, California 94305, December 1982.

[18] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, July 1985.

[19] Barbara Hayes-Roth and Frederick Hayes-Roth. A cognitive model of planning. *Cognitive Science*, 3(4):275–310, October–December 1979.

[20] Frederick Hayes-Roth and Victor R. Lesser. Focus of attention in the Hearsay-II system. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 27–35, Tiblisi, Georgia, USSR, August 1977.

[21] Eva Hudlická and Victor R. Lesser. Meta-level control through fault detection and diagnosis. In *Proceedings of the National Conference on Artificial Intelligence*, pages 153–161, Austin, Texas, August 1984.

[22] Norman F. Carver III. *Sophisticated Control for Interpretation: Planning to Resolve Sources of Uncertainty*. PhD thesis, University of Massachusetts, September 1990.

[23] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, Carnegie Mellon University, 1991. (Also published as Technical Report CMU-CS-91-120, School of Computer Science, Carnegie Mellon University.).

[24] Craig A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 686–691, San Diego, California, July 1991.

[25] Vipin Kumar and Laveen N. Kanal. The CDP: A unifying formulation for heuristic search, dynamic programming, and branch-and-bound. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, Symbolic computation, chapter 1, pages 1–27. Springer-Verlag, 1988.

[26] V. R. Lesser, D. D. Corkill, R. C. Whitehair, and J. A. Hernandez. Focus of control through goal relationships. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, August 1989.

[27] V. R. Lesser, R. C. Whitehair, D. D. Corkill, and J. A. Hernandez. Goal relationships and their use in a blackboard architecture. In V. Jagannathan, Rajendra Dodhiawala, and Lawrence Baum, editors, *Blackboard Architectures and Applications*, pages 9–26. Academic Press, Inc., 1989.

[28] Victor R. Lesser and Jasmina Pavlin. Performing approximate processing to address real-time constraints. COINS Technical Report 87-126, University of Massachusetts, 1988.

[29] Victor R. Lesser, Jasmina Pavlin, and Edmund Durfee. Approximate processing in real-time problem solving. *AI Magazine*, 9(1):49–61, Spring 1988.

[30] Marvin Minsky. Steps towards artificial intelligence. In E. A. Fiegenbaum and J. Feldman, editors, *Computers and Thought*, pages 406–450. McGraw-Hill, 1963.

[31] A. Newell, J. C. Shaw, and H. a. Simon. The process of creative thinking. In *Contemporary Approaches to Creative Thinking*, pages 63–119. Atherton Press, New York, 1962.

[32] A. Newell, J. C. Shaw, and H. a. Simon. Empirical explorations with the logic theory machine: A case history of heuristics. In E. A. Fiegenbaum and J. Feldman, editors, *Computers and Thought*, pages 109–133. McGraw-Hill, 1963.

[33] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[34] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, first edition, 1984.

[35] A. L. Samuel. Some studies in machine learning using the game of checkers. In E. A. Fiegenbaum and J. Feldman, editors, *Computers and Thought*, pages 71–105. McGraw-Hill, 1963.

[36] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, 1969.

[37] Mark Stefik. Planning and meta-planning (MOLGEN: Part 2). *Artificial Intelligence*, 16:141–170, 1981.

[38] G. C. Stockman. A minimax algorithm better than Alpha-Beta? *Artificial Intelligence*, 12:179–196, 1979.

[39] Robert Wilensky. Meta-planning: Representing and using knowledge about planning in problem solving and natural language understanding. *Cognitive Science*, 5(3):197–233, July–September 1981.