Constraint Manipulation for Example Generation [1]

Daniel D. Suthers and Edwina L. Rissland [2]

COINS Technical Report 88-71

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Abstract

Given a set of weighted and possibly conflicting constraints on feature dimensions used to describe examples, constrained example generation is the task of producing an example that meets a subset of these constraints which is in some sense optimal. ExGen achieves this through a mixture of retrieval of a previously known example and modification of it to meet constraints not satisfied by retrieval. We discuss the basic architecture of ExGen, and algorithms to manage constraint interaction problems through re-arrangement and protection of items on the constraint list, including application of Kernighan-Lin variable depth search techniques.

[1]This research was supported in part by the National Science Foundation under grant number MDR 8751362, and benefited from discussions with Robbie Moll and Beverly Woolf.

Contents

1 Introduction

The ability to generate examples is required in a variety of applications, such as in assistants for generating adversarial arguments, where one needs to pose hypotheticals having certain characteristics [Ashley & Rissland 1987, Rissland & Ashley 1987]; in mathematics, where one needs counter-examples to refute and refine conjectures [Lakatos, 1976; Polya, 1973; Rissland, 1984]; human tutoring [Tennyson et al., 1972; Tennyson, 1973; Tennyson et al, 1975], and intelligent tutoring systems, where one teaches concepts by giving examples to the (human) learner [Brown, Clement, & Murray 1986; Murray, T., Schultz, K., Clement, J., & Brown, D. (in press); Woolf, Suthers, & Murray 1988]; and in machine learning [Winston, 1975; Bareiss & Porter, 1987; Rissland, 1988, and Buchanan et al., submitted], where the training examples can greatly influence the performance of the machine learner. Example generation is also useful in some planning domains, where an automated reasoner may need to generate cases on which to test its hypotheses or plans [Wall & Rissland, 1982].

This paper describes a domain-independent example generation tool, called ExGen, focusing on its architecture and algorithms, including the use of local search to improve the generated example. ExGen is a successor of the "constrained example generation" architecture of Rissland [1980]. It was developed to be embedded in a "client" program like an interactive tutor. ExGen is being tested and improved in the context of tutors in the domains of thermodynamics and statics in introductory physics. It has also been tested in the "boxes" domain used as an example application in this paper. We foresee its use by other programs such as case-based reasoners that depend heavily on a modification of past cases to solve a new problem situation [Kopeikina et al, 1988]. Thus, its design emphasizes the need to respond in time which is tolerable to the user. Cognitive modeling was not a goal of ExGen.

In ExGen, a domain of examples is defined by a finite number of **feature dimensions**, each of which may have an arbitrary number of possible **values**. An **example** is a vector that specifies a value for each of these feature dimensions. We call a feature/value binding an **attribute** of the example. Given a **query**, which is a list of

weighted constraints on attributes, some of which may conflict with each other, ExGen returns an example which satisfies a maximal subset of these constraints, under a notion of maximality to be discussed. In doing so, ExGen draws on an **Example Base** of known examples, retrieving a close match and modifying it to better match the query. The modification knowledge is embodied in procedural attachments to the feature dimensions used to describe examples. Constraint interaction problems are managed through re-arrangement and protection of items on the constraint list. When an example is created through modification, it is cached in the Example Base, which "grows" as needed to adapt to the circumstances encountered by the client system calling ExGen.

In section 2, we discuss how ExGen compares to previous work in constraint satisfaction and example generation. Section 3 contains a description of the architecture and algorithms of ExGen, and illustrations in a simple "box" domain. Having laid the groundwork, we discuss the application of local search techniques to reordering the query in an attempt to increase the "value" of the returned example, in Section 4. In particular, we examine the applicability of "variable depth search" [Lin & Kernighan 1973] to our constraint reprioritization problem. As this technical report is intended to moonlight as a users manual, footnotes are used where needed to reconcile the abstract treatment of the main body of text with the actual implementation, and there is an appendix on obtaining and using ExGen. The language used in the illustrations is Common Lisp, typeset in `typewriter font`.

2 Background

The present work is related to previous research in constraint satisfaction [Sussman & Steele 1980] and Truth Maintenance Systems (TMS) [Doyle, 1979], but differs in ways crucial to the intended applications. Constraint networks typically require that sufficient information be provided to fully constrain the unknown values. An example generator needs to respond to potentially underconstrained queries for an example. The Example Base in ExGen serves as a way to utilize past experience to provide reasonable default values. The propagation of changed values in ExGen may be seen as a special case of a TMS. However, while a TMS is designed to inform you of the consequences of your inferences, and in particular of when inconsistencies occur, ExGen avoids inconsistent modifications during the construction of an example. As a result, there is no backtracking. ExGen's modification algorithm is less general than constraint propagation and TMS schemes, but at the gain of efficiency for interactive applications.

ExGen uses a retrieve and modify paradigm which originated in Rissland's [1980] analysis of people's behavior while generating examples of mathematical functions and of LISP lists. Her subjects tended to start with an example which was close to the given specifications, and then modify it to meet the remaining requirements. This motivated the original "constrained example generation" or CEG architecture [Rissland, 1980] utilizing modules for retrieval, modification, and construction of examples. In CEG, an Executive module controls an agenda-based approach, with the help of a Judgment module to determine whether an example matches the given constraints. The Executive first tries to satisfy the constraints through the Retriever, and calls the Modifier only when this fails. If modification also fails, the CEG executive calls on the services of a Constructor to build the example from scratch. The major points of difference between ExGen and CEG are given below.

Weighted, Conflicting Constraints. While CEG used discrete constraints, ExGen uses weighted constraints, and when they conflict attempts to optimize the value

of the example based on these weights, through reasoned re-prioritization of the constraints.

Dynamic Reasoning about Retrieval. For a given problem, CEG used a fixed retrieval order, for instance, one based on *a priori* taxonomic knowledge (e.g., consider "reference" examples before counter-examples before "start-up" examples etc.). In ExGen, retrieval takes into account what the Modifier can and cannot do.

Dynamic Reasoning about Modification. The CEG modifier applied modification operators based on their fixed ordering in an operator table. ExGen uses weights on constraints to choose the modification order. The Modifier works on the constraints in an order which reflects their interdependencies as well as their individual priorities.

Modification Operators. CEG chose difference reducing operators from a table indexed by some difference between the current example and the constraints. In ExGen, the Modifier determines which feature dimension must be modified to satisfy the constraint, and invokes a procedure written for modifying that dimension. This is a weaker form of difference indexing which requires correspondingly stronger modification operators.

Unifying Modification with Construction. There is considerable overlap between the knowledge required to modify an existing example to have a given value on some descriptive feature dimension, and that required to construct an example with that value on the feature dimension. For this reason, modification and construction are not separate processes in ExGen. The same mechanism can modify no, few, or all features of the retrieved example, so the distinction between modification and construction is one of degree.

Modifying One Example. ExGen's Modifier is given one example to work on, while CEG's Modifier kept an agenda of examples to modify, and performed a form of search in deciding which would be most fruitful to work on next. In ExGen we are concerned with real-time generation as the user waits, so wish to avoid the computational cost of modifying multiple examples, only one of which will be used. We do this by reasoning about what the Modifier will be able to make the most of, and performing retrieval in service of modification.

3 Architecture of ExGen

This section provides an overview of ExGen's architecture (Figure 1), including its algorithms and knowledge sources. We illustrate the discussion with a "toy" domain about boxes of varying shapes and densities.

3.1 Feature Dimensions

A major component of ExGen's knowledge is a domain epistemology of feature dimensions. A **feature dimension** is a knowledge structure used to specify a descriptive aspect of examples of interest in the domain [Rissland et al., 1984]. Some features are designated **derived** in that they are combinations or abstractions of other features. The abstraction hierarchy is grounded in a set of **primitive** features. The representation of an ExGen dimension includes a declaration of the dependency relations with other feature dimensions, methods for calculating the value of an example on it (in the case of derived features), and methods for modifying an example with respect to the dimension.

For example, consider a simple domain of examples consisting of boxes. The primitive features which fully define a box are its `height`, `width`, `length`, and `density` of the material it is made of. Derived features are: `volume`; `mass`; `size`, a categorization of boxes by `volume` into `small`, `medium`, or `large`; and `type`, a categorization into `cube`, `rod`, `box`, `square`, `rectangle`, `line` or `point`. As shown in Figure 2, `volume` and `type` are direct abstractions of the three primitive features, while `size` and `mass` are abstracted from other derived features.

In addition to representing the physical attributes of examples, the idea of a "dimension" is used to capture the fact that some examples can be ordered with respect to a particular point of view [Rissland et al., 1984; Rissland & Ashley, 1986, 1987]. For instance, the Statics and Thermodynamics tutors use feature dimensions to represent the pedagogical complexity of their examples from the points of view of the various topics which are taught (e.g., complexity of initial energy distribution).

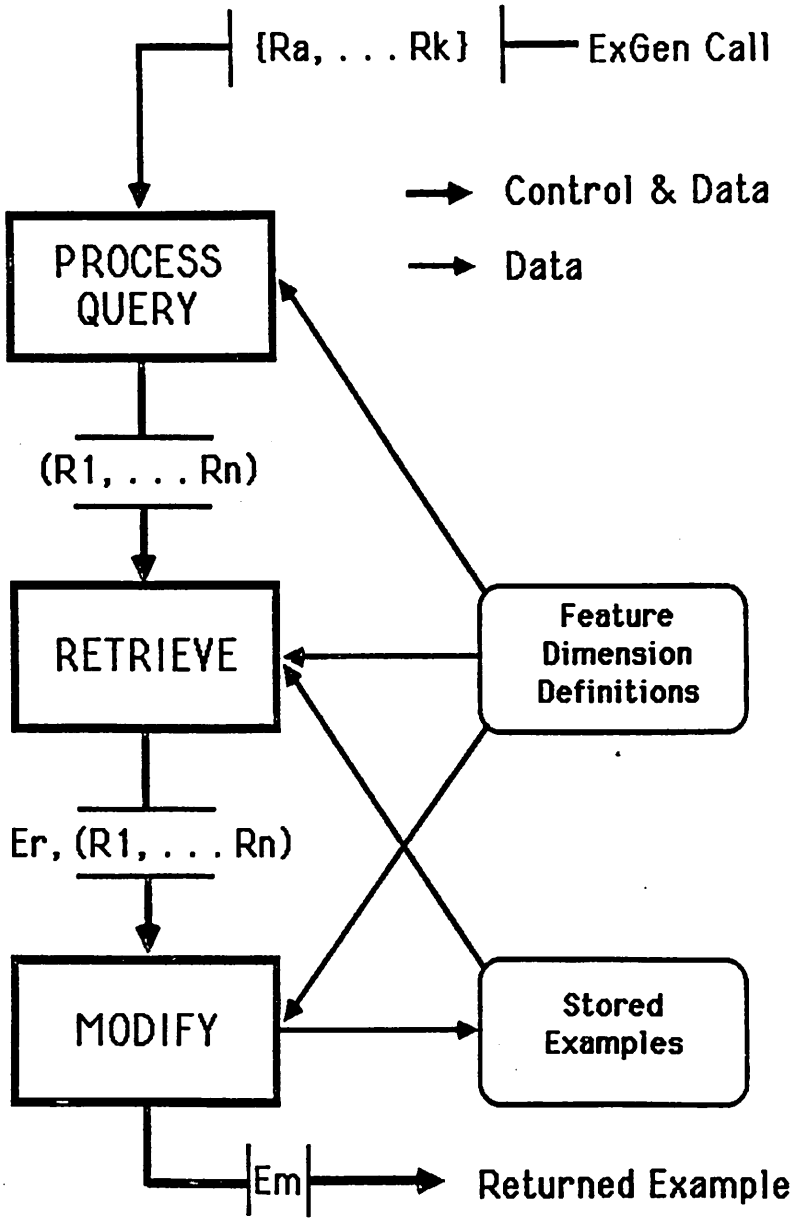


Figure 1: Architecture of ExGen

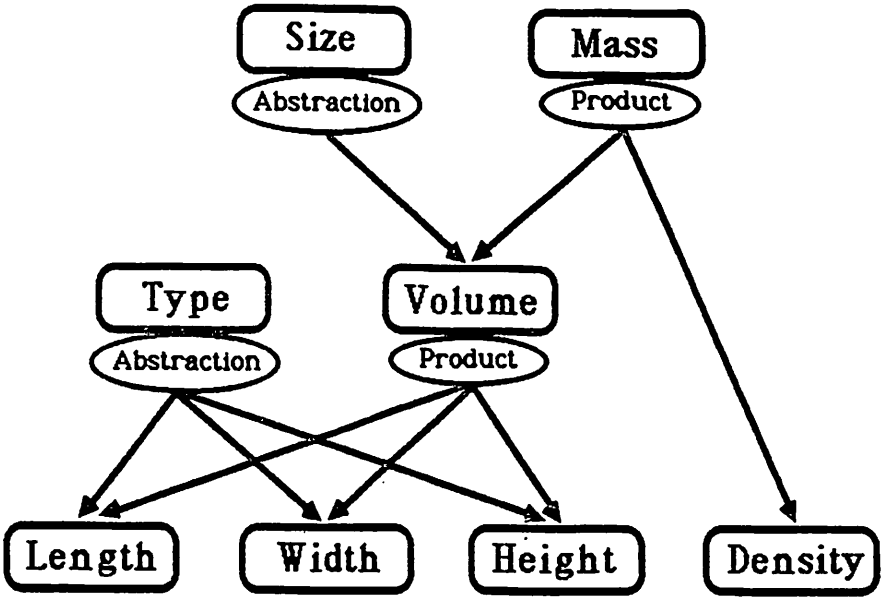


Figure 2: Epistemological Structure of the Box Domain

Each feature dimension is defined by specifying the following components: [3]

If-Needed: A function which maps an example to the value of the example on the feature dimension. `If-needed` is used to determine values for derived feature dimensions, by examining values on other feature dimensions on which the dimension depends. [4]

To-Modify: A procedure specifying how to modify an example along this dimension. The `to-modify` procedures constitute the specific knowledge used by ExGen to modify examples. `To-modify` either modifies an example to have one of a set of requested values and signals success, or signals that it cannot modify the example to have any of these values. The `to-modify` method for a derived feature is written in terms of calls to the `to-modify` methods for the features it depends on. [5]

Depends-On: A list of the feature dimensions on which the feature dimension depends. These are identical to those referenced by the `if-needed` method and which are invoked recursively for modification by the `to-modify` method. Their

[3]There are also `has-index` and `comments` slots. A feature dimension can only be used for retrieval if `has-index` is `T`. You can save space by setting it to `nil` for features you will never reference directly in a constraint (eg. primitives which are only referenced indirectly, via derived features.).

[4]Use the `example-attribute` function, which caches `if-needed` computations used to get the value.

[5]`To-modify` takes three arguments: the example, a list of values, and a list of protected features, and signals success by returning any non-nil value. Use `call-to-modify` for recursive modifications: this function checks for protection violations.

explicit declaration here enables ExGen to reason about the interactions between constraints. The relation induced by the `depends-on` lists must be a directed acylic graph to ensure proper operation of the algorithms.

Sample-Values: A list of some sample values for the feature dimension which allows ExGen to extend the virtual (constructible) example base beyond that which would be possible by enumerating all combinations of values in the initial "seed" examples. Thus, the `sample-values` list aids in knowledge engineering by greatly reducing the number of seed examples needed; it implicitly defines a whole neighborhood of examples derivable from the initial Example Base. [6]

In our box domain, the primitive feature dimensions such as `height` and `density` have no `if-needed` method. Their `to-modify` methods simply set the attribute of the example to one of the legal requested values, and their `depends-on` lists consists solely of the feature itself.

The definition of a derived feature like `mass` (figure 3) has more information concerning how it is abstracted from its underlying primitives. Its `if-needed` function computes the product of `volume` and `density`. Its `to-modify` method converts a request for a particular `mass` into requests for values on `volume` or `density`, and recursively calls the `to-modify` methods for those features.

3.2 Initializing the Feature Dimensions

ExGen is initialized by computing the `protect` and `update` relations, which are particularly important to the operation of its algorithms. [7]

The Protect Relation. When the Modifier determines that the current value on a given feature f satisfies a request, it consults the `protect` slot to determine what features must be protected to disallow violation of the satisfied request.

[6] ExGen is *not* restricted to using known values: if a constraint includes reference to a new value not appearing in the Example Base or in `sample-values`, the `to-modify` method may yet be able to construct an example with this new value.

[7] Additionally, the `if-needed` and `to-modify` methods are compiled. (When debugging, set `*use-compiled-methods*` to `nil` if necessary.) On feature dimensions for which `has-index` is non-nil, inverted indices are constructed which map values on the dimensions to examples having those values. A `precludes` relation is also computed, to be described in Section 4.

```
(feature-dimension :MASS
 :depends-on (:mass :volume :density)
 :if-needed
   (lambda (e)
     (* (example-attribute e :volume) (example-attribute e density)))
 :to-modify
   (lambda (e vals protected)
     (let ((v (example-attribute e :volume))
           (d (example-attribute e :density))
           (mutable (set-difference '(:volume :density) protected)))
       (if (= v 0) (setq mutable (delete :density can-modify)))
       (if (= d 0) (setq mutable (delete :volume  can-modify)))
       (cond
         ((and (member :volume mutable) (member :density mutable))
          (or (call-to-modify :volume
                e (mapcar #'(lambda (m) (/ m d)) vals) protected)
              (call-to-modify :density
                e (mapcar #'(lambda (m) (/ m v)) vals) protected)))
         ((member :volume mutable)
          (call-to-modify :volume
            e (mapcar #'(lambda (m) (/ m d)) vals) protected))
         ((member :density mutable)
          (call-to-modify :density
            e (mapcar #'(lambda (m) (/ m v)) vals) protected)))))
 :has-index T
 :sample-values (1 10 50 100)
 :comments "Mass is the product of volume and density.  Can modify
   unprotected features whose complement is not 0.  Prefer to modify
   volume if there is a choice, but try density if that fails. The
   recursive modification call asks for any value that will achieve
   one of the requested masses.")
```

This illustrates several techniques, including encoding a modification preference, trying the other modification if the recursive call to the first one fails (the `or`), and asking for all values that will get what you want (the `mapcar`), as well as use of the `example-attribute` and `call-to-modify` functions. Success of modification is signaled according to recursive success.

Figure 3: Definition of MASS

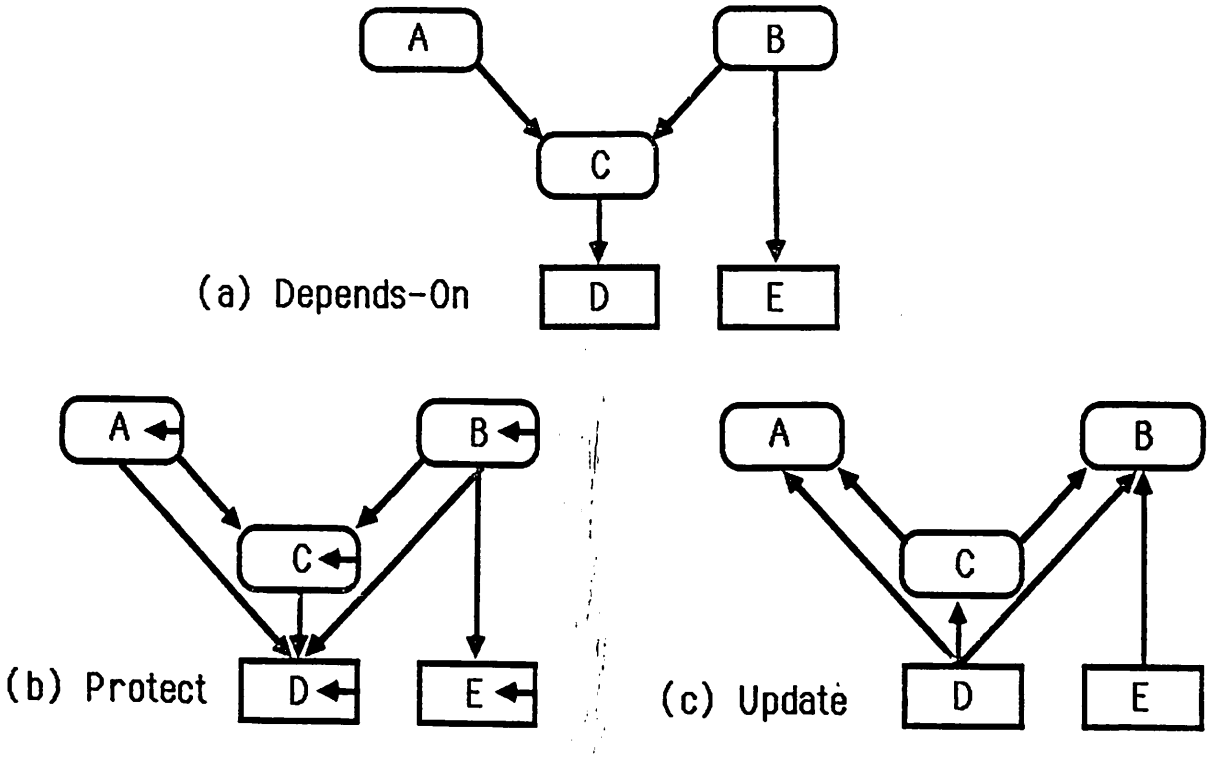


Figure 4: Dependency Relations

Suppose subfigure (a) shows the dependency relations as originally declared; then (b) shows `protect` as the reflexive, transitive closure of `depends-on`, and (c) shows `update` as the irreflexive, transitive closure of the inverse of `depends-on`.

`Protect`(f) is defined to be the reflexive, transitive closure of `depends-on`(f). This closure is computed with a standard depth first search algorithm. ExGen assumes that `depends-on`(f) is initialized to include all feature dimensions that the value of f *directly* depends on, namely all features referenced in its `if-needed` and `to-modify` methods. Figure 4 (b) shows an example of the effect of this closure.

The Update Relation. Since changes must propagate non-locally through the `if-needed` methods, ExGen computes an `update` slot which tells the Modifier which feature dimensions will need to have their values recomputed after a modification succeeds on a feature dimension f. The modification algorithm guarantees that for all features f_i modified recursively during modification of f, `update`(f_i) will be updated. Because of this, `update`(f) need only contain those which directly or indirectly depend

on f itself, not those which only depend on a feature f_i which f is derived from.

`Update(`f`)` is defined to be the irreflexive, transitive closure of the *inverse* of `depends-on`, and is computed by removing reflexivity from the inverse of `protect`. See Figure 4 (c) for an illustration of this closure.

3.3 The Example Base

Since ExGen generates new examples from known ones, it is necessary to provide ExGen with a "seed" Example Base, consisting of a set of standard examples. This initial Example Base should include aspects of examples it cannot generate, since ExGen relies on retrieval for constraints it can do nothing about (e.g., because of constraint conflicts or lack of modification methods). Each example is specified by giving the values on its primitive feature dimensions. Derived features are computed using the `if-needed` methods. ExGen also maintains a `derivation` history [8] which records how each example was constructed from the original seed examples.

As illustration, the following initial Example Base will be used for our box domain:

```
(example CUBE-1       :length 1 :width 1 :height 1 :density 1.5)
(example ROD-1        :length 1 :width 1 :height 4 :density 2.0)
(example BOX-1        :length 4 :width 2 :height 6 :density 1.0)
(example BOX-2        :length 4 :width 1 :height 2 :density 0.5)
(example BOX-3        :length 2 :width 2 :height 4 :density 3.0)
(example SQUARE-1     :length 2 :width 2 :height 0 :density 0)
(example RECTANGLE-1  :length 4 :width 2 :height 0 :density 0)
(example LINE-1       :length 2 :width 0 :height 0 :density 0)
(example POINT-1      :length 0 :width 0 :height 0 :density 0)
```

3.4 Requests to ExGen

A call to ExGen is parameterized by a list of requests, which indicate the constraints on the example to be generated, and their relative importance. Each **request** consists of a constraint and a weight. [9]

[8] A predefined feature dimension.

[9] It also may include identification of the knowledge source which generated the request, recorded solely for the benefit of the client program. This is placed in a *list* in the `source` slot.

A **constraint** is an expression which indicates the desired attributes of the example. The following grammar describes allowable constraint expressions:

```
constraint ::= ()
             | (:and constraint *)
             | (:or constraint *)
             | (:not constraint)
             | (:attrib feature value)
             | (:member value feature)
             | (:holds feature relation value)
```

where * is the Kleene-star. `And`, `or`, and `not` have the standard logical semantics. The expression (`:attrib` *feature value*) means the example has the indicated value on the feature, and (`:member` *value feature*) selects any examples which have list values containing the value. `Holds` requires that the actual value on the indicated dimension holds the indicated relation to the given value, providing a means for arbitrary extension of the constraint language. [10]

The **weight** is an arbitrary real number. In manipulating weights, ExGen assumes interval semantics under summation; for example, a weight of 2 is taken to indicate that the request is twice as important as one weighted 1, and two requests weighted 0.3 each outweigh one request of weight 0.5 when taken together.

For instance, suppose in our box domain we wished to come up with a box high enough to sit on, light enough to lift, and with various other requirements for the sake of illustration: [11]

```
(request :constraint (:holds  #':mass < 25)          :weight 5)
(request :constraint (:attrib :height 5)             :weight 10)
(request :constraint (:not (:attrib :size :large))   :weight 3)
(request :constraint (:and (:attrib :type :box)
                           (:attrib :width 2))
                                                     :weight 5)
(request :constraint (:or (:attrib :size :small)
                          (:attrib :size :medium))   :weight 3)
```

[10] Specifically, (`funcall` *relation actual-value value*) must return a non-nil value.

[11] In most applications, one would not use the `request` macro. Instead, the requests would be created within various knowledge sources using the `create-request` function and collected together into a list.

3.5 Preliminary Parsing of an ExGen Query

For the sake of efficiency and simplicity in writing the Modifier and the `to-modify` methods as well as permitting certain query optimizations (to be discussed), the initial request to ExGen is "parsed" by rewriting and combining certain constraints.

Where possible, a request's constraint is parsed into an equivalent constraint indicating that some particular feature dimension is to have one of a disjunction of values. This succeeds only if all the constraints nested in an `or` expression reference the same feature dimension, and no nontrivial `and` expression is involved. If this fails, the Retriever will use the constraint as is, but the Modifier will ignore it. Parsing of `not`, `member` and `holds` expressions relies on a closed world assumption (CWA) that the currently known values on feature dimensions (those which are either in the Example Base or on the `sample-values` list for a given feature) are all that exist.

Any requests which have identical features and value sets are merged, their weights being combined by addition. [12] The resulting list of requests is sorted by the weights in decreasing order. The result of applying these methods yields the basic query which ExGen will work from.

For instance, the box domain example query after parsing is:

```
(request :feature :height :values (5)              :weight 10)
(request :feature :size   :values (:small :medium) :weight 6)
(request :feature :mass   :values (0 1 1.5 4 8 10) :weight 5)
(request :constraint (:and (:attrib :type box)
                           (:attrib :width 2))     :weight 5)
```

Note that parsing of the `holds`, `attrib`, `or`, and `not` constraints has yielded a disjunction of values on a feature dimensions, while the `and` constraint was not parsable into this form, and so will be usable only to constrain retrieval. Two of the requests, originally stated with apparently different `or` and `not` constraints, have been found to be identical after parsing. These have been merged into one constraint on `size` and their weights added. The values for `mass` were obtained using CWA on the Example Base of Section 3.3 and the `sample-values` list of Figure 3. The requests have been sorted by weight.

[12] The `source` slots are combined as well, using `union`.

3.6 Retrieval in the Service of Modification

The Retriever takes a prioritized list of requests, resulting from preliminary parsing and re-ordering processes to be described, and returns a set of examples which satisfy as many of the requests as possible in the order given in the prioritized list. It initializes a `retrieved` set to the universe of known examples. Then the Retriever processes each request by restricting `retrieved` to those that meet its constraint, but rejects as unsatisfied any request which would cause the `retrieved` set to go empty. Then it goes on to the next, lower priority request, intersecting the sets of satisfying examples as it goes, until all requests have been processed. The resulting retrieved set is guaranteed to be nonempty, provided there is at least one example in the Example Base.

The goal of the Retriever is to present the Modifier with an example which meets as many of the constraints as possible according to the prioritization. If a request was unparsable, or a feature dimension has no `to-modify` method defined for it, a request for a value on that dimension *must* be met during retrieval if it is to met at all, for the Modifier will be helpless. This motivates a manipulation of the query before the retrieval phase is attempted. This is to temporarily re-prioritize the query, placing all requests which are unparsable or having constraints on a feature dimension with no `to-modify` method ahead of the remaining requests, which presumably the Modifier will be able to deal with. [13] The original query is retained for the Modifier's use. The result is that the Retriever will attempt to satisfy constraints that the Modifier cannot be expected to.

Preparation of the box domain query for retrieval results in the movement of the single unparsed constraint to the front of the list (the weights are not shown in the remainder of this section to emphasize that the retriever and modifier consider only the *order* of the prioritized lists):

```
(request :constraint (:and (:attrib :type box) (:attrib :width 2)))
(request :feature :height :values (5))
(request :feature :size   :values (:small :medium))
(request :feature :mass   :values (0 1 1.5 4 8 10))
```

Then retrieval proceeds as follows. The `retrieved` set is initialized to the set of known examples. The first request is met by two examples (from the Example Base previously

[13] In some applications, the dependency structure is such that the Modifier can not achieve enough of the demoted requests, and their loss outweighs the gain of the requests which can be achieved by retrieval only. As this is an application dependent issue, the flag `*prioritize-retrieve-only-requests*` is provided to control this feature.

listed), namely {BOX-1, BOX-3}. Since the intersection of this set and the retrieved set is nonempty, the latter is now assigned this intersection. No examples meet the height request, so this is not used to constrain the retrieved set. The next request, for a size of small or medium, is met by {BOX-3} (assuming the cutoff for small is below a volume of 48), and the retrieved set is again restricted to its intersection with this set. Subsequent requests could not restrict this set further without making it go empty, so BOX-3 is returned as the retrieved example.

3.7 Modification

The Modifier gets from the Retriever one of the retrieved examples, [14] plus the prioritized list of constraints for which modification methods are known (those which were parsed and reference a feature with a to-modify method). Modification proceeds by iterating down the constraint list in the given order, and attempting to satisfy the constraints which have not already been met. Goal protection ensures that satisfaction of a given constraint will not violate a higher-priority request, and dynamic propagation of changes ensures that all values are consistent at each step of the modification process.

The modification algorithm is as follows. A protected-features list is initially empty and a copy of the example is made. Then the constraints in the query are processed in the order given. Each constraint requesting that f_i have one of values $\{v_{i,1}, ...v_{i,k}\}$ is handled as follows:

1. If f_i is protected, the request is ignored.
2. Else if the request has already been satisfied, protected-features is set to its union with protect(f_i).
3. Else a call to to-modify(f_i) is made. It may recursively call to-modify(f_k), for $f_k \in$ depends-on(f_i):
   (a) It is an error if any recursive to-modify call occurs on a protected feature.
   (b) When returning from any successful invocation of to-modify(f_k), all values of features in update(f_k) immediately have their values recomputed by their respective if-needed methods, so that active to-modify methods will see updated and consistent values.

[14] Randomly chosen, in the case that more than one example was retrieved. Randomness is justified when all examples are equivalent with respect to the constraints, provides the user with variety, and ensures that the growth of the example base through modification is uniform in all "directions".

If the top level call succeeds to modify the current example to have one of $\{v_{i,1}, ...v_{i,k}\}$ on f_i, `protect`(f_i) is adjoined to `protected-features`, and features on `update`(f_i) are updated. Otherwise (on failure to modify f_i), nothing is done.

Once all the requests are processed in this manner, if modification has taken place, the new example is indexed into the Example Base to be retrieved and used without modification should a similar situation ever arise in the future. In either case the example is returned.

This algorithm has the property that no constraint which is satisfied, whether by retrieval or modification, will be violated by a constraint which is further down on the constraint list. This is achieved by disallowing direct or recursive invocation of `to-modify`(f) for any f in `protected-features`. The latter list is included as an argument to the `to-modify` methods to provide them with an opportunity to work around any protections on the features they normally change. The desired value may be obtainable by modification of an alternate feature. For example, in the box domain it may still be possible to achieve a given `volume` through modification of `width` and `length`, even though `height` is protected.

Note that it is possible for a recursive modification of f_k to succeed while that for the top level f_i fails due to failure of recursive modification of some other f_j invoked by `to-modify`(f_i). In this case, the new value for f_k is not protected, but is left as is, and so may affect the outcome of the returned example. Thus it is possible to write modification methods which modify an example to be "closer" to the desired value, even when they fail to achieve it. Consistency is maintained through *immediate* updating of features in `update`(f_k) when returning from the recursive `to-modify`(f_k) call.

The box domain example and query given to the Modifier is:

```
(example BOX-3 :length 2 :width 2 :height 4 :density 3.0)
(request :feature :height :values (5))
(request :feature :size   :values (:small :medium))
(request :feature :mass   :values (0 1 1.5 4 8 10))
```

A copy is made of the example, and the `protected-features` list is initially empty. Iterating over the constraint list, the Modifier determines that the first constraint has

not been met, and calls `to-modify(height)`. This succeeds (`height` is a primitive feature, and so may immediately modify its value), so all features on `update(height)`, namely `type`, `volume`, `size`, and `mass`, have their values recomputed with their `if-needed` methods, and `protected-features` is set to `(height)`. Suppose the constraint on `size` is still satisfied: its `protect` list is adjoined to `protected-features` to give `(size volume length width height)`. The Modifier determines that the next constraint is not satisfied by the current value of `mass`, and calls `to-modify(mass)`. The latter method (Figure 3) determines that `volume` is protected, and attempts to achieve one of the desired values by calling `to-modify(density)` with a list of appropriate density values, namely `(0 .05 .075 .2 .4 .5)` (note the use of the updated `volume` in computing these values). One of these is chosen as the new density, say `.4`, and `to-modify(density)` signals success. This causes members of `update(density)` to be recomputed before control is returned to `to-modify(mass)`, which itself then signals success, again resulting in protection and propagation of changes. There are no more requests, so new example is returned: [15]

```
(example <new-name> :length 2 :width 2 :height 5 :density .4)
```

Note that while no attempt was made to protect the unparsed request during modification, its presence in the retrieval query impacted on which example the Modifier was given.

[15]This also has a derivation of ((:parent . BOX-3) (:time . *time-stamp*) (:modifications (*source* . (:attrib height 5)) (*source* . (:holds :mass #'< 25)))).

4 Reordering the Query with Local Search

We have seen that the retrieval and modification algorithms use only the *order* of the constraint list to determine priority, and ignore the weights. There are situations in which a list of requests which is sorted by the weights on the constraints may not yield an optimal result. This section examines the use of local search techniques to re-arrange the order of the query after parsing in order to improve on the "value" of the generated example.

Taking the weights to be a measure of the value of satisfying the requests, we utilize the sum of weights of satisfied requests as a measure of the value of a retrieved example. It may be possible to increase the value of the example with respect to this overall objective function by reordering requests. In particular, since ExGen works to satisfy requests higher on the priority list first, a high priority request r_0 may prevent the satisfaction of several lower priority requests $\{r_1, ...r_k\}$ whose weights sum to more than the weight of r_0. The goal of the constraint manipulation techniques described here is to determine when demotion of r_0 to a lower priority is worthwhile in terms of the impact on the sum of weights of satisfied requests.

4.1 Choice of Granularity

There is a choice in the granularity at which optimization can be attempted. **Value-based** reordering would involve examining the functional relations between *values* requested on each of the feature dimensions, and using a dependency network to determine whether the requests are actually incompatible. This would enable true optimization of the queries. However, in the current architecture it carries the computational cost of executing arbitrary user-supplied `if-needed` methods in the process of determining compatibility using the dependency network, and doing so every time a new query is processed. The time required to do this is too great for our interactive applications.

At a coarser granularity, **feature-based** reordering ignores the particular values requested, and considers only whether it is *possible* that one request will preclude another, given the declared dependencies between the features involved. For example, if the the protected set of some satisfied request r_0 includes the features of a collection of pending requests $\{r_1, ...r_k\}$ whose weights sum to greater than that of the satisfied request, then r_0 could be demoted to be of lower priority than r_k. This would be done only on the basis that r_0 "locks" the features which the remaining r_i want to change, regardless of whether r_0 actually makes it impossible to achieve the *values* asked for by the remaining r_i.

While optimization is not guaranteed under feature-based reordering, the cost of computing feature-based compatibility between requests is quite low, as the needed incompatibility relation is directly derivable from the precomputed `protect` relation. The real-time demands of our applications motivated selection of feature-based compatibility, and ome analytic and empirical work was done to determine the best way to attempt optimization under this approach.

4.2 Formal Description of the Problem

The abstract problem definition assumes a feature-based compatibility function:

$$\texttt{precludes} : F \longrightarrow 2^F$$

where `precludes`(f) denotes the set of features whose further modification is "blocked" by achieving a request on f. In Section 4.5, we discuss the relation of `precludes` to `protect`.

Problem Elements. Let $q = (r_1, ..., r_n)$ be a query of n requests, where each request $r_i = (f_i, w_i)$ is a tuple consisting of a feature dimension f_i in F and a real valued weight. Note that the index on f_i is not meant to imply a $1 - 1$ correspondence between requests and feature dimensions: distinct requests may reference the same feature dimension. To access the components of the tuple, we define the functions $f(r_i) = f_i$ and $w(r_i) = w_i$. We choose the indices on the r_i such that $i > j \rightarrow w(r_i) \geq w(r_j)$ (the greedy initial approximation used by ExGen).

Objective Function. The function we want to *maximize* is derived from feature-based incompatibility, so is called the **F-value** of the query. First we define a function

which tells us whether it is possible for a request to be blocked in a given sequence. Given $q = (r_1, ...r_n)$,

$$s(r_j, q) = \begin{cases} 0 & \text{if } \exists r_i \in q,\ i < j,\ \ni s(r_i, q) = 1, \\ & \text{and } f(r_j) \in \texttt{precludes}(f(r_i)); \\ 1 & \text{otherwise.} \end{cases}$$

Having $s(r_j, q) = 1$ is a sufficient but not necessary condition for r_j to be always satisfiable in q. Now define the F-value as the sum of the weights of the requests which are guaranteed to be satisfiable:

$$\text{F-value}(q) = \sum_{r_i \in q} w(r_i) \cdot s(r_i, q)$$

Optimization Task. Ideally, the task at hand is to find a permutation π of q to some q^* with an optimum F-value across all permutations of q. However, under the given time constraints we may have to approximate this by finding $q_N^+ = (r_{\pi(1)}, ...r_{\pi(n)})$ where q_N^+ is a *local* optimum with respect to the neighborhood N. [16] An *exact* neighborhood is one in which $q_N^+ = q^*$. The choice of neighborhood is thus an important part of the solution.

4.3 Search Algorithms

Here we describe the search algorithms tested in terms of the choice of neighborhood structure, whether or not "variable depth search" was used, and choice of pruning method.

Full Demotion. In this neighborhood, the neighbors of a query q are all permutations of q obtained by moving precisely one *active* request r_i (for which $s(r_i, q) = 1$) further down the list of requests, just past the *last* request r_j which r_i precludes:

$$\begin{array}{lrl} \text{Given:} & q = & (r_1, ...r_n), \\ \text{Define:} & N(q) = \{q' \mid & q' = (r_1, ...r_{i-1}, r_{i+1}, ...r_j, r_i, ...r_n), \\ & & \text{where } s(r_i, q) = 1,\ f(r_j) \in \texttt{precludes}(f(r_i)), \\ & & \text{and } \neg\exists r_k \ni j < k \wedge f(r_k) \in \texttt{precludes}(f(r_i)).\} \end{array}$$

[16] A *neighborhood* is a function defining which alternate solutions a given solution may be transformed into in one step of the search. Thus it defines the structure of the search space.

Moving only active r_i ensures that only permutations which could make a difference are attempted, and placing r_i after all the requests it is in conflict with reduces the size of the neighborhood while "putting r_i in its place". Given n requests in a query, at most n are eligible for reinsertion, and each is reinserted in one unique location. This constrains the neighborhood size to $O(n)$. In practice, neighborhoods are smaller in typical dependency structures, as the requirement that $s(r_i, q) = 1$ eliminates many requests from consideration for movement. This is because there is at least one request r_j further down in q for which $s(r_j, q) = 0$ by virtue of r_i. This neighborhood has been shown to be inexact by counter example.

Minimum Demotion. If the `precludes` relation is symmetric, then moving r_i just after the *first* request it precludes will "turn off" r_i, giving the same score under our abstract problem definition. The Minimum Demotion neighborhood is defined as Full Demotion, except replace $\neg\exists r_k \ni j < k$ with $\neg\exists r_k \ni i < k < j$. It was conjectured that this would be a better neighborhood because it is faster to find the first reinsertion point, yet the value of the resulting neighbor is equivalent whenever `precludes` is symmetric. The size of the neighborhood is again $O(n)$, and it is inexact.

Other less restricted neighborhoods were tested for comparison and rejected, including those in which the $s(r_i, q) = 1$ restriction was lifted, and one which simply reinserts arbitrary r_i at arbitrary locations in q. These did not increase the quality of solution, and greatly increased the number of nodes visited.

Kernighan-Lin (Variable Depth) Search. A **Kernighan-Lin** (KL) neighborhood [Lin & Kernighan 1973; Papadimitriou & Steiglitz 1980] is generated by repeated but restricted application of search under another neighborhood. One must identify meaningful "elements" of a problem solution which are the units manipulated during search. The restriction is that each element is manipulated only once. In our problem, the elements are the requests and the restriction is that each request is moved only once. The solution expanded during the next reapplication of the underlying neighborhood search is the neighbor of maximum value, *even if there was a net loss in value* from the expanded solution to its maximum neighbor. This reapplication of the underlying neighborhood is repeated until no more problem elements may be moved. For example, assume the query is `(a b c d)`. If the Full Demotion neighbor of maximum value is `(b c a d)` (moving `a`), then Full Demotion search will be reapplied to `(b c a d)` except that neighbors moving `a` are not considered. The result of the KL neighborhood search is the solution of maximum value in the *sequence* of solutions visited, which need not be the last solution in the sequence. KL search has been found

to be a powerful technique in many applications, due to its ability to get out of local minima by allowing temporary loss in value.

Kernighan-Lin neighborhoods were used with both Full Demotion and Minimum Demotion as the underlying neighborhoods. Two search algorithms treated the KL neighborhood as in ordinary search, and applied KL repeatedly. Since each KL neighborhood is itself a complete search, it was conjectured that a *single* application of KL may be adequate, giving rise to two more search algorithms tested.

A single Kernighan-Lin neighborhood search repeats the $O(n)$ Full Demotion or Minimum Demotion search at most n times, moving each request at most once, and so is clearly $O(n^2)$. This is also reduced in practice by the s requirement, as well as by the elimination of one request from consideration at each depth. Counter-examples to exactness of iterated Minimum Demotion KL search have been found, as well as for the uniterated versions. The issue of exactness has not been settled for iterated Full Demotion KL search.

Search Pruning. To gain further insight into the nature of these neighborhoods, two search pruning techniques were implemented and combined with the neighborhoods just discussed. These were **All-Best**, where all of the best improvements were expanded, and **One-Best**, where only one of the best improvements (arbitrarily chosen) is expanded. The KL neighborhoods used only One-Best pruning.

This leaves us with eight search algorithms. Full Demotion or Minimum Demotion combined with All-Best or One-Best gives four, and KL applied once or iteratively to Full Demotion or Minimum Demotion gives the other four.

4.4 Results

An analytic approach to comparing these algorithms would be difficult without further assumptions about the dependency structure and statistical distributions of possible queries. For this reason we elected to compare them empirically. Five dependency structures were used, three being variations of our Thermodynamics feature dimensions, and two constructed artificially. A total of eleven request sets (sets of feature-weight pairs) were selected. Both greedy (sorted by weight) and random initial orderings of these sets were used. The resulting 22 problems were given to each of the eight algorithms described above (as well as to other algorithms used for comparison).

The tests were run on a Macintosh II in Allegro Common Lisp with the compilation optimized for speed. We tabulated and compared the resulting performance in terms of the F-value of the solution found, the number of nodes visited in the search space, and the CPU time required.

Best Performers. In terms of F-value, the Kernighan-Lin applied iteratively with Full Demotion always found the best solution found (so we have no counter-example to exactness). In terms of nodes visited, Minimum Demotion with either One-Best or All-Best pruning visited the fewest nodes in all but one problem: the exception was on a random initial approximation, not used in ExGen. In terms of CPU time, the Minimum Reinsertion solutions performed best (albeit not quite as decisively), requiring at most one tenth of a second. The Full Reinsertion algorithms were close to Minimum Reinsertion in these last two measures, and always did as least as good and sometimes better in terms of the F-value.

Kernighan-Lin. As mentioned, KL with Full Insertion performs quite well in terms of F-value when iterated. However, the KL algorithms usually exceeded a half second and at occasionally a full second of CPU time, so are not suitable for most interactive applications.

KL applied once using Full Demotion yielded as good an F-value as iterative KL on all but three trials (all three were random starts), often in half the time. Minimum Demotion made KL less predictable: on some problems it took longer to find a worse solution, though on most it took less time to find one which was almost as good.

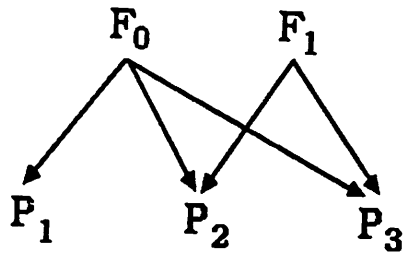
Full vs. Minimum Reinsertion. In general, Minimum Demotion was never better, and was sometimes worse on the F-value. Without a KL restriction, it is possible for the same request to be moved more than once under Minimum Reinsertion. Perhaps the finer granularity of this search is more likely to find local minima.

Expanding All vs. One Best. Expanding a multiple search frontier of all the best neighbors rather than just one did not improve the F-value of solution in these trials. However, it is conceivable that One-Best relies on fortuitous choice of which node to expand.

4.5 Discussion

Our experiments have shown that optimization based on the coarse-grained, static dependencies between feature dimensions can provide some improvement without using up too much time. The current version of ExGen uses Full Demotion with One-Best pruning, the best overall performer given we have ruled out KL on the basis of time, and wish to balance time with quality of solution. However, we recommend consideration of Full Demotion KL neighborhoods (either iterative or single) in applications with more time to spare.

Consider now how `precludes` is related to `protect`. We could take these to be identical. Given ExGen's algorithms, if a feature f is on `protect`(f_0), the only way a request involving f will be satisfied once involving f_0 is satisfied is by a fortunate coincidence of the requested value with the actual. Note that under this definition of `precludes`, feature-based incompatibility is neither sufficient nor necessary for value-based incompatibility. A request which is "precluded" under feature-based incompatibility may yet be satisfied fortuitously, as just mentioned. A request which is not feature-precluded may yet be value-precluded, since some f_1 *not* on `protect`(f_0) may decompose into primitive feature dimensions all of which are in that protected set:



We could tighten the definition of `precludes` to cover this latter case, by saying that f_1 is in `precludes`(f) iff all the primitives in `protect`(f_1) are in `protect`(f). Under these conditions, we know that `to-modify`(f_1) cannot succeed if `protect`(f) are all protected. This necessary condition for value-preclusion is the definition of `precludes` used in ExGen.

The utility of this constraint list reordering depends on what examples are available in the Example Base and on the nature of the dependencies between feature dimensions, as well as on the queries involved. For this reason, the use of this reordering is optional. [17]

[17] Toggled by `*use-local-search*`.

5 Conclusions

The constraint-based approach of ExGen makes it appropriate for embedding in client programs which generate constraints from a collection of possibly competing sources to be weighed off against each other. For instance, in an interactive system for tutoring concepts through the presentation of examples Woolf et al. [1988] utilize a collection of knowledge sources called "example generation specialists" to generate a set of requests for features of the next example. These specialists base their recommendations on the state of the student model (encoding the system's understanding of how the student is doing) and the discourse model (encoding the history of the interaction). Weighting of the requests they generate is guided by the weights placed on the specialists themselves by the current tutoring strategy.

This illustrates the use of ExGen as a kernel program, around which more expressive but application specific example generation "shells" may be constructed. In such a shell, we may write example generation specialists which encode heuristics such as "incrementally generalize the student's understanding by gradually making the examples more complex". This is translated by the specialists into requests that change values along pedagogically relevant feature dimensions. These dimensions in turn automatically translate these requests into modifications of the primitive dimensions they are derived from, as described in this report. Thus, the person encoding pedagogical knowledge may express that knowledge at an appropriate level of abstraction, and leave it to ExGen to realize it at the level of primitive feature dimensions.

Two inadequacies of the ExGen kernel are its expressiveness and its ability to satisfy a consistent yet interacting set of constraints. The expressiveness problem arises because, in spite of the somewhat enriched language parsing provides (Section 3.4), ExGen ultimately deals with constraints expressed as disjunctions of values. The satisfaction problem arises because the Modifier is over-protective of satisfied requests. It protects *values* on feature dimensions when *relationships between values* are of interest. For example, currently ExGen cannot perform modification to achieve a requested `volume` if a previous request for a `type` has been met, resulting in the protection of `length`, `width`, and `height`. If `type` had been protected by saying, for example, that `(= length width height)`, then `to-modify(volume)` could still achieve one of its requested values through equal changes to each of the three dimensions.

As illustrated above, expressiveness may be increased by building application-specific shells around the ExGen kernel. However, we have found that even with the increased expressiveness these afford, there remains considerable work in identifying and encoding the knowledge used to map situational parameters from the user

and discourse models to selection of appropriate examples expressed as weighted constraints. Our current work is focusing on the satisfaction problem, though improved expressiveness may come as a by-product. We hope to redesign ExGen to propagate constraints expressed in terms of relationships between values rather than disjunctions of values themselves. The primary question to be addressed is determining what restrictions on expressiveness in the constraint language are required for goal protection based on relationships to be computationally feasible in interactive applications.

6 References

Ashley, K. & Rissland, E. (1987). Creating Neighborhoods of Cases: Projections Through a Case Space; IJCAI-87.

Bareiss, R., Porter, B., & Wier, C. (1987). Protos: An Exemplar-Based Learning Apprentice; *Proceedings of the Fourth International Workshop on Machine Learning,* U. C. Irvine. Published by Morgan Kaufmann, pp. 12-24.

Brown, D., Clement, J., and Murray, T. (1986). Tutoring Specifications for a Computer Program Which Uses Analogies to Teach Mechanics; Cognitive Processes Research Group Working Paper, *Department of Physics, University of Massachusetts,* Amherst, MA.

Doyle, J. (1979). A Truth Maintenance System; Artificial Intelligence, 12(3), pp. 231-272.

Lakatos, I. (1976). *Proofs and Refutations* , Cambridge University Press.

Lin, S. & Kernighan, B. (1973). An efficient heuristic for the traveling salesman problem; *Operations Research* 21, pp. 498-516.

Murray, T., Schultz, K., Clement, J., & Brown, D. (in press). An Analogy-Based Computer Tutor for Remediating Physics Misconceptions; *AI & Education.*

Papadimitriou, C. H., & Steiglitz, K. (1982). *Combinatorial Optimization.* Englewood Cliffs, N.J.: Prentice Hall.

Polya, G. (1973). *How To Solve It.* Second Edition, Princeton University Press, New Jersey.

Rissland, E. (1980). Example Generation; *Proceedings Third National Conference of the Canadian Society for Computational Studies of Intelligence,* Victoria, B.C.

_____ **(1984).** The Ubiquitous Dialectic; *Proceedings of the European Conference on Artificial Intelligence, September 1984,* Pisa, Italy. Published by North-Holland.

_____ **(1984).** Examples and Learning Systems; *Adaptive Control of Ill-Defined Systems*, O. Selfridge, E. Rissland, & M. Arbib, eds.; Plenum.

_____ **(1988).** Example Selection: The Underlying Issues; to appear in *International Journal of Man-Machine Studies.*

Rissland, E. & Ashley, K. (1987). HYPO: A Case-Based Reasoning System; *IJCAI-87.*

Sussman, G. & Steele, G. (1980). CONSTRAINTS — A language for expressing almost-hierarchical descriptions; *Artificial Intelligence* 14(1), pp. 1-39.

Tennyson, R. (1973). Effect of negative instances in concept acquisition using a verbal learning task; *Journal of Educational Psychology* 64(2), pp. 247-260.

Tennyson, R., Steve, M., & Boutwell, R. (1975). Instance Sequence and Analysis of Instance Attribute Representation in Concept Acquisition; *Journal of Educational Psychology* 67(6), pp. 821-827.

Tennyson, R., Woolley, R., & Merrill, D. (1972). Exemplar and nonexemplar variables which produce correct concept classification behavior and specified classification errors; *Journal of Educational Psychology* 63(2), pp. 144-152.

Wall & Rissland, E. (1982). Scenarios as an Aid to Planning; *AAAI-82*, Pittsburgh, August 1982, pp. 176-180.

Winston (1975.) Learning Structural Descriptions from Examples; *Psychology of Computer Vision*, Winston, ed. Published by Addison-Wesley.

Woolf, B., Murray, T., Suthers D., and Schultz, K. (1988). Primitive Knowledge Units for Tutoring Systems; *Proceedings of the International Conference on Intelligent Tutoring Systems*, Montreal.

Woolf, B., Suthers, D., & Murray, T. (1988). *Discourse Control for Tutoring: Case Studies in Example Selection.* Forthcoming COINS Technical Report, Computer and Information Science, University of Massachusetts, Amherst MA.

A Using ExGen

Obtaining Sources

Local users may find sources and the LaTeX source for this document on the Vaxen in the directory `cai$disk:[cai.modules.exgen]`. Outside users please contact:

Dan Suthers
Department of Computer and Information Science
Lederle Graduate Research Center
University of Massachusetts
Amherst, MA 01003
(413) 545-0582
`suthers@cs.umass.edu`

Mail us either a $5\frac{1}{4}$" floppy formatted under MS-DOS, or a $3\frac{1}{2}$" diskette formatted for the Macintosh. We can also make tapes for TI Explorers or HP 9000 Unix machines.

Loading ExGen

1. Edit the file whose path name will be bound to `*feature-dimensions*`, to define all your feature dimensions using the `feature-dimension` macro.
2. Edit the file EXGENDEFS to define the representation of an example, by adding a slot for each of the feature dimensions defined above.
3. Edit the file whose path name will be bound to `*examples*` to create your seed example base, using the `example` macro.
4. Edit the file EXGEN to contain the correct path names.
5. Enter lisp and load the file EXGEN. It will load the rest.

Obtaining Documentation

All exported variables, functions, and macros are documented for access by the documentation function. See also the file EXGENDOC, which contains the same information.